Rendering tridimensionale: metodi numerici, analitici e probabilistici

Massimo A. Picardello

Dipartimento di Matematica, Università di Roma "Tor Vergata", Via della Ricerca Scientifica, 00133 Roma

Email address: picard@mat.uniroma2.it

Indice

Parte 1

Metodi classici

CAPITOLO 1

Due metodi classici di rimozione di aree nascoste: $z-$buffer e ray tracing

1.1. Metodi di rimozione di aree nascoste: precisione di immagine o precisione di oggetto

1.1.1. Rimozione delle facce posteriori (*back face culling*). Quando si rende una scena tridimensionale proiettandola su un piano di visuale, le facce posteriori degli oggetti della scena vengono coperte da quelle anteriori, e quindi, almeno se la scena consiste di solidi opachi, non possono essere visualizzate (stiamo qui facendo l'ipotesi usuale che tutti gli oggetti siano solidi chiusi, altrimenti una faccia posteriore potrebbe essere visibile da dietro). Pertanto è opportuno rimuoverle dalla modellazione della scena prima di passarle al renderer, in maniera da evitare una mole di calcoli inutili. Questo procedimento di rimozione delle facce posteriori (back face culling) è estremamente semplice: è sufficiente identificare le facce posteriori, che sono quelle il cui versore normale esterno $\boldsymbol{N}$ forma un angolo di più di 90 gradi con la direzione dell'osservatore, che indichiamo con $\boldsymbol{V}$ (si tratta del vettore applicato dal punto di osservazione che va verso il punto osservato). Il versore $\boldsymbol{N}$ è fornito dal modellatore: quindi, per ogni faccia dell'oggetto, basta considerare un suo punto, calcolare il vettore $\boldsymbol{V}$ e poi verificare se il prodotto scalare $\langle \boldsymbol{N}, \boldsymbol{V} \rangle$ ha segno positivo o negativo: nel secondo caso abbiamo una faccia posteriore, e la rimuoviamo.
Normalmente, il modellatore descrive la scena tramite una rete di maglie poligonali: i vertici dei poligoni sono punti campionati sulle superficie della scena. Molto spesso questi poligono sono triangoli, perché, dati tre punti campionati, il triangolo da essi sotteso è necessariamente una figura piana, mentre un poligono con più vertici in generale non lo è. Il versore normale al triangolo è quindi il versore normale al piano che lo contiene: ma è compito del modellatore specificare il verso del versore normale esterno (ossia uscente dal solido di cui il triangolo è una faccetta). Talvolta il modellatore, invece che il verso di tale versore, fornisce il senso di percorrenza del bordo del triangolo (si limita ad assegnare l'ordine ciclico dei tre vertici). In tal caso, il verso positivo (ossia uscente verso l'esterno) del versore normale è quello che determina il semipiano visto dal quale il verso di percorrenza del bordo del triangolo è antiorario, ed il versore normale esterno si calcola immediatamente formando il prodotto vettoriale di due lati consecutivi del triangolo (intesi come vettori), e poi normalizzando.

1.1.2. Precisione di immagine e precisione di oggetto. Gli algoritmi per la determinazione delle superficie visibili possono essere divisi in due categorie: algoritmi a precisione d'immagine ed algoritmi a precisione d'oggetto. I primi, per ogni pixel nell'immagine, determinano quale degli oggetti toccati dal raggio proiettore che parte dal centro di osservazione e passa attraverso quel pixel sia il più vicino all'osservatore; i secondi, per ogni oggetto della scena, determinano quali parti dell'oggetto siano visibili. Gli algoritmi a precisione di immagine sono solitamente effettuati alla risoluzione dello schermo, e determinano la visibilità in

ogni pixel, mentre gli algoritmi a precisione d'oggetto sono effettuati al livello di precisione con cui ogni oggetto è definito, e determinano la visibilità di ogni oggetto. Poiché i calcoli a precisione ad oggetto sono effettuati senza tener conto della particolare risoluzione dello schermo, essi devono essere seguiti da un processo in cui gli oggetti sono effettivamente resi alla risoluzione desiderata; se viene cambiata la grandezza dell'immagine finale, deve essere ripetuto solo quest'ultimo passo, poiché la geometria di ogni oggetto visibile è rappresentata con un database con la risoluzione completa di quell'oggetto. Invece, usando un algoritmo a precisione d'immagine, se vogliamo allargare un'immagine e vederne maggiori dettagli, dobbiamo rifare i calcoli per la determinazione delle superficie visibili in quanto precedentemente erano stati effettuati usando una risoluzione minore.

1.1.3. Metodi di rimozione a precisione di immagine. I metodi di rimozione delle aree nascoste a precisione di immagine operano nel modo seguente. Per ogni pixel nella parte del piano di proiezione su cui si vuole rendere l'immagine prospettica, si determina se e quale degli n oggetti della scena è visibile attraverso quel pixel. Per far questo, per ogni pixel nell'immagine si trova quale sia l'oggetto più vicino all'osservatore che viene intersecato dal raggio uscente dalla posizione dell'osservatore e passante per il centro del pixel (detto raggio proiettore); il pixel viene poi colorato col colore che quell'oggetto ha nel punto di intersezione col raggio. Ecco lo pseudocodice:

```
for (ciascun pixel nell'immagine)
{
si determina l'oggetto piu' vicino all'osservatore che
viene intersecato dal raggio di proiezione dal punto di
visuale che passa per quel pixel;

si colora il pixel col colore corrispondente a questo
oggetto}
```

Questo tipo di metodo richiede di trovare, per ciascun pixel, le intersezioni del corrispondente raggio di proiezione con tutti gli n oggetti per determinare quale è il più vicino all'osservatore. Se la risoluzione di immagine consiste di p pixels, la complessità del calcolo è proporzionale a np. Si osservi che per un'immagine tipica di risoluzione fra VGA (1024 x 768 pixel) a HD, p è dell'ordine di qualche milione. Di solito il numero n di oggetti della scena è molto minore.

1.1.4. Metodi di rimozione a precisione di oggetto. I metodi a precisione di oggetto paragonano gli oggetti della scena direttamente fra loro, eliminando interamente gli oggetti o le parti di essi che non sono visibili dal punto di vista dell'osservatore. In altre parole, di ogni oggetto della scena si determinano quelle parti la cui vista non è impedita da altre parti di esso o da altri oggetti, e si disegnano queste parti nel colore appropriato. Ecco lo pseudocodice:

```
for (ciascun oggetto nella scena)
{
si determinano le sue parti la cui visuale dal punto di
osservazione non e' ostruita da altre sue parti o da
altri oggetti;
```

```
si colorano queste parti visibili con il colore
dell'oggetto;

si proiettano le parti colorate sul viewport}
```

Per eseguire il rendering a precisione di oggetto dobbiamo quindi confrontare ognuno degli n oggetti con sé stesso e con gli altri oggetti, scartando le porzioni non visibili. Il tempo di calcolo quindi è proporzionale a n^2. Sebbene questo approccio potrebbe richiedere meno confronti di quello a precisione di immagine se $n < p$ (il che è spesso vero nella pratica), i suoi confronti sono di solito più complessi e richiedono più tempo, e pertanto esso è di solito più lento e più complesso da implementare.
Gli algoritmi a precisione d'immagine sono tipicamente elaborati alla risoluzione del monitor, perché determinano la visibilità per ogni pixel. Gli algoritmi a precisione d'oggetto sono invece elaborati alla precisione con cui ogni oggetto è definito nella sua modellazione matematica, e con questa precisione ne determinano la visibilità. Dal momento che i calcoli della precisione d'oggetto sono effettuati senza riferimento ad una particolare risoluzione del monitor, devono essere seguiti da un passaggio in cui gli oggetti di cui si è determinata la visibilità sono poi disegnati sul monitor alla risoluzione di quest'ultimo. Solo quest'ultimo passo dipende dalla risoluzione del monitor, e quindi dal numero di pixel dell'immagine, e solo esso deve essere ricalcolato se si desidera cambiare la risoluzione dell'immagine (per esempio se si cambia il numero di pixel visualizzati dalla scheda grafica, oppure la scala con cui viene disegnata l'immagine, e quindi la sua grandezza sul monitor).
Invece il cambiamento della risoluzione o della scala in un algoritmo di rendering a precisione di immagine richiede di effettuare da capo tutto il calcolo. Ad esempio, immaginiamo di voler ingrandire un'immagine prodotta da un tale algoritmo. I calcoli della superficie visibile sono stati inizialmente sviluppati alla risoluzione originale più bassa, e devono essere ripetuti se vogliamo rivelare ulteriori dettagli. Nell'impostazione di pensiero suggerita dal campionamento, possiamo pensare agli algoritmi a precisione di oggetto come operanti sui dati dell'oggetto originale, modellato in termini di forme continue, e gli algoritmi a precisione d'immagine come operanti su dati campionati. Così gli algoritmi a precisione d'immagine sono soggetti ad aliasing nel calcolo della visibilità, mentre gli algoritmi a precisione d'oggetto no.

1.2. $z-$buffer

L'algoritmo di $z-$buffer (o depth-buffer) a precisione di immagine, sviluppato da Catmull in [**7**], è uno degli algoritmi per la determinazione delle superficie visibili più semplici da implementare sia a livello di software sia a livello di hardware. Richiede due spazi di storage (*buffers*): un frame buffer F in cui sono memorizzati gli identificativi numerici dei triangoli (o più in generale poligoni) ed in particolare i loro valori di colore, ed uno $z-$buffer Z, con lo stesso numero di dati, in cui per ogni pixel viene memorizzata la profondità del punto della scena osservato attraverso quel pixel. Lo $z-$buffer è inizializzato al valore zero, che rappresenta la profondità del piano posteriore di clipping, ed il frame buffer è inizializzato al colore dello sfondo. Per convenzione, l'asse z che indica la profondità è preso con verso crescente verso l'osservatore. Quindi il più grande valore che può essere memorizzato nello $z-$buffer rappresenta la profondità del piano anteriore di clipping.

Questo algoritmo elabora una scena modellata con maglie poligonali (di solito triangolari, il che offre il vantaggio che i poligoni delle maglie sono piani). I poligoni sono proiettati uno

dopo l'altro sul viewport. A questo fine, vengono proiettati i loro vertici (diciamo i tre vertici, nel caso di triangoli), e dopo il viewport viene scandito riga dopo riga per determinare quali pixel sono interni al triangolo sotteso di tre vertici proiettati (questa procedura si chiama *scan conversion*. Per ciascun pixel interno a questo triangolo (o poligono) proiettato, viene memorizzata la profondità z del punto del poligono che si proietta su quel pixel, ottenuta tramite un ovvio procedimento interpolativo durante la scan conversion (si veda la Sottosezione 1.2.2 in seguito).

I poligoni vengono scanditi seguendo un ordine arbitrario; durante questo processo, se un punto appartenente al poligono che viene scandito si proietta sul pixel di coordinate (x,y) e la sua profondità è maggiore (ossia più vicina all'osservatore) di quella che era precedentemente memorizzata nei buffer, allora il suo colore e la sua profondità prendono il posto di quelli precedentemente memorizzati nei rispettivi buffer. In pratica, per ogni pixel vengono memorizzati il colore (o meglio, l'indice del poligono) e la profondità del punto appartenente alla scena fino a quel momento più vicino all'osservatore.
Non sono necessari confronti fra punti dello spazio tridimensionale. L'intero processo consiste nel ricercare il più grande valore della profondità dei punti della scena che si proiettano su ciascun dato pixel. Lo $z-$buffer ed il frame buffer registrano per ogni (x,y) solo l'informazione associata al più alto valore z (il più vicino all'osservatore) scandito precedentemente. Perciò, i poligoni appaiono sulla scena nell'ordine in cui vengono processati (ne appaiono solo le parti più frontali delle precedenti).

1.2.1. Pseudocodice dello $z-$buffer.

```
void zBuffer ( void )
{
   int x, y;

   for ( y = 0; y < YMAX; y++ )   // Inizializziamo il frame buffer e lo z-buffer:
   {  for ( x = 0; x < XMAX; x++ )
      {  WritePixel ( x, y, Colore_sfondo );
         WriteZ ( x, y, 0 );
      }
   }

   for ( ogni poligono )         // Disegna i poligoni
   {
      for ( ogni pixel contenuto nella proiezione del poligono )
      {  double pz = z del poligono alle coordinate ( x, y );

if ( pz >= ReadZ ( x, y ) )  // Il punto e' finora il piu' vicino (o vicino quanto
    il precedente) all'osservatore in quel pixel
         {
            WriteZ ( x, y, pz );
            WritePixel ( x, y, colore del poligono
                               alle coordinate ( x, y ) );
         }
      }
```

```
   }
}
```

1.2.2. Scan conversion e calcolo incrementale della profondità. Traiamo vantaggio dalla coerenza di profondità (depth coherence): possiamo semplificare il calcolo della profondità z per ogni punto di ogni scan line sfruttando il fatto che i triangoli sono piani. Normalmente, per calcolare la z, dovremmo risolvere per la variabile z l'equazione che caratterizza il piano dove giace il triangolo da proiettare, che è del tipo $Ax + By + Cz + D = 0$: da qui $z = z(x, y) = -(Ax + By + D)/C$. Notiamo che, una volta scandito un punto di un triangolo che si proietta sul pixel, diciamo (x, y), il pixel subito a destra o a sinistra di esso ha coordinate $(x \pm \Delta x, y)$, dove Δx è lo spessore dei pixel del monitor. Pertanto, se questo pixel appartiene alla proiezione prospettica dello stesso triangolo, su di esso il valore della profondità deve essere

$$z(x + \Delta x, y) = z - \frac{A}{C}\,\Delta x. \tag{1.2.1}$$

Quindi, finché non usciamo dalla proiezione di quel triangolo, i valori della profondità variano su ciascuna riga in maniera incrementale, e si calcolano grazie ad una addizione ed ad una moltiplicazione (quest'ultima sempre per la stessa costante A/C, che dipende solo dal poligono che stiamo scandendo: la divisione per C si fa una volta per tutte all'inizio e poi non si deve fare alcuna altra divisione, solo moltiplicazioni e somme).

Supponiamo che i tre vertici del triangolo che stiamo scandendo si proiettino sui punti del viewport di coordinate rispettive $\boldsymbol{q}_1 = (x_1, y_1)$, $\boldsymbol{q}_2 = (x_2, y_2)$ e $\boldsymbol{q}_3 = (x_3, y_3)$, e che di questi tre punti proiettati il punto (x_1, y_1) sia quello più in alto nel viewport (ossia la sua ordinata y_1 sia la *minore* delle tre: rammentiamo che nella scansione seguiamo l'ordine lessicografico, e l'altezza y aumenta quando procediamo dalle righe alte a quelle basse).

Per le righe più in alto dell'altezza y_1 non c'è niente da scandire. Alla riga y_1 la scansione restituisce solo un punto, il vertice proiettato $\boldsymbol{q}_1$, ossia il punto di ascissa x_1. Ora scendiamo alla riga successiva. Le righe distano fra loro di una distanza fissa Δy, che è l'altezza dei pixel del monitor. Il lato di sinistra della proiezione del triangolo congiunge i punti $\boldsymbol{q}_1$ e, diciamo, $\boldsymbol{q}_2$, mentre il lato destro va da $\boldsymbol{q}_1$ a $\boldsymbol{q}_3$: le loro equazioni sono, rispettivamente,

$$x_\pm - x_1 = (y_\pm - y_1)\,\frac{x_1 - x_j}{y_1 - y_j}$$

dove $j = 2, 3$. Denotiamo le pendenze di queste rette con $C_\pm$, ovvero

$$C_- = \frac{y_2 - y_1}{x_2 - x_1}\,,$$

$$C_+ = \frac{y_3 - y_1}{x_3 - x_1}\,.$$

Allora è semplice calcolare il punto iniziale ed il punto finale della proiezione sulla riga successiva alla prima: essi sono, rispettivamente, $x_\pm = x_1 + \Delta y/C_\pm$. Analogamente, ad ogni riga successiva le ascisse di inizio e di fine della proiezione si determinano con questo metodo incrementale, dove l'unica divisione è sempre per la costante C_- per l'incremento a sinistra e C_+ per l'incremento a destra. Per interpolazione, lo stesso succede alla variazione della

profondità nei punti iniziale e finale della proiezione su ciascuna riga: all'altezza y_s, con $y_1 \leqslant y_s \leqslant \min\{y_2, y_3\}$, troviamo

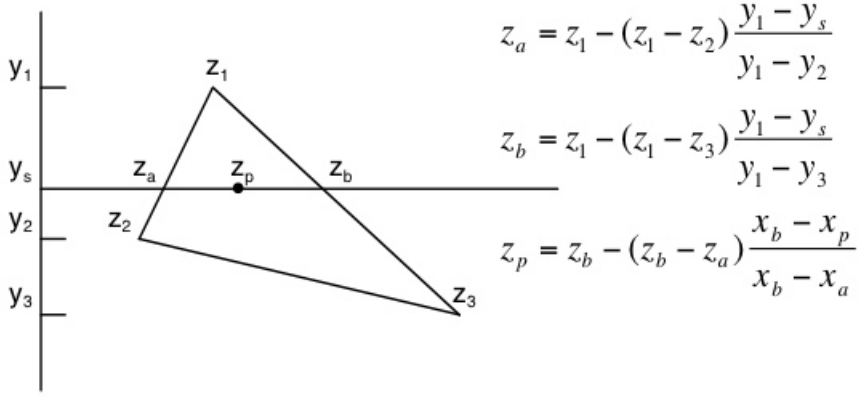
$$z_-(s) = z_1 - (z_1 - z_2)\,\frac{y_1 - y_s}{y_1 - y_2}\ ,$$

$$z_+(s) = z_1 - (z_1 - z_3)\,\frac{y_1 - y_s}{y_1 - y_3}\ .$$

Allora, in base a (1.2.1), la profondità nel generico punto $\boldsymbol{q}_p(s)$ all'interno della riga ad altezza y_s della proiezione di questo triangolo vale

$$z_p(s) = z_-(s) - (z_-(s) - z_2 + (s))\,\frac{x_-(s) - x_p(s)}{x_-(s) - x_+(s)}\ ,$$

e quindi si calcola anch'essa in modo incrementale, al variare del punto di indice p sulla riga di indice s, con l sola moltiplicazione per il fattore $(z_-(s) - z_2 + (s))/(x_-(s) - x_+(s))$, il quale, sulla riga s, è costante ed indipendente dal punto p. Riassumendo, abbiamo calcolato la profondità interpolando le coordinate z dei vertici del poligono lungo coppie di lati (ossia di spigoli della modellazione tridimensionale a maglie), e dopo lungo ogni scan line.



1.2.3. Implementazione, precisione numerica ed aliasing. Lo $z-$buffer è spesso implementato a livello hardware con valori interi a 16 bit o 32 bit, mentre le implementazioni a livello software usano di solito valori in virgola mobile.

Anche se uno $z-$buffer a 16 bit offre un'adeguata gamma di valori per molte applicazioni CAD/CAM, 16 bit non danno abbastanza precisione per rappresentare ambienti in cui oggetti definiti al millimetro sono posti lontani chilometri. A peggiorare le cose, se viene usata una proiezione prospettica, la compressione risultante dalla prospettiva dei valori z più distanti può causare problemi per un corretto ordinamento in profondità e nell'intersezione tra gli oggetti più lontani. Infatti, in seguito ad insufficiente precisione dello $z-$buffer, due coppie di punti con la stessa differenza inella variabile z potrebbero, dopo la proiezione, apparire una più avanti dell'altro se presi vicini al viewplane, ma alla stessa profondità se lontani (e nel secondo caso quale dei due entri nello $z-$buffer dipende, in maniera casuale, solo dagli arrotondamenti e troncamenti numerici).

La precisione finita dello $z-$buffer è responsabile di un altro problema: l'aliasing. Di solito gli algoritmi di scan-conversion rendono separatamente i due gruppi di pixel quando disegnano

la parte comune di due spigoli della modellazione a maglie che appartengono alla stessa retta ma che hanno estremi diversi. Alcuni di quei pixel condivisi potrebbero vedersi assegnati valori di z leggermente diversi a causa della possibile inaccuratezza numerica nell'effetturare i calcoli per l'interpolazione delle z. Questo effetto può diventare evidente per gli spigoli condivisi delle facce di un poliedro. Alcuni dei pixel visibili di uno spigolo potrebbero risultare come appartenenti ad una faccia ed altri ad un'altra adiacente: in tal caso lo spigolo viene colorato a tratti con i due corrispondenti colori, e quindi assume una clorazione a bande. Questo problema può essere mitigato inserendo dei vertici aggiuntivi, per ridurre la lunghezza degli spigoli e quindi il sommarsi degli errori di approssimazione numerica.

1.2.4. Vantaggi dello $z-$buffer, accorgimenti. Lo $z-$buffer non richiede che gli oggetti della scena siano necessariamente poligoni. Lo stesso algoritmo funziona con maglie di forma qualsiasi (ma se le maglie non sono facce piane si perde il vantaggio della interpolazione incrementale per l'effettuazione della scan conversion ed il calcolo della profondità, illustrato nella Sottosezione 1.2.2). Quindi, uno dei suoi punti di forza è che può essere usato per rendere ogni tipo di oggetto se per ogni punto nella sua proiezione può essere determinato il suo valore z e la sua tonalità di colore (se quest'ultima informazione èassegnata dal modellatore punto per punto).

Lo $z-$buffer può essere usato in maniera vantaggiosa anche dopo aver già reso un'immagine. Dato che è l'unica struttura dati usata propriamente dall'algoritmo per la determinazione delle superficie visibili, può essere salvato con l'immagine e usato successivamente per essere incorporato in altri oggetti di cui vogliamo calcolare la profondità z, ad esempio quando, in una animazione, vogliamo aggiungere nuovi oggetti che entrano nella scena osservata.

L'algoritmo può essere inoltre codificato in modo che nel corso della scan conversion di alcuni oggetti specifici non venga modificato il contenuto dello $z-$buffer. In tal modo, un oggetto specifico può essere disegnato su un viewplane separato sovrapposto a quello standard che ha le superficie nascoste rimosse, e quindi può essere cancellato senza per questo toccare i contenuti dello $z-$buffer. Perciò, un semplice oggetto, come per esempio un righello o un puntatore, può essere mosso lungo l'immagine nelle coordinate x, y del viewplane e nella profondità z, e servire da cursore 3D che oscura gli oggetti nell'ambiente o ne viene oscurato: infatti, per ogni fotogramma dell'animazione, basta confrontare il valore z del punto del righello proiettato sul pixel x, y con il valore dello $z-$buffer degli altri oggetti, e disegnare o no quel pixel con il colore del righello a seconda del risultato di questo confronto.

In [**40**] l'algoritmo di $z-$buffer viene adattato per la costruzione di oggetti nel CSG (Constructive Solide Geometry), un metodo per descrivere la geometria di una scena complessa applicando un insieme di operazioni booleane (unione, intersezione, complemento, differenza) a primitive grafiche, ossia ad oggetti geometrici specifici predefiniti. Ogni punto che si trova sulla proiezione di una superficie viene disegnato solo se, sul pixel di proiezione, è il più avanti in profondità (valore z massimo), ma viene inserito nel solido unione (od intersezione) di due primitive della scena solo se appartiene ad uno (rispettivamente, ad entrambi) i due solidi. Analogamente si procede nel caso di complemento o differenza.

Una estensione che permette di trattare efficientemente le operazioni booleane ma anche la parziale trasparenza è stata introdotta da Atherton [**2**]. Invece di memorizzare solo il punto con la z più vicina ad ogni pixel, questa estensione dell'algoritmo salva una lista di tutti i punti che si proiettano su quel pixel, ordinati secondo la z, accompagnati da un identificativo di ogni superficie, al fine di formare un object-buffer. Come mostrare l'immagine viene stabilito

da regole che dipendono dalle proprietà dei materiali coinvolti: ad esempio, processando la lista dei punti che insistono sullo stesso pixel, di ogni pixel, si possono ottenere in un colpo solo vari effetti, come la trasparenza, il clipping e le operazioni booleane, senza il bisogno di riscansionare e riconvertire gli oggetti.

1.2.5. Differenze fra $z-$buffer e ray tracing. Il ray tracing, che verrà spiegato in dettaglio nella prossima Sezione 1.4, per ogni pixel traccia un raggio che parte dal punto dell'osservatore ed interseca questi raggi di proiezione con ogni oggetto presente sulla scena al fine di determinare l'oggetto più vicino, e dunque il colore di quel pixel.

L'algoritmo di $z-$buffer approssima un oggetto tramite un set di valori z che misurano la profondità) lungo i raggi di proiezione che intersecano quell'oggetto. Invece il ray tracing approssima un oggetto tramite l'insieme dei punti di intersezione (in tre dimensioni) lungo ogni raggio proiettore che interseca quell'oggetto.

Per ogni pixel, l'algoritmo di $z-$buffer calcola l'informazione solamente per quegli oggetti la cui proiezione cade su quel pixel, sfruttando la coerenza. Il ray tracing, invece, interseca ogni raggio che parte dall'osservatore con ogni oggetto che si trova sulla scena. Dunque lo $z-$buffer non calcola alcuna dispendiosa operazione di intersezione (come invece fa il ray tracing) poiché esso opera soltanto un confronto fra le coordinate z ottenute nella trasformazione prospettica sul viewplane: queste coordinate gli vengono passate direttamente dalla routine di trasformazione prospettica, ed esso si limita eventualmente ad interpolarne i valori lungo le righe di scansione, sfruttando il calcolo incrementale.

1.3. Esempi sullo $z-$buffer

ESEMPIO 1.3.1. Una scena consiste dei seguenti poligoni, visti frontalmente nel modo in cui si ricoprono. I due rettangoli orizzontali si indicano con 0, 1, dall'alto in basso, quelli verticali con 2, 3, 4 da sinistra a destra. Le profondità dei vertici sono le seguenti:

- 0: vertici di sinistra z= 1, vertici di destra z=0
- 1: vertici di sinistra z= 0, vertici di destra z=1
- 2: vertici alti z= 0, vertici bassi z=1
- 3: vertici alti z= 1, vertici bassi z=1
- 4: vertici alti z= 1, vertici bassi z=0

La scena è illustrata in Figura 1.3.1 .

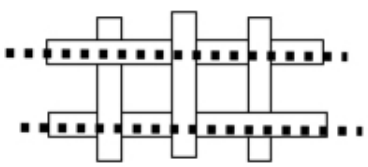


FIGURA 1.3.1. Proiezione sul viewplane della scena dell'Esempio 1.3.1.

Applichiamo il metodo di $z-$buffer scandendo i poligoni nell'ordine ciclico a partire dal poligono il cui indice è 1. Si traccino i grafici dello $z-$buffer uno per volta, dopo la scansione

di ciascun poligono, sui pixel di uno dei due assi orizzontale tratteggiati, diciamo quello più in alto. Il calcolo sull'asse in basso è proposto al lettore come esercizio.

Svolgimento.

La riga di scansione interseca i poligoni 0, 2, 3 e 4. Si tratta di tracciare i grafici dello $z-$buffer (ossia della profondità) limitato alla scan conversion sulla retta tratteggiata, che aggiorniamo via via che viene scandito un nuovo poligono. Per ogni nuovo poligono inseriamo nello $z-$buffer, punto per punto della retta di scansione, il valore massimo fra la profondità del nuovo poligono e quella trovata in base ai poligoni precedenti. Il valore iniziale dello $z-$buffer è 0. Esaminiamo i valori di profondità di ciascuno dei poligoni sulla retta di scansione.

Profondità del poligono 0: la scansione produce il grafico in Figura 1.3.2.

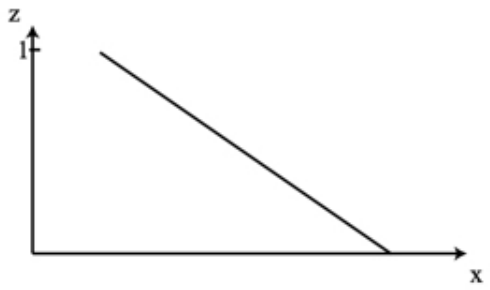


FIGURA 1.3.2. Il grafico della profondità nella scan conversion del poligono 0.

Profondità del poligono 2: la profondità varia da 0 (in alto) a 1, la retta di scansione si trova a circa 4/5 dell'altezza del poligono, quindi lì la profondità vale 0.2, come illustrato in Figura 1.3.3.

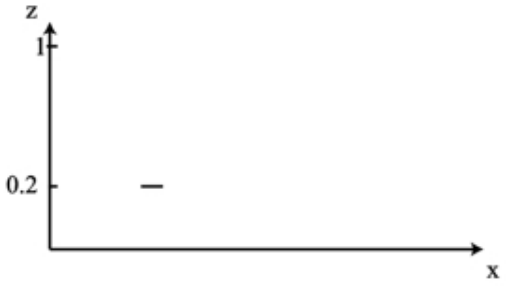


FIGURA 1.3.3. Il grafico della profondità nella scan conversion del poligono 2.

Profondità del poligono 3: esso si trova a profondità costante 1 (si veda la Figura 1.3.4).

Profondità del poligono 4: varia da 1 a 0, e sulla retta di scansione, a circa 4/5 dell'altezza, vale 4/5=0.8. Il grafico è in Figura 1.3.5

Ora scandiamo i poligoni uno dopo l'altro aggiornando ogni volta i grafici, nell'ordine, diciamo, 0,2,3,4.

Sezione dello $z-$buffer dopo la scansione del poligono 0: è uguale al grafico della profondità del poligono 0, ed il suo grafico è in Figura 1.3.6.

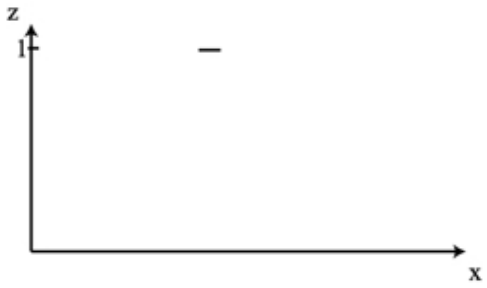


FIGURA 1.3.4. Il grafico della profondità nella scan conversion del poligono 3.

FIGURA 1.3.5. Il grafico della profondità nella scan conversion del poligono 4.

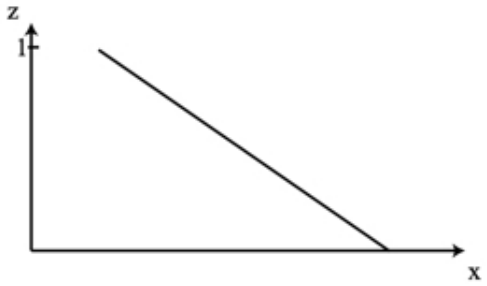


FIGURA 1.3.6. Aggiornamento dello $z-$buffer al passo iniziale (poligono 0).

Dopo aver scandito anche il poligono 2, il grafico rimane lo stesso, perché la profondità del poligono 2 è dappertutto inferiore a quella del poligono 0.

Dopo aver scandito anche il poligono 3, il grafico cambia perché il poligono 3, nel segmento della retta di scansione in cui viene intersecato, ha profondità superiore a quella dello $z-$buffer fino a quel momento, e diventa quello della Figura 1.3.7.

La stessa cosa capita dopo aver scandito anche il poligono 4. Prendendo il massimo fra la profondità del poligono 4, pixel per pixel sulla retta di scansione, e lo $z-$buffer precedente, si ottiene il grafico nella Figura 1.3.8.

Questo è il risultato finale. Si arriva allo stesso risultato finale se si effettua la scansione dei poligoni in qualsiasi altro ordine, ma i grafici alle varie fasi sono diversi. □

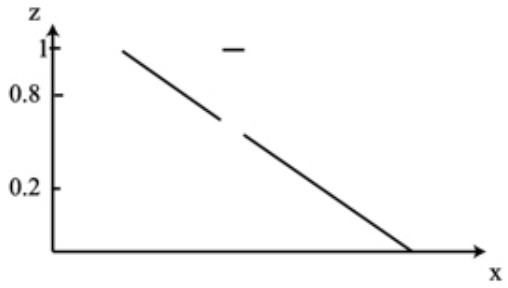


FIGURA 1.3.7. Aggiornamento dello $z-$buffer dopo la scan conversion del poligono 3.

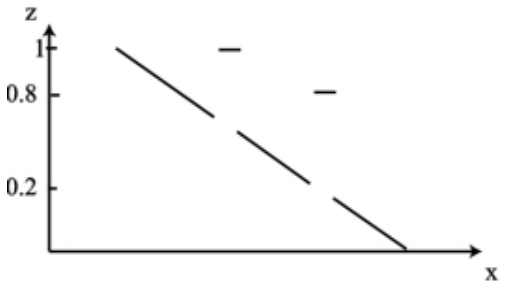


FIGURA 1.3.8. Il grafico finale della profondità, dopo la scan conversion del poligono 4.

ESEMPIO 1.3.2. Una scena viene proiettata sul piano $\{z = 0\}$ sul viewport V dato dal quadrato $[0, 10] \times [0, 10]$, che si pensa suddiviso nei pixel dati dai sottoquadrati di coordinate intere da 0 a 10 (100 pixel in tutto). La scena consiste di due rettangoli, il primo, R_1, con vertici in $(0, 0, -1)$, $(0, 6, -1)$, $(6, 6, -1)$ e $(6, 0, -1)$, ed il secondo, R_2, con vertici in $(3, 0, -3)$, $(3, 6, -3)$, $(9, 6, -1)$ e $(9, 0, -1)$.

(i) Si calcoli lo $z-$buffer sul viewport V nel caso della proiezione ortogonale sul piano $\{z = 0\}$ (si assuma come al solito che lo $z-$buffer sia inizializzato con valori tutti nulli). Si consideri ogni pixel è come ricoperto dalla proiezione di uno dei rettangoli se questa proiezione copre una parte del pixel di area positiva.

(ii) Si consideri ora la proiezione centrale su V con centro di proiezione in $(0, 0, 1)$. Si determini il punto di fuga del fascio di rette parallele ai lati opposti di R_2 ad altezza $y = 0$ e $y = 6$.

(iii) Si calcoli lo stesso $z-$buffer della prima parte del problema quando la proiezione è quella centrale della seconda parte.

Svolgimento.
La parte (i) riguarda la proiezione ortogonale sul piano $\{z = 0\}$. Con questa proiezione ogni punto (x, y, z) finisce in $(x, y, 0)$. Pertanto il primo rettangolo, R_1, occupa i pixel del rettangolo con vertici opposti $(0, 6)$ e $(6, 0)$: questo sottoquadrato riceve quindi, dalla scansione di R_1, valore di $z-$buffer pari alla profondità di R_1 che e' $z = -1$ perche' R_1 giace nel piano $\{z = -1\}$. Il secondo rettangolo, R_2, si proietta nel rettangolo con vertici opposti $(3, 6)$ e $(9, 0)$. Il lato di sinistra proviene da $\{z = -3\}$, quello di destra da $\{z = -1\}$. Le due proiezioni si sovrappongono nel rettangolo di lati opposti $(3, 6)$ e $(6, 0)$: per interpolazione si vede

che in questa intersezione R_2 ha profondità compresa fra -3 e -2. Quindi in questo rettangolo di sovrapposizione lo $z-$buffer ha i valori di R_1, ossia $z=-1$. Il resto della proiezione prospettica di R_2 è il rettangolo di vertici opposti $(6,6)$ e $(9,0)$: esso è coperto solo da R_2 e quindi ovviamente in esso lo $z-$buffer è quello dato da R_2, che varia linesrmente da -2 a -1. Su ogni riga di pixel che verifica $0<y<6$, la z di R_2 si trova per interpolazione lineare da $z=-3$ sul lato sinistro (ossia per $x=3$) a $z=-1$ sul lato destro (ossia per $x=9$). Da 3 a 9 ci sono sei pixel: quindi i valori di z scalano di $2/6=1/3$ nel passare da un pixel a quello subito a destra. Abbiamo quindi $z-$buffer con valore -1 sulle righe da 1 a 6 per $0\leqslant x\leqslant 6$, e valori da -2 a -1 per $6\leqslant x\leqslant 9$. Più precisamente, visto che l'incremento vale $1/3$, abbiamo i tre valori $-2+\frac{1}{3}$, $-2+\frac{2}{3}$, 1 riepettivamente per $x=7$, 8, 9. È però più preciso assegnare ad ogni pixel $6<x<7$, $7<x<8$ e $8<x<9$ il valore di profondità che ha R_2 non all'inizio del pixel ma al suo centro, ed in tal modo si ottengono i tre valori di $z-$buffer $-2+\frac{1}{6}=-\frac{11}{6}$ (per $x=6.5$), $-2+\frac{1}{6}+\frac{1}{3}=-2+\frac{1}{2}=-\frac{3}{2}$ (per $x=7.5$) e $-2+\frac{1}{2}+\frac{1}{3}=-2+\frac{5}{6}=-1+\frac{1}{6}=-\frac{5}{6}$ (per $x=8.5$).
Su tutte e sei queste righe l'andamento orizzontale è identico perché i valori di profondità cambiano orizzontalmente ma sono costanti verticalmente. Ovviamente sulle parti restanti (le righe da 6 a 10 e l'ultima colonna) non insistono poligoni, e quindi il valore dello $z-$buffer resta 0.

Il punto di fuga richiesto nella parte (ii) è quello del fascio di rette parallele ai lati superiore ed inferiore di R_2, che sono diretti da $(9,0,-3)$ a $(3,0,-1)$, e quindi nella direzione di $\mathbf{u}=(6,0,-2)$. Poiché il piano di proiezione $\{z=0\}$ ha versore normale $\mathbf{e}_3$ e la distanza fra il centro di proiezione $(0,0,1)$ ed il piano di proiezione è $d=1$, dal Teorema 16.6.1 sul punto di fuga di un fascio nella teoria della prospettiva centrale segue che il punto di fuga è

$$\mathbf{f_u}=\mathbf{c}+\frac{d}{\langle\mathbf{e}_3\,,\,\mathbf{u}\rangle}\mathbf{u}=(0,0,1)-\frac{1}{-2}(6,0,-2)=(3,0,0)$$

e quindi è il punto del piano di proiezione di coordinate $x=3, y=0$ (in effetti il valore della y lo sapevamo già perché l'immagine prospettica sia il centro di proiezione sia il lato basso di R_2 giacciono sul piano $\{y=0\}$ e quindi deve giacervi anche l'immagine prospettica di tale lato ed il punto di fuga).

Per quanto riguarda la parte (iii), consideriamo il lato destro di R_2, che è un segmento verticale di estremi $y=0$ e $y=6$ giacente sulla retta $x=9$, $z=-1$. Anche senza applicare la matrice della trasformazione prospettica, il fatto che il piano di visuale sia a metà strada fra il suo parallelo $\{z=-1\}$ che contiene il segmento e l'altro suo parallelo $\{z=1\}$ che contiene il centro di proiezione rende geometricamente evidente che il segmento tracciato dall'estremo basso $(9,0,-1)$ al centro di proiezione $(0,0,1)$ interseca il piano di proiezione $\{z=0\}$ proprio al suo punto centrale $(9/2,0,0)$, mentre il segmento dall'estremo alto $(9,6,-1)$ al centro $(0,0,1)$ interseca il piano $\{z=0\}$ di nuovo al punto centrale, che ora è $(9/2,3,0)$. Quindi i due pixel corrispondenti sono quelli relativi alle coordinate x,y $(4.5,0)$ e $(4.5,3)$, quindi a distanza 3 fra loro sulla verticale $x=4.5$: pensati come sottoquadrati, questi due pixel sono rispettivamente $[4,5]\times[0,1]$ e $[4,5]\times[3,4]$. Ora, i lati verticali (sinistro e destro) di R_2 dopo la proiezio e prospettica rimangono verticali (ovvero con x costante) perché le rette $x=$ costante sono parallele al piano di proiezione e pertanto il loro punto di fuga è all'infinito in base al Teorema sul punto di fuga di un fascio provato nel Capitolo sulla Prospettiva. Abbiamo visto che il lato di destra viene proiettato sulla retta $x=4.5$ del piano $\{z=0\}$: esattamente lo stesso argomemento mostra che il lato di sinistra viene proiettato sulla retta $x=1.5$. Con la

stessa facilità di prima lo stesso argomento mostra che gli estremi inferiore e superiore della proiezione del lato di sinistra sono $y = 0$ e $y = 6/4 = 1.5$ (il secondo è perché questa volta il lato di sinistra si trova tre volte più distante da V di quanto non sia il centro di proiezione, e quindi la divisione prospettica diventa ora di un fattore 4 invece che 2). Ad ogni modo, poiché conosciamo i punti estremi del primo lato ed il punto di fuga dei lati superiori, conosciamo le equazioni delle due rette su cui tali lati giacciono. Da qui ricaviamo i punti del quadrilatero costituito dalla proiezione di R_2.
La proiezione di R_1 invece è un rettangolo, perché i suoi lati giacciono sul piano $\{z = -1\}$ che è parallelo al piano di proiezione, e quindi lati opposti restano paralleli anche dopo la proiezione. La stessa proporzione di prima ci fornisce i loro vertici proiettati: sono $(0, 0)$, $(0, 3)$, $(3, 3)$, $(3, 0)$. Come nella parte (i), la proiezione di R_1 copre (ossia ha z maggiore di) quella di R_1 nell'area di sovrapposizione, ma anche senza invocare il calcolo della parte (i) questo fatto segue dal ragionamento di allora applicato ai nuovi valori proiettati, perché la profondità di R_2 varia linearmente da $z = -3$ a $z = -1$, e quindi, quando scandiamo ad esempio da sinistra a destra, laddove finisce R_1 (ossia esattamente sull'asse verticale centrale $x = 4.5$ della proiezione di R_2) la profondità di R_2 e' -2: pertanto nella zona di sovrapposizione la profondità di R_2 varia da -3 a -2, mentre quella di R_1 vale 1. Quindi in tale zona R_2 è sempre coperto da R_1.
A questo punto, l'andamento lineare al crescere di x dei valori di profondità di R_2 nella parte della sua proiezione per $x > 4.5$ fornisce i valori restanti per lo $z-$buffer: i tediosi calcoli aritmetici sono lasciati al lettore. □

1.4. Ray Tracing

Il Ray Tracing è un algoritmo a precisione d'immagine che, per ogni pixel del rettangolo di visuale (viewport), determina la superficie della scena visibile attraverso quel pixel tracciando un raggio di proiezione che dal punto di visuale (centro di proiezione) attraversa il centro di quel pixel e continua fino a toccare l'oggetto della scena più vicino. Il colore del pixel è quello di questo primo oggetto intersecato. La procedura è illustrata in Figura 1.4.1.

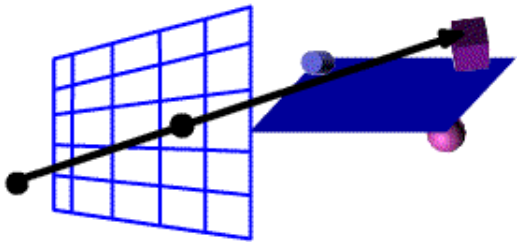


FIGURA 1.4.1. La procedura di proiezione centrale del Ray Tracing.

1.4.1. Pseudocodice del Ray Tracing (versione non ricorsiva, mirata alla rimozione di aree nascoste.

```
Selezioniamo il centro di proiezione e il piano della
      finestra;
Per   ogni scan line  {
      Per   ogni pixel di questa scan line   {
```

```
            Determiniamo il raggio (parametrico)
            dall'osservatore che passa per il centro del
            pixel;
            Per   ogni oggetto della scena   {
                      se il raggio interseca l'oggetto
                      ed il punto di intersezione con
                      questo oggetto e' piu' vicino
                      all' osservatore rispetto a tutti
                      i precedenti:
                      memorizziamo il punto di
                      intersezione e l'identificativo
                      dell'oggetto;
            }
            inizializziamo il  colore del pixel con
                      il colore dell'oggetto intersecato
                      to piÄč vicino all'osservatore;
      }
}
```

Il Ray Tracing fu sviluppato per la prima volta da Appel [**1**] e da Goldstein e Nagel [**17**, **18**] per determinare le superficie visibili dal punto di osservazione ed il loro colore in base ad una equazione di illuminazione. In seguito fu esteso ad una versione ricorsiva in grado di calcolare effetti di luce riflessa e rifratta, ed effetti di ombra, da Whitted [**52**] e Kay [**26**]: la versione ricorsiva verrà presentata in dettaglio nella successiva Sezione 4.2.

Per ogni pixel, l'algoritmo di $z-$buffer calcola le informazioni solo per quegli oggetti la cui proiezione prospettica sul piano di visuale copre almeno in parte quel pixel. Invece, la versione elementare del Ray Tracing appena descritta traccia il raggio di proiezione che passa per il centro del pixel e lo interseca con ogni oggetto della scena. Non è sorprendente quindi l'evidenza documentata da Whitted [**52**], nel cui elaboratore l'algoritmo impiegava dal 75% al 95% del tempo per i calcoli delle intersezioni, in scene tipiche. Di conseguenza, gli approcci per migliorare l'efficienza di questo algoritmo cercano di accelerare i calcoli delle singole intersezioni, o di evitarli completamente. Presenteremo tipici calcoli di intersezione nella prossima Sottosezione 1.4.2, poi metodi di ottimizzazione nella successiva Sottosezione 1.4.3, ed infine metodi per evitare calcoli inutili di intersezione nella Sottosezione 1.4.4.

1.4.2. Esempi: calcoli di intersezione del raggio proiettore con sfere e con poligoni piani. Il Ray Tracing ha il compito di determinare l'intersezione di un raggio proiettore con un oggetto. Per far ciò, rappresentiamo il raggio proiettore tramite la rappresentazione parametrica della retta. Ogni punto (x, y, z) lungo un raggio che parte dal centro di proiezione (posizione dell'osservatore) (x_0, y_0, z_0) e passa per un punto (x_1, y_1, z_1) (ad esempio, il centro di un pixel del viewport) è del tipo

$$\begin{aligned} x &= x_0 + t(x_1 - x_0), \\ y &= y_0 + t(y_1 - y_0), \\ z &= z_0 + t(z_1 - z_0). \end{aligned}$$

Per convenienza poniamo

$$\begin{aligned}\Delta x &= x_1 - x_0,\\ \Delta y &= y_1 - y_0,\\ \Delta z &= z_1 - z_0.\end{aligned}$$

(si osservi che il vettore $(\Delta x, \Delta y, \Delta z)$ è il vettore che, applicato al punto di osservazione, individua la direzione del centro del pixel). Con questa notazione, l'equazione parametrica del raggio proiettore che parte al punto di visuale (x_0, y_0, z_0) e passa per il centro (x_1, y_1, z_1) di un pixel sulla finestra di visualizzazione (viewport) diventa

$$\begin{aligned}x &= x_0 + t\Delta x,\\ y &= y_0 + t\Delta y,\\ z &= z_0 + t\Delta z.\end{aligned} \tag{1.4.1}$$

I valori di t fra 0 e 1 corrispondono al segmento fra il centro di visuale (x_0, y_0, z_0) ed il centro del pixel, (x_1, y_1, z_1). Valori negativi di t rappresentano i punti dietro il centro di proiezione, mentre valori di t maggiori di 1 corrispondono a punti sull'altro lato del centro di proiezione rispetto al piano del viewport. Se gli oggetti della scena sono dati da primitive grafiche, cioè da forme geometriche standard, per determinare il parametro di intersezione t con il raggio proiettore abbiamo bisogno della posizione degli oggetti e dell'equazione che definisce ciascuna di queste forme geometriche. Spesso gli oggetti sono definiti dalla loro superficie, modellata a maglie poligonali (wireframe). In tal caso il problema consiste nel calcolare il punto di intersezione dei raggi proiettori con i poligoni: lo tratteremo fra poco. Prima consideriamo un esempio facile di intersezione con un oggetto non poligonale. In questo contesto, l'oggetto più semplice è la sfera. La sfera con centro (a, b, c) e raggio r è rappresentata dall'equazione $(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$. L'intersezione si trova sostituendo in questa equazione i valori di x, y, z dell'equazione parametrica del raggio proiettore:

$$\begin{aligned}&\big((\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2\big)t^2\\ &\qquad + 2t\big((x_0 - a)\Delta x + (y_0 - b)\Delta y + (z_0 - c)\Delta z\big)\\ &\qquad\qquad + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0.\end{aligned}$$

Questa equazione è quadratica in t. Se essa non ammette radici reali allora il raggio e la sfera non s'incontrano; se c'è una sola radice reale (con molteplicità 2) il raggio è tangente alla sfera. Altrimenti ci sono due radici reali, che corrispondono ai punti di intersezione con la sfera: la radice (positiva) più piccola corrisponde al punto di intersezione più vicino all'osservatore (e frontale rispetto ad esso: se la sfera circonda l'intera scena, incluso l'osservatore, allora l'altra radice è negativa e corrisponde ad un punto alle spalle dell'osservatore). Per calcolare l'illuminazione della superficie nel punto di intersezione è necessario determinare anche la direzione normale alla superficie in quel punto, come vedremo in seguito nel Capitolo 2.

Ciò è particolarmente facile nel caso di una sfera, poiché la direzione normale coincide con il vettore raggio, dal centro al punto d'intersezione. La sfera con centro in (a, b, c) ha quindi al suo punto (x, y, z) la direzione normale $\big((x-a)/r, (y-b)/r, (z-c)/r\big)$.

Nel caso di un poligono, la direzione normale è ancora più facile da calcolare: se il poligono giace sul piano $Ax + By + Cz + D = 0$, allora il versore normale al piano è proporzionale al

vettore (A, B, C), quindi è dato da $(A, B, C)/\|(A, B, C)\|$, cioè

$$(A\boldsymbol{e}_1 + B\boldsymbol{e}_2 + C\boldsymbol{e}_3)/\sqrt{A^2 + B^2 + C^2}\ .$$

Invece, ciò che è più complicato è trovare il punto di intersezione di un raggio proiettore con un poligono. Questo perché, al fine di determinare dove un raggio incontra un poligono, sono necessarie due fasi di calcolo: dobbiamo prima determinare se il raggio interseca il piano del poligono, e poi se il punto d'intersezione si trova all'interno del poligono.
Partiamo di nuovo dall'equazione del piano, $Ax+By+Cz+D = 0$. Sostituendovi le equazioni parametriche del raggio(1.4.1), si ottiene

$$A(x_0 + t\Delta x) + B(y_0 + t\Delta y) + C(z_0 + t\Delta z) + D = 0,$$

cioè

$$(A\Delta x + B\Delta y + C\Delta z)t + Ax_0 + By_0 + Cz_0 + D = 0,$$

da cui

$$t = -(Ax_0 + By_0 + Cz_0 + D)/(A\Delta x + B\Delta y + C\Delta z).$$

Se $A\Delta x+B\Delta y+C\Delta z := \langle (A, B, C), (\Delta x, \Delta y, \Delta z)\rangle = 0$, allora il raggio e il piano sono paralleli e non s'intersecano (opure il raggio giace interamente nel piano). Se invece questa equazione non è verificata c'è un punto di intersezione fra raggio proiettore e piano del poligono, che si trova subito sostituendo nelle equazioni parametriche (1.4.1) il valore del parametro di intersezione t appena trovato. Resta da vedere se il punto di intersezione è interno o esterno al poligono.

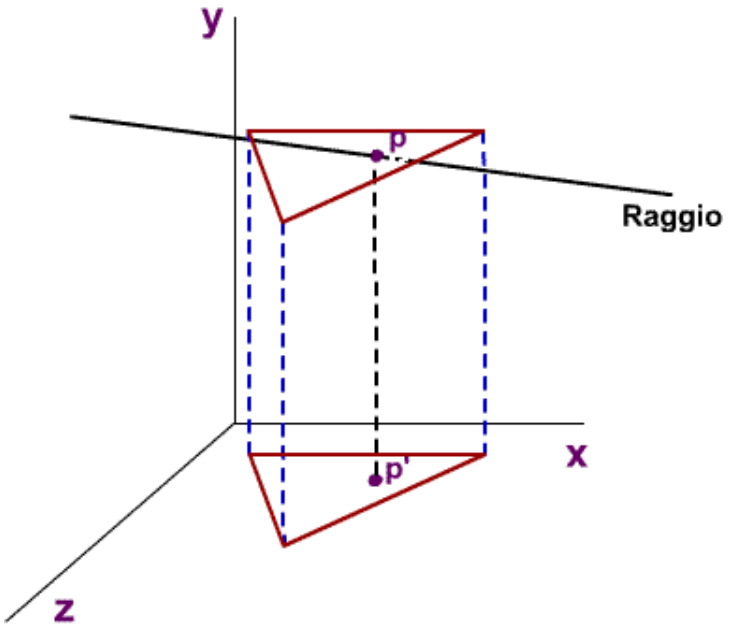


FIGURA 1.4.2. Proiezione del punto della scena intersecato dal raggio proiettore sul piano coordinato più vicino ad essere parallelo al piano del poligono.

Un modo semplice per determinare se il punto d'intersezione si trova all'interno del poligono è proiettare ortograficamente sia il poligono sia il punto di intersezione su uno dei tre piani che definiscono il sistema di coordinate, come mostrato nella Figura 1.4.2. Si noti che

un punto è interno ad un poligono se e solo se la stessa cosa succede dopo averlo proiettato, tranne che nel caso in cui la proiezione del poligono degeneri in un segmento (quindi abbia area nulla). Quindi selezioniamo l'asse lungo cui proiettare in maniera che l'area proiettata sia la più grande: ovvero quello il cui coefficiente nell'equazione del poligono ha il valore assoluto maggiore. Infatti, la coordinata il cui coefficiente ha il valore assoluto maggiore è quella il cui asse ha la direzione più vicina alla retta perpendicolare del piano del poligono: rammentiamo che, se il poligono giace sul piano $Ax + By + Cz = D$, allora il versore normale al piano è un multiplo di (A, B, C), quindi, se ad esempio A è molto più grande di B e C (in valore assoluto), il versore normale è vicino a $(1, 0, 0)$, cioè al versore dell'asse x (il cui coefficiente è quello più grande di tutti). Osserviamo che questo significa che il piano è disposto in modo tale che piccole variazioni della variabile x producono grandi variazioni delle altre due variabili, cioè, appunto, che il piano ha pendenza minore rispetto alla variabile x che alle altre due. Ad esempio, se $A = 100$, $B = 30$ e $C = 1$, l'asse di proiezione è l'asse x e quindi proiettiamo sul piano $\{y, z\}$: la proiezione sul piano $\{y, z\}$ corrisponde a buttare via la variabile x, cioè a porre $x = 0$. Il fatto di proiettare sul piano su cui la proiezione ha area maggiore serve ad escludere che, dopo la proiezione, l'area proiettata, ancorché non nulla, sia piccola, ad esempio dello stesso ordine di grandezza dell'errore di arrotondamento, il che renderebbe affetta da errore la determinazione del fatto che il punto proiettato p' sia interno od esterno al poligono proiettato, e quindi il punto di intersezione p sia interno od esterno al poligono originale. Si noti che, una volta eseguita la proiezione ortogonale sul piano coordinato più favorevole, il problema diventa bidimensionale, e quindi facile da risolvere. Supponiamo ad esempio che il poligono sia un triangolo: allora basta prendere i suoi tre lati come vettori applicati ai vertici, in ordine ciclico, diciamo antiorario quando visto da sopra, e determinare se il punto proiettato giace sempre nei semipiani sinistri rispetto a questi tre vettori. Se uno di questi tre vettori, applicato al vertice $\boldsymbol{v} = (v_1, v_2)$, lo scriviamo $\boldsymbol{n} = (n_1, n_2)$, allora iun vettore ortogonale è $\boldsymbol{n}^\perp = (n_2, -n_1)$, e considerando la scelta arbitraria del verso otteniamo in tal modo due vettori ortogonali $\pm\boldsymbol{n}^\perp = \pm(n_2, -n_1)$. La retta generata da $\boldsymbol{n}$ consiste dei punti $\boldsymbol{u} = (u_1, u_2)$ che verificano l'equazione $\langle \boldsymbol{u}, \boldsymbol{n}^\perp \rangle = \langle \boldsymbol{v}, \boldsymbol{n}^\perp \rangle$. Una volta fissato il verso di $\pm\boldsymbol{n}^\perp$, diciamo col segno $+$, il semipiano positivo rispetto a tale retta consiste dei punti $\boldsymbol{v}$ che verificano $\langle \boldsymbol{u}, \boldsymbol{n}^\perp \rangle > \langle \boldsymbol{v}, \boldsymbol{n}^\perp \rangle$, mentre il semipiano negativo è caratterizzato dalla disuguaglianza $\langle \boldsymbol{u}, \boldsymbol{n}^\perp \rangle < \langle \boldsymbol{v}, \boldsymbol{n}^\perp \rangle$). Scegliamo il verso di $\boldsymbol{n}^\perp$ in maniera che il prodotto vettore $\boldsymbol{n} \times \boldsymbol{n}^\perp$ punti verso l'alto (ossia dal lato vista dal quale la percorrenza del triangolo è antioraria). Allora il semipiano positivo della retta su cui giace un lato del triangolo è quello sinistro rispetto a questo verso di percorrenza. Perché un punto sia interno al triangolo proiettato, condizione necessaria e sufficiente è che esso verifichi le tre disuguaglianze, tutte e tre con il segno $>$, rispetto ai tre vettori $\boldsymbol{n}^\perp$ scelti con verso concorde rispetto alla percorrenza antioraria del triangolo.

Codice C++ per determinare, tramite proiezione su uno dei piani coordinati, se un punto nel piano di un triangolo nello spazio è interno o esterno al triangolo (comunicato in **[36]***:*

```
// triangolo di vertici p0, p1 e p2 presi in senso antiorario, e1 = p1-p0, e2 =
    p2-p0 e n = e1^e2
//
```

```
//        p2
//       /  \
//      /    \
//    e2/     \
//     /       \
//    /         \
//   p0__________p1
//         e1
//
// il raggio \‘{e} nella forma p = ray.o + t*ray.d


bool intersectTriangle(const Ray & ray, Intersection & hit){
  double det = -n*ray.d; // calcolo il determinante per la risoluzione del sistema
      con cramer
  if(det < D_EPS) // backface culling
    return false;
  Vec3d s = ray.o - p0;
  Vec3d m = s^ray.d;
  double u = m*e2; // prima coordinata baricentrica non normalizzata rispetto al
      determinante
  if(u < 0. || u > det) // controllo se rispetto la prima coordinata baricentrica
      l’intersezione sia dentro il triangolo
    return false;
  double v = -m*e1; // seconda coordinata baricentrica non normalizzata rispetto
      al determinante
  if(v < 0. || u+v > det) // controllo se rispetto la seconda coordinata
      baricentrica l’intersezione sia dentro il triangolo
    return false;
  double t = n*s; // distanza di intersezione non normalizzata rispetto al
      determinante
  if(t > D_EPS){ // controllo che l’intersezione sia avvenuta nel verso positivo
      della direzione del raggio
    double invDet = 1./det;
    double tf = t*invDet; // a questo punto normalizzo la distanza rispetto al
        determinante
    if(tf < hit.t){ // se il nuovo valore e’ minore del precedente procedo con i
        calcoli finali
      hit.t = tf;
      hit.u = u*invDet; // normalizzo la prima coordinata baricentrica rispetto al
          determinante
      hit.v = v*invDet; // normalizzo la seconda coordinata baricentrica rispetto
          al determinante
      return true;
    }
  }
  else
    return false;
}
```

Il Ray Tracing determina il colore dei pixel calcolando l'intersezione del raggio proiettore che passa per il loro centro con tutti gli oggetti della scena. Se si memorizza, per ciascun pixel, il valore del parametro corrispondente all'intersezione, allora, se si cambia la scena aggiungendo un nuovo oggetto, non è necessario rieseguire tutto il calcolo, ma solo calcolare, per ciascun pixel, l'intersezione del relativo raggio con questo nuovo oggetto e poi confrontare i parametri (una caratteristica analoga a quella dell'algoritmo di $z-$buffer). Naturalmente, per entrambi gli algoritmi, la determinazione del colore richiede anche il calcolo del versore normale alla superficie intersecata.

1.4.3. Ottimizzazione dei calcoli di intersezione. Molti dei termini delle equazioni per le intersezioni fra raggi di proiezione ed oggetti contengono delle espressioni che non cambiano nel corso dell'esecuzione e possono essere calcolate in anticipo e memorizzate una volta per tutte, come per esempio la proiezione ortografica di un poligono su un piano. Grazie a questo fatto si possono sviluppare metodi veloci per il calcolo dell'intersezione. Ad esempio si può utilizzare la trasformazione prospettica che sposta il punto di visuale all'infinito. In tal modo i raggi di proiezione diventano tutti paralleli all'asse z, e la proiezione centrale diventa ortografica (fissiamo il piano di visuale a $z = 0$). Ciò semplifica il calcolo delle intersezioni e consente di determinare l'oggetto più vicino mettendo in ordine crescente i valori di z assunti dagli oggetti trasformati. Il punto d'intersezione viene poi ritrasformato attraverso la trasformazione inversa per potere eseguire il calcolo dell'illuminazione.

Un altro modo di diminuire i tempi dei calcoli d'intersezione consiste nel considerare, invece degli oggetti della scena, opportuni poliedri circoscritti (bounding volumes) agli oggetti della scena. Se un oggetto ha una forma complicata che richede troppo tempo per la determinazione di intersezioni, lo si può rinchiudere in un poliedro circoscritto di forma più semplice (quindi i calcoli d'intersezione diventano meno onerosi), come una sfera, un ellissoide o un parallelepipedo. Il risparmio di tempo di calcolo è dovuto al fatto che non è necessario sottoporre quell'oggetto al calcolo della intersezione con il raggio di proiezione se questo raggio non interseca il suo poliedro delimitante.

Questo procedimento, proposto da Kay e Kajiya **[29]**, usa come poliedro delimitante un poliedro convesso formato dall'intersezione di fette (slab), ognuna definita da due piani paralleli che circondano l'oggetto. La Figura 1.4.3 mostra, visto da un lato, un oggetto circondato da quattro fette (definite, in questa proiezione piana, da coppie di linee parallele) e la loro intersezione. Così ogni fetta è rappresentata dall'equazione $Ax + By + Cz + D = 0$, dove A, B e C sono costanti, e D varia fra due valori estremi $D_{\min}$ e $D_{\max}$. Si usano fette parallele per tutti gli oggetti, e quindi ogni volume delimitante (*bounding box*) è determinato dai valori di $D_{\min}$ e $D_{\max}$ di ognuna delle sue fette. L'intersezione di un raggio con la superficie del bounding box si esegue considerando una fetta per volta. Consideriamo un raggio di proiezione $\boldsymbol{r}(t) = \boldsymbol{p_0} + t\boldsymbol{r} = (x_0 + tr_x, y_0 + tr_y, z_0 + tr_z)$. Per trovare l'intersezione di questo raggio con una fetta si sostituiscono le coordinate di $\boldsymbol{r}(t)$ nell'equazione $Ax + By + Cz + D = 0$ per ciascuno dei due piani delle fetta, cioè si pongono al posto di D i valori $D_{\min}$ e $D_{\max}$, rispettivamente: in tal modo si trovano i valori vicini e lontani del parametro di intersezione t. Quindi la soluzione t verifica $Ax_0 + By_0 + Cz_0 + D + t(Ar_x + Br_y + Cr_z) = 0$, e pertanto $t = -(Ax_0 + B_y0 + Cz_0 + D)/(Ar_x + Br_y + Cr_z)$.

Nel delimitare gli oggetti, utilizziamo un numero finito di fette. Raggruppiamole nei sottoinsiemi di fette parallele (cioè con gli stessi valori di A, B e C, ma valori diversi di D). Allora, in base a quanto appena visto, i parametri di ingresso ed uscita da una fetta sono

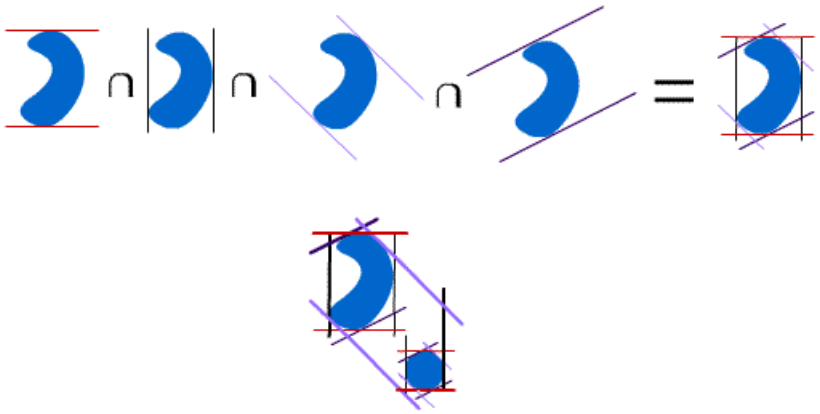


FIGURA 1.4.3. Poliedro formato da intersezioni di fette. Nella prima riga, in proiezione ortogonale sul piano $z = 0$, abbiamo un oggetto limitato da un set fisso di fette parametrizzate. Nella seconda riga vediamo un volume delimitato da due poliedri delimitanti.

$t = (S + D)T$, dove $S = Ax_0 + By_0 + Cz_0$ e $T = -1/(Ax + By + Cz)$. Questo migliora l'efficienza del calcolo delle intersezioni, perché sia S sia T possono essere calcolate una sola volta per un dato raggio ed un dato insieme di fette parallele. Poiché ogni volume delimitante è una intersezione di fette, l'intersezione del raggio con un volume delimitante è l'intersezione delle intersezioni del raggio con ciascuna fetta. Per ottenere il parametro giusto basta quindi calcolare il massimo dei valori di t corrispondenti alle intersezioni vicine con i piani delle varie fette ed il minimo dei valori di t corrispondenti alle intersezioni più lontane. Per ottimizzare il procedimento evitando di calcolare casi in cui non c'è intersezione, man mano che vengono considerate le fette vengono aggiornati i valori massimi delle intersezioni vicine ed i minimi di quelle lontane per ogni coppia di piani delimitanti: fermiamo il procedimento quando i valori vicini smettono di essere inferiori a quelli lontani.

1.4.4. Suddivisione dello spazio. Esaminiamo ora come evitare i calcoli d'intersezione non necessari. Idealmente, per ogni raggio vorremmo verificare l'intersezione solo con oggetti con cui esso s'interseca veramente, ed inoltre solo con quello che dà luogo al punto di intersezione più vicino al punto di visuale (centro di proiezione). Illustriamo solo uno dei possibili approcci in tal senso: la partizione spaziale.

Questo procedimento suddivide lo spazio in maniera progressivamente più fine. Dapprima viene calcolato l'inviluppo poliedrico (bounding box) dell'intera scena, diciamo un parallelepipedo, che per semplicità di calcolo sceglieremo con le facce parallele ai piani coordinati. Esso viene poi suddiviso in una griglia regolare, come nella Figura 1.4.4, dove viene disegnato anche un raggio proiettore che attraversa le celle della griglia. Ad ogni cella della partizione, il procedimento associa la lista degli oggetti che essa contiene interamente o in parte. Viceversa, ogni lista contiene puntatori grazie ai quali ogni oggetto in essa è associato a tutte le celle che lo contengono. Man mano che un raggio di proiezione viene tracciato dentro la scena, si deve calcolare l'intersezione del raggio solo con quegli oggetti che sono contenuti nella cella che esso sta attraversando. Inoltre le celle vengono esaminate in base all' ordine di attraversamento, così quando di determina che in una cella c'è un'intersezione, non se ne deve verificare nessun'altra successiva. Però si calcolano le intersezioni anche con tutti gli oggetti restanti nella cella, perché se il raggio ne interseca più di uno allora bisogna determinare

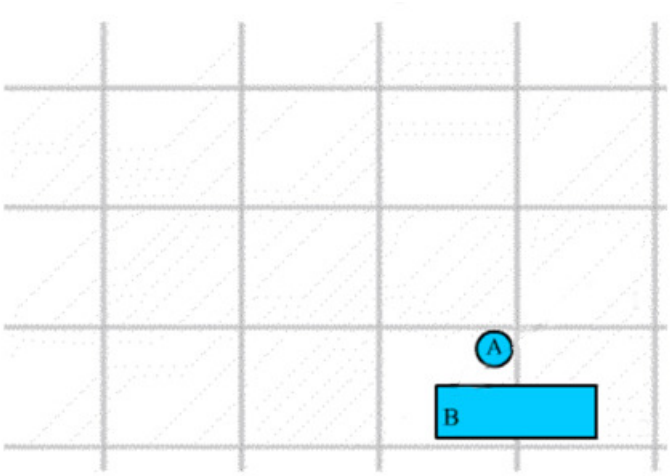


FIGURA 1.4.4. Una scena inclusa in una griglia regolare che ripartisce lo spazio

quello che dà luogo al punto di intersezione più vicino all'osservatore. Inoltre, se un raggio interseca un oggetto in una cella, si deve necessariamente controllare se il punto d'intersezione si trova nella stessa cella; è possibile infatti che l' intersezione ci sia, ma sia localizzata in una cella successiva, e che invece un altro oggetto nella cella corrente possa avere un punto di intersezione più vicino. Questo fenomeno è illustrato nella Figura 1.4.5. Il raggio viene

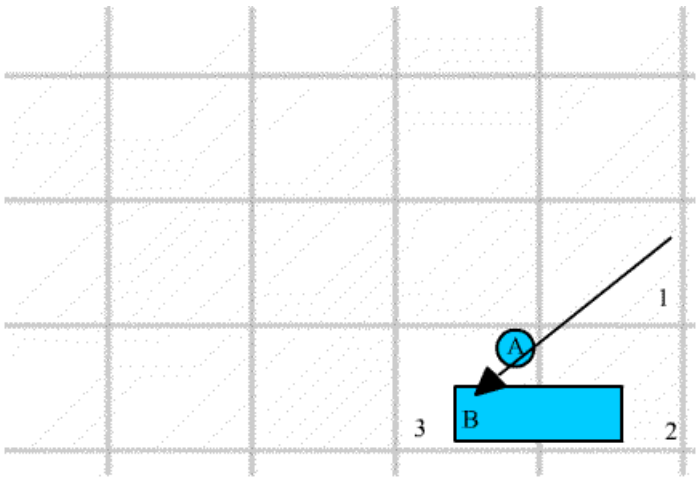


FIGURA 1.4.5. L'oggetto b è intersecato dal raggio proiettore nella cella 3 sebbene l'esistenza della intersezione sia stata determinata quando fu presa in esame la cella 2

seguito mentre attraversa la griglia di celle finché si determina un'intersezione con un oggetto contenuto in una di esse (ad esempio l'oggetto a nella cella 3). Per evitare di ricalcolare l'intersezione di un raggio con un oggetto che si estende a varie celle, il punto di intersezione ed il pixel che corrisponde al raggio vengono memorizzati insieme all'oggetto intersecato la prima volta che il raggio lo visita.
Una variante a suddivisione variabile produce partizioni diseguali ottenute suddividendo lo

spazio mediante un octree [**13**]. (un octree è una partizione binaria dello spazio a celle variabili in maniera adattabile, introdotto in [**16**], che qui verrà presentato nella Sottosezione **??**). Per gli octree esiste un algoritmo efficiente per determinare le celle vicine ad una data cella, presentato nella successiva Sottosezione 1.5.2, il quale può essere impiegato qui per determinare la cella successiva che si trova lungo il percorso raggio: questo migliora ulteriormente l'ottimizzazione.

1.5. Una struttura di dati adatta alla suddivisine dello spazio con bisezioni iterative: gli octrees

1.5.1. Quadtrees ed octrees. I quadtrees e gli octrees sono strutture ad albero i cui vertici (detti anche nodi) sono celle ottenute da partizioni binarie successive (cioè suddivisioni progressive di passo due), rispettivamente di un cubo o di un quadrato, entro il quale si trova una figura, rispettivamente solida o piana, che si vuole approssimare come unione di celle della partizione. Da questi alberi ricaviamo una parametrizzazione delle celle della partizione in cui si trova l'oggetto. Si ottiene un quadtree suddividendo iterativamente a metà in entrambe le dimensioni un quadrato che contiene la figura piana che vogliamo rappresentare: ciascuna suddivisione dà luogo a quattro celle. Ogni cella può trovarsi in varie condizioni di inclusione rispetto alla figura: può essere piena (completamente interna all'oggetto), parzialmente piena oppure vuota (completamente esterna all'oggetto). Le successive suddivisioni organizzano le varie celle come vertici di un albero genealogico. I vertici dell'albero corrispondono alle celle; i suoi spigoli uniscono ciascuna cella con le quattro celle ottenute dalla sua suddivisione, o viceversa. Possiamo fissare una soglia, cioè una percentuale prefissata, e dichiarare piena una cella che è interna all'oggetto per una parte della propria area superiore a tale percentuale, e vuota una cella esterna all'oggetto per un'area inferiore a tale percentuale. Nelle figure, per enfatizzare, abbiamo scelto una percentuale molto bassa, dell'ordine del 15%.
Una cella parzialmente piena viene ulteriormente suddivisa in maniera ricorsiva. Questa suddivisione continua fino a che tutte le celle sono omogenee (ognuna piena o vuota entro la soglia prefissata) o fino a che si raggiunge un prefissato numero di iterazioni.
Gli octree costituiscono la analoga rappresentazione ad albero per figure tridimensionali: ogni suddivisione avviene per ciascuna delle tre dimensioni e dà luogo ad otto celle.
Nella parte di sinistra della Figura 1.5.1 è rappresentata la griglia delle celle relativa alla partizione più fine. Nella parte di destra è visualizzata la stessa immagine con la suddivisione delle celle del quadtree: le celle ora non appartengono più tutte alla partizione più fine, perché per alcune zone, a seconda della figura che si vuole rappresentare, il processo di suddivisione si ferma prima. Le suddivisioni successive vengono rappresentate in termini di un albero i cui vertici rappresentano le varie celle di ogni generazione. Quelli parzialmente pieni sono qui indicati con una P (*partially*), quelli pieni con una F (*full*) ed infine quelli vuoti con una E (*empty*). Inoltre alle quattro celle create in ogni ripartizione è associato un numero da zero fino a tre, nell'ordine presentato nella tabella della Figura 1.5.2.

Nel caso degli octrees, ogni ripartizione crea otto celle tridimensionali, che vengono numerate da 0 a 7 nell'ordine visualizzato nella Figura 1.5.4, che rappresenta due partizioni successive (nella seconda viene ripartita solo la cella frontale in alto a destra).

1.5.2. Algoritmo di ricerca della cella adiacente in un octree. Un'altra operazione importante nei quadtrees e negli octrees consiste, per ogni cella, ossia per ogni vertice dell'albero, nel trovare, fra gli altri vertici, quale sia quello che corrisponde ad una cella

FIGURA 1.5.1. Costruzione del quadtree di un oggetto piano mediante suddivisioni binarie successive del suo bounding box

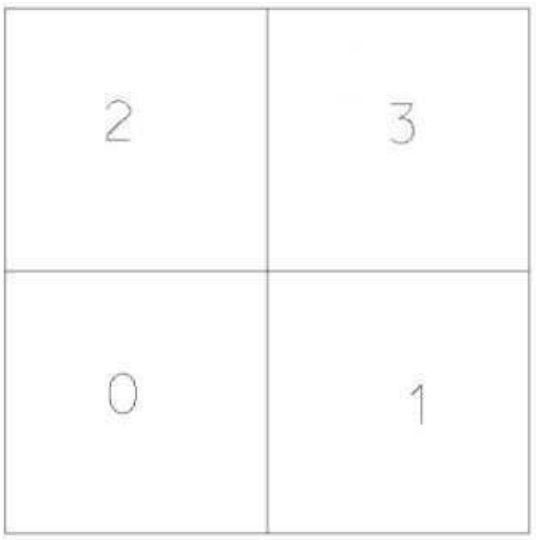


FIGURA 1.5.2. Numerazione dei discendenti dei nodi dell'albero in Figura 1.5.3 a seconda della loro posizione

adiacente a quella associata al vertice considerato, cioè localizzare un vertice adiacente (nel senso della decomposizione in celle del piano o dello spazio) al vertice originale (con la cui cella, cioè, condivide una faccia, un lato o un vertice) e di generazione non superiore (se la generazione è superiore, le celle adiacenti sono più d'una!).

Un nodo di un quadtree (ossia una cella nella geometria bidimensionale) ha le celle adiacenti in otto possibili direzioni. Le sue celle vicine a nord, sud, est e ovest sono contigue lungo un lato comune, mentre quelle vicine a nord-ovest, nord-est, sud-ovest e sud-est sono contigue lungo un vertice comune. Invece un nodo di un octree (ossia una cella tridimensionale) ha vicini in 26 possibili direzioni: 6 vicini lungo una faccia, 12 vicini lungo un lato e 8 vicini lungo un vertice. Un algoritmo per trovare il vicino del vertice lungo una direzione data è stato introdotto in [**41**]. Si considerano due celle adiacenti, ed il vertice associato ad una di esse (quella di generazione più elevata, quindi la più piccola in grandezza). Si vuol trovare il vertice associato all'altra cella. Si parte dal vertice originale e si risale il quadtree (o octree)

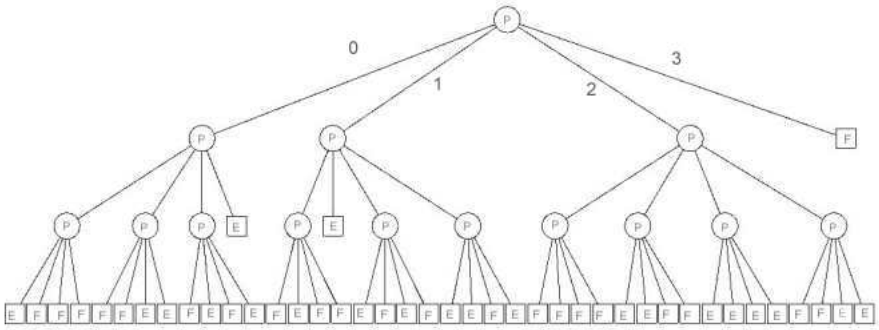


FIGURA 1.5.3. L'albero associato alla ripartizione tramite quadtree della Figura 1.5.1. I numeri sui segmenti indicano quale delle quattro celle della suddivisione stiamo raggiungendo nell'avanzare di una generazione: la numerazione è quella della tabella nella Figura 1.5.2

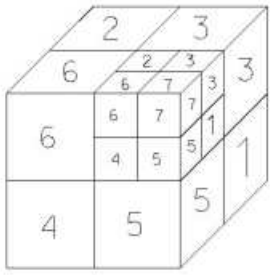


FIGURA 1.5.4. Numerazione delle sottocelle di un octree: per ogni generazione, la numerazione si riferisce solo all'ubicazione entro la cella genitore

fino al primo antenato comune fra le due celle. Poi si discende l'albero fino a trovare il vertice corrispondente alla cella adiacente. Per questo bisogna risolvere due problemi: trovare l'antenato comune e determinare quale dei suoi discendenti è il vicino cercato.
Per prima cosa spieghiamo come si determina l'antenato comune. Il caso più semplice è quando si vuol trovare il vicino del vertice di un octree nella direzione di una delle sue facce: L (*left*), R (*right*), U (*up*), D (*down*), F (*front*) oppure B (*back*). Quando risaliamo l'albero a partire dal vertice originale, l'antenato comune è il primo vertice che non è raggiunto da un figlio sul lato del vertice. Per esempio se cerchiamo un vicino sul lato sinistro L, allora il

primo antenato comune è il primo vertice che non è raggiunto da un figlio in posizione LUF, LUB, LDF oppure LDB. Questo perché un vertice che è stato raggiunto da uno di questi figli non può avere alcun figlio che è a sinistra del vertice originale.
È opportuno spiegare qual è il modo a tal fine più opportuno di codificare le direzioni con cui ci si muove da un vertice figlio al padre o viceversa: cioè codificare gli spigoli. Nel caso di un quadtree, ci sono quattro figli, denotati dai numeri 0, 1, 2, 3. Scriviamo questi quattro numeri nel sistema binario. Allora i due figli in basso sono codificati dai numeri 00 e 01, quelli in alto da 10 e 11. In altre parole, il bit più alto vale 0 sulla riga bassa, 1 sulla riga alta; il bit basso vale 0 sulla colonna di sinistra, 1 su quella di destra. Pertanto, raggiungere il progenitore partendo dal suo lato basso significa percorrere un segmento codificato da un numero binario a due cifre la cui prima cifra è 0; raggiungerlo dal lato di sinistra significa percorrere un segmento con un codice binario la cui ultima cifra è 0, e così via. Analogamente, per un octree, i segmenti hanno codice binario di tre cifre: quelli che portano a figli sul lato posteriore hanno bit più significativo uguale a zero (si tratta dei numeri 0 = 000, 1 = 001, 2 = 010, 3 = 011), quelli del lato sinistro (0 = 000, 2 = 010, 4 = 100, 6 = 110) hanno bit meno significativo 0, quelli sul lato basso (0 = 000, 1 = 001, 4 = 100, 5 = 101) hanno bit centrale uguale a zero, e così via.

Una volta trovato l'antenato comune, per determinare il vertice corrispondente alla cella adiacente nella direzione assegnata è sufficiente discendere l'albero in maniera speculare a come lo si era risalito, nel senso della riflessione speculare riflesso al bordo comune (nel caso dell'esempio più sopra, questo significa nello scegliere la direzione R ogni volta che ci si era mossi in direzione L, e viceversa).
Cosa vuol dire riflesso speculare in termini della precedente codifica binaria? Vuol dire, ad esempio in un octree, che per riflettere specularmente la destra nella sinistra dobbiamo cambiare il valore del bit meno significativo, da 1 a 0 (passando da 1,3,5,7 a 0,2,4,6 rispettivamente, cioè da 001, 011, 011, 111 a 000, 010, 100, 110, rispettivamente).

Se la cella vicina è più grande della cella originale, cioè se il vertice corrispondente appartiene ad una generazione più antica di quello originale, allora il percorso riflesso discendente termina prima (cioè arriva ad un vertice terminale in meno passi di quelli fatti in salita: ha una lunghezza inferiore a quello ascendente).
Un metodo simile può essere usato per trovare un vicino del vertice di un quadtree nella direzione di uno dei suoi lati.

Per una implementazione inel linguaggio C++ dell'algoritmo di ricerca della cella adiacente in un octree, si veda l'Appendice 1.7.

1.6. Antialiasing nel Ray Tracing

Il Ray Tracing campiona i punti di una griglia e quindi produce immagini con aliasing, commisurato alle dimensioni delle celle della griglia. In **[52]** fu sviluppato un metodo adattivo che emettere più raggi verso quelle parti dell' immagine che altrimenti produrrebbero l'aliasing. Questo sovracampionamento variabile serve ad ottenere una maggiore precisione per ogni pixel. Si associano i raggi ai vertici di ogni pixel piuttosto che al suo centro. Dopo che i raggi sono stati inviati attraverso i quattro angoli del pixel, si fa la media delle luminosità effettive corrispondenti. Se i quattro valori sono vicini (entro un soglia prefissata), allora illuminiamo il pixel con questo livello medio di luminosità. Se no, il pixel viene ulteriormente suddiviso tramite i punti mediani dei suoi lati ed il suo centro, in modo quindi da formare

quattro sottopixels verso i quali si sparano nuovi raggi che attraversano i loro vertici. La suddivisione procede in modo ricorsivo fino a che viene raggiunta un livello minimo prefissato di dimensioni lineari delle celle che costituiscono i sottopixels, oppure finché le luminosità siano sufficientemente vicine, entro la soglia prefissata. A questo punto la luminosità del pixel è data dalla media, pesata in proporzione all'area, delle luminosità dei suoi sottopixels. Chiaramente, il sovracampionamento adattivo produce una approssimazione migliore del campionamento di area non pesato. La Figura 1.6.1 mostra i raggi tracciati attraverso i

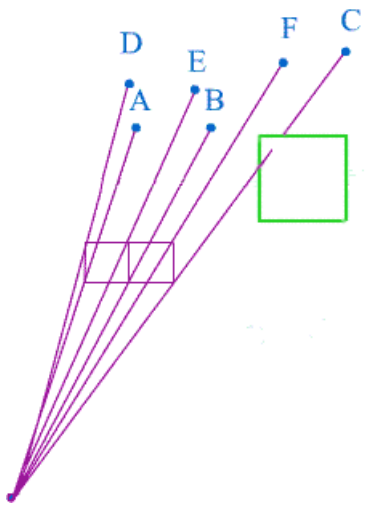


FIGURA 1.6.1. Raggi proiettori attraverso i vertici di due pixel adiacenti; la linea verde è un tratto del profilo di un oggetto della scena visibile attraverso quei pixel

vertici di due pixels adiacenti, ed un oggetto della scena schematizzato in verde. La scena si proietta sul viewport, ed appare nei pixel della Figura 1.6.1 come illustrato in Figura 1.6.2. I

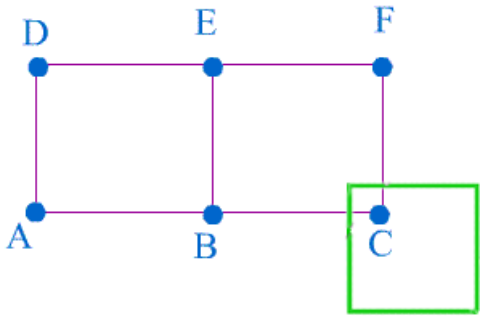


FIGURA 1.6.2. La proiezione dell'oggetto verde della scena sui pixel della Figura 1.6.1

raggi proiettori attraverso i vertici del pixel di sinistra (punti $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{D}$, $\boldsymbol{E}$) non intersecano oggetti della scena e quindi contribuiscono lo stesso valore di illuminazione, quello dello sfondo, che in generale assumiamo che abbia variazioni molto lievi. La stessa cosa succede per il raggio attraverso $\boldsymbol{F}$. Indichiamo con A il valore di illuminazione dovuto al raggio attraverso

A, con B quello attraverso ***B***, e così via. Quindi non è necessaria una suddivisione ulteriore per il pixel determinato dai raggi A, B, D, E, e la luminosità di quel pixel vale la media dei contributi, $(A + B + D + E)/4$.
Nel pixel adiacente, il raggio attraverso ***C*** colpisce l'oggetto verde, e può produrre un contributo assai diverso dagli altri. Quindi il pixel adiacente richiede un'ulteriore suddivisione, e pertanto vengono tracciati nuovi raggi attraverso ***G***, ***H***, ***K***, ***J*** e ***I*** (ovvero i vertici dei quattro sottopixel di questo pixel), come illustrato in Figura 1.6.3. Il sottopixel in basso a

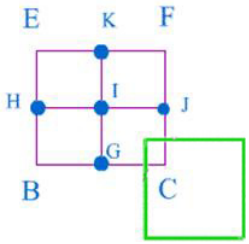


FIGURA 1.6.3. Prima generazione di infittimento dei proiettori per il pixel a destra nella Figura 1.6.1

destra ha tre vertici, G, I e J, attraverso cui i proiettori si perdono sul fondo, ed un quarto, C, il cui proiettore colpisce l'oggetto verde. Perciò questo sottopixel, e solo esso, deve essere diviso ulteriormente suddiviso, ed a questo scopo si tracciano i raggi attraverso ***L***, ***M***, ***P***, ***O*** e ***N*** (Figura 1.6.4). Supponiamo che a questo punto si sia raggiunto il livello massimo

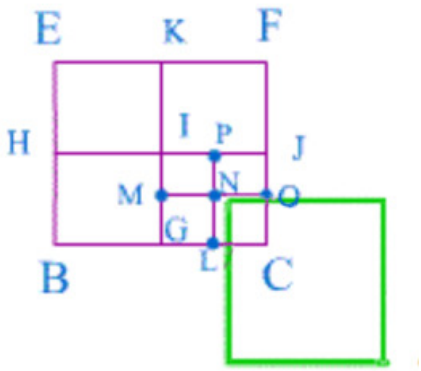


FIGURA 1.6.4. Deconda generazione di infittimento dei proiettori per il sottopixel in basso a destra nella Figura 1.6.3

predefinito di suddivisione. La luminosità risultante sul pixel allora è:

$$\frac{1}{4}\Bigg(\frac{B+G+H+I}{4}+\frac{H+E+K+I}{4}+\frac{I+K+F+J}{4} +\frac{1}{4}\Big(\frac{G+M+N+L}{4}+\frac{M+I+P+N}{4}+\frac{N+P+J+O}{4} +\frac{L+N+O+C}{4}\Big)\Bigg).$$

Possono sorgere problemi di aliasing su un pixel quando i raggi che attraversano il pixel mancano un oggetto piccolo. Ciò produce effetti evidenti se gli oggetti sono collocati in disposizioni regolari, ad esempio a scacchiera, ed alcuni rimangono visibili mentre altri scompaiono, oppure se in una animazione (una serie di immagini consecutive di un oggetto in movimento) l'oggetto appare e scompare periodicamente da un fotogramma al successivo (a seconda che venga colpito o mancato dal raggio proiettore). Il procedimento di Whitted in **[52]** evita questi fenomeni circondando ogni oggetto con un inviluppo (bounding volume) sferico sufficientemente grande da essere intersecato da almeno un raggio proveniente dall'osservatore. Poiché i raggi si diramano dal centro di proiezione, la grandezza dell'inviluppo deve crescere con la distanza dal centro di osservazione. Se un raggio interseca l'inviluppo ma non interseca l'oggetto, allora tutti i pixels che appartengono a quel raggio sono ulteriormente suddivisi finché l'oggetto non viene intersecato.

1.7. Appendice: codice in linguaggio C++ per la ricerca della cella adiacente in un octree

Il codice seguente, sviluppato da Marco Petreri [**35**], utilizza la numerazione binaria delle partizioni dell'octree per implementare l'algoritmo di ricerca delle celle vicine illustrato nella sottosezione precedente.

```
#include <iostream>

#include "geometry/vec3d.hpp"
#include "geometry/voxel.hpp"
#include "geometry/octree.hpp"

using namespace std;

int main(int argc, char const *argv[]) {

  Octree oct(2, Vec3d(0), Vec3d(4)); // creo l'octree di generazione 2 con la sua
      root di vertice minimo {0,0,0} e massimo {4,4,4}
  oct.initialize(); // costruisco l'albero

  cout << oct.toString() << "\n";

  Vec3d p(1,2,1), p1(1,2,2);

  oct.adjoined(p, p1); // controllo se i due punti sono in celle adiacenti

  return 0;
}
```

```
#ifndef _OCTREE
#define _OCTREE

#include "voxel.hpp"

class Octree{
public:
  Octree(): gen(1), root(new Voxel()){}
  Octree(int _gen, const Vec3d & min, const Vec3d & max): gen(_gen), root(new
      Voxel(min, max, 0, 0)){}
  ~Octree(){
    delete root;
  }

  void initialize(){ // avvia la costruzione della root dell'octree
    build(root, 1);
  }
```

```
void build(Voxel * node, int tgen) const{ // costruisce ricorsivamente l'albero
    di voxel fino al raggiungimento della generazione desiderata
  node->voxelize(tgen);
  ++tgen;
  if(tgen > gen)
    return;

  for(auto & i : node->childs)
    build(i, tgen);
}

Voxel * contains(const Vec3d & p, Voxel * node) const{ // controlla
    ricorsivamente in quale voxel Ăš presente il punto
  if(node->hasIn(p)){
    // std::cout << "Ăš in: " << node->idx << ", min: " << node->min << ", max: "
        << node->max << "\n";
    if(node->childs.size() > 0){
      for(auto & i : node->childs){
        Voxel * r = contains(p, i);
        if(r)
          return r;
      }
    }
    else
      return node;
  }
  else
    return nullptr;

}

void adjoined(const Vec3d & p1, const Vec3d & p2) const{ // controlla se due
    punti sono contenuti in celle adiacenti
  Voxel * v1 = contains(p1, root), * v2 = contains(p2, root);
  std::cout << "Punti " << p1 << " e " << p2 << " contenuti nei voxel " <<
      v1->idx << " e " << v2->idx << " rispettivamente\n";

  std::cout << "Adiacenze di p1 per facce: \n";
  std::cout << "right:" << moveR(v1->idx, v1->gen) << ", left:" << moveL(v1->idx,
      v1->gen) << ", up:" << moveU(v1->idx, v1->gen) << ", down:" <<
      moveD(v1->idx, v1->gen) << ", front:" << moveF(v1->idx, v1->gen) << ",
      back:" << moveB(v1->idx, v1->gen) << std::endl;
  for(int i = 0; i < 3; ++i){
    if(v2->idx == move(v1->idx, v1->gen, i, 1) || v2->idx == move(v1->idx,
        v1->gen, i, 0)){
      std::cout << "I voxel sono adiacenti per facce.\n";
      break;
    }
  }
```

```
  std::cout << "Adiacenze di P1 per spigoli\n";
  std::cout << "movimento R+F: " << moveRF(v1->idx, v1->gen) << "\n";
  std::cout << "movimento L+F: " << moveLF(v1->idx, v1->gen) << "\n";
  std::cout << "movimento R+B: " << moveRB(v1->idx, v1->gen) << "\n";
  std::cout << "movimento L+B: " << moveLB(v1->idx, v1->gen) << "\n";
  std::cout << "movimento R+U: " << moveRU(v1->idx, v1->gen) << "\n";
  std::cout << "movimento L+U: " << moveLU(v1->idx, v1->gen) << "\n";
  std::cout << "movimento R+D: " << moveRD(v1->idx, v1->gen) << "\n";
  std::cout << "movimento L+D: " << moveLD(v1->idx, v1->gen) << "\n";
  std::cout << "movimento F+D: " << moveFD(v1->idx, v1->gen) << "\n";
  std::cout << "movimento B+D: " << moveBD(v1->idx, v1->gen) << "\n";
  std::cout << "movimento F+U: " << moveFU(v1->idx, v1->gen) << "\n";
  std::cout << "movimento B+U: " << moveBU(v1->idx, v1->gen) << "\n";

}

int move(int n, int gen, int axis, int dir) const{ // controllo ricorsivamente
    se esiste il voxel in direzione dir sull'asse axis, in caso affermativo
    ritorno il suo id intero, se non esiste ritorno -1
  if(n < 0)
    return -1;
  int g = gen;
  int p = n;
  int a = pow(2,axis);
  if(get(n,a) == a*dir){
    while(g > 1){
      --g;
      p = parent(p);
      if(get(p,a) == a*(1-dir)){
        return toggle(toggle(n,(gen-g)*8*a), a);
      }
    }
    return -1;
  }
  else
    return toggle(n,a);
}

int moveR(int n, int gen) const{
  return move(n, gen, 0, 1);
}

int moveL(int n, int gen) const{
  return move(n, gen, 0, 0);
}

int moveU(int n, int gen) const{
  return move(n, gen, 1, 1);
```

```
}

int moveD(int n, int gen) const{
  return move(n, gen, 1, 0);
}

int moveF(int n, int gen) const{
  return move(n, gen, 2, 1);
}

int moveB(int n, int gen) const{
  return move(n, gen, 2, 0);
}

int moveRF(int n, int gen) const{
  return moveF(moveR(n, gen), gen);
}

int moveLF(int n, int gen) const{
  return moveF(moveL(n, gen), gen);
}

int moveRB(int n, int gen) const{
  return moveB(moveR(n, gen), gen);
}

int moveLB(int n, int gen) const{
  return moveB(moveL(n, gen), gen);
}

int moveRU(int n, int gen) const{
  return moveU(moveR(n, gen), gen);
}

int moveLU(int n, int gen) const{
  return moveU(moveL(n, gen), gen);
}

int moveRD(int n, int gen) const{
  return moveD(moveR(n, gen), gen);
}

int moveLD(int n, int gen) const{
  return moveD(moveL(n, gen), gen);
}

int moveFD(int n, int gen) const{
  return moveD(moveF(n, gen), gen);
}
```

```
int moveBD(int n, int gen) const{
  return moveD(moveB(n, gen), gen);
}

int moveFU(int n, int gen) const{
  return moveU(moveF(n, gen), gen);
}

int moveBU(int n, int gen) const{
  return moveU(moveB(n, gen), gen);
}

int firstCommonParent(int a, int b) const{ // trova il primo parent in comune
    tra due celle
  int i = 0;
  if(a == b)
    return 0;
  while(a != 0 || b != 0){
    a >>= 3; b >>= 3;
    ++i;
    if(a == b)
      return i;
  }
  return i;
}

int set(int n, int i) const{
  return n & (~i);
}

int get(int n, int i) const{
  return n & i;
}

int toggle(int n, int i) const{
  return n ^ i;
}

int parent(int n) const{
  return n >> 3;
}

std::string toString() const{
  std::string s = "Octree: [ gen: " + std::to_string(gen) + "\n";
  recString(s, root);
  return s;

}
```

```
  void recString(std::string & s, Voxel * node) const{
    s += "node: " + node->toString() + "\n";
    if(node->childs.size() > 0){
      s += "childs:\n";
      for(auto & i : node->childs){
        recString(s, i);
      }
      s += "\n";
    }
  }

  int gen;
  Voxel * root;
};


#endif
```

```
#ifndef _VEC_3D_
#define _VEC_3D_

#include <iostream>
#include <math.h>
#include <assimp/vector3.h>

class Vec3d{
public:
  Vec3d(): x(), y(), z(){}
  Vec3d(double s): x(s), y(s), z(s){}
  Vec3d(double _x, double _y, double _z): x(_x), y(_y), z(_z){}
  Vec3d(const Vec3d & v): x(v.x), y(v.y), z(v.z){}
  Vec3d(const aiVector3D & v): x(v.x), y(v.y), z(v.z){}

  Vec3d operator+(const Vec3d & v) const{return Vec3d(x+v.x, y+v.y, z+v.z);}
  Vec3d & operator+=(const Vec3d & v){x += v.x; y += v.y; z += v.z; return *this;}
  Vec3d operator-(const Vec3d & v) const{return Vec3d(x-v.x, y-v.y, z-v.z);}
  Vec3d & operator-=(const Vec3d & v){x -= v.x; y -= v.y; z -= v.z; return *this;}
  Vec3d operator*(double s) const{return Vec3d(x*s, y*s, z*s);}
  Vec3d & operator*=(double s){x *= s; y *= s; z *= s; return *this;}
  double operator*(const Vec3d & v) const{return x*v.x + y*v.y + z*v.z;}
  Vec3d operator^(const Vec3d & v) const{return Vec3d(y*v.z - z*v.y, z*v.x -
     x*v.z, x*v.y - y*v.x);}
  Vec3d operator%(const Vec3d & v) const{return Vec3d(x*v.x, y*v.y, z*v.z);}
  Vec3d operator/(double s) const{double d = 1./s; return Vec3d(x*d, y*d, z*d);}
  Vec3d & operator/=(double s){double d = 1./s; x *= d; y *= d; z *= d; return
     *this;}
  Vec3d operator-() const{return Vec3d(-x, -y, -z);}
```

```
  bool operator==(const Vec3d & v) const{return x == v.x && y == v.y && z == v.z;}
  bool operator!=(const Vec3d & v) const{return x != v.x || y != v.y || z != v.z;}
  bool operator>(const Vec3d & v) const{return x > v.x && y > v.y && z > v.z;}
  bool operator>=(const Vec3d & v) const{return x >= v.x && y >= v.y && z >= v.z;}
  bool operator<(const Vec3d & v) const{return x < v.x && y < v.y && z < v.z;}
  bool operator<=(const Vec3d & v) const{return x <= v.x && y <= v.y && z <= v.z;}
  Vec3d & operator=(const Vec3d & v){x = v.x; y = v.y; z = v.z; return *this;}
  Vec3d & operator=(const aiVector3D & v){x = v.x; y = v.y; z = v.z; return *this;}
  double operator[](int i) const{return (&x)[i];}
  double & operator[](int i) {return (&x)[i];}

  double lengthSq() const{return x*x + y*y + z*z;}
  double length() const{return sqrt(lengthSq());}
  Vec3d hat() const{return (*this)/length();}
  Vec3d & normalize(){return (*this)/=length();}
  Vec3d reflect(const Vec3d & n) const{return *this - n*((*this)*n)*2.;}
  Vec3d & reflect(const Vec3d & n){return *this -= n*((*this)*n)*2.;}
  Vec3d floorVec() const{return Vec3d(floor(x),floor(y),floor(z));}
  Vec3d & floorVec(){x = floor(x); y = floor(y); z = floor(z); return *this;}

  std::string toString() const{
    return "{ "+std::to_string(x)+", "+std::to_string(y)+", "+std::to_string(z)+"}";
  }

  double x, y, z;
};

inline Vec3d operator*(double s, const Vec3d & v){return v*s;}
inline std::ostream & operator<<(std::ostream & os, const Vec3d & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << "}";
}

#endif
```

```
#ifndef _VOXEL
#define _VOXEL

#include "vec3d.hpp"
#include <vector>

class Voxel{
public:
  Voxel(): idx(0), gen(0), min(), max(){}
  Voxel(const Vec3d & _min, const Vec3d & _max, int i, int g): idx(i), gen(g),
      min(_min), max(_max){}
  Voxel(const Voxel & v): idx(v.idx), gen(v.gen), min(v.min), max(v.max){}
  ~Voxel(){
    for(auto & i : childs)
```

```
      delete i;
  }

  void voxelize(int tgen){ // suddivide il voxel corrente in 8 voxel figli,
      assegnandoli a childs
    Vec3d sd = (max - min)*.5;
    Vec3d tmin = min;
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 0, tgen));
    tmin = min + sd%Vec3d(1,0,0);
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 1, tgen));
    tmin = min + sd%Vec3d(0,1,0);
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 2, tgen));
    tmin = min + sd%Vec3d(1,1,0);
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 3, tgen));
    tmin = min + sd%Vec3d(0,0,1);
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 4, tgen));
    tmin = min + sd%Vec3d(1,0,1);
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 5, tgen));
    tmin = min + sd%Vec3d(0,1,1);
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 6, tgen));
    tmin = min + sd%Vec3d(1,1,1);
    childs.push_back(new Voxel(tmin, tmin + sd, idx*8 + 7, tgen));
  }

  bool hasIn(const Vec3d & p) const{ // controlla se il punto ĂŠ presente nel
      voxel o sui suoi bordi
    return (p >= min && p <= max) ? true : false;
  }


  std::string toString() const{
    return "[ idx: " + std::to_string(idx) + ", gen: " + std::to_string(gen) + ",
        min: " + min.toString() + ", max: " + max.toString() +", nChilds: " +
        std::to_string(childs.size()) + "]";
  }

  int idx, gen;
  Vec3d min, max;
  std::vector<Voxel *> childs;

};

#endif
```

1.8. Appendice: una implementazione dello $z-$buffer in Java

La seguente implementazione del procedimento di $z-$buffer nel linguaggio Java è stata comunicata da Giacomo Nazzaro in [**32**]. L'immagine di una scena composta da tre solidi (un ottaedro, una sfera ed un cubo) prodotta da questo renderer di $z-$buffer è in Figura 1.8.1.

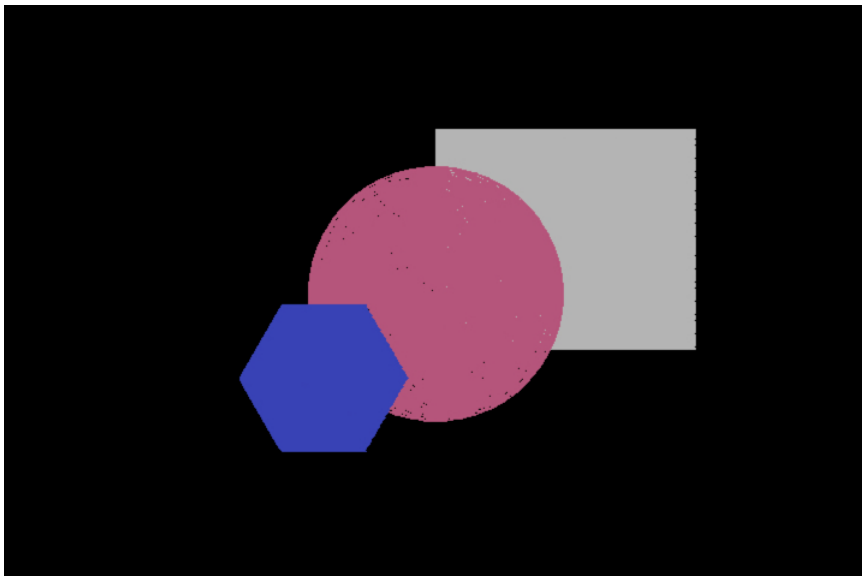


FIGURA 1.8.1. L'immagine di una scena consistente di un ottaedro, una sfera ed un cubo, prodotta dal renderer di $z-$buffer della Sezione 1.9.

```
import java.awt.*;
import java.util.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;

public class Main extends JFrame{
   public MyJPanel canvas;
   public int f;
   public static Scene scn;
   public static Main sv;

   public Main(String s){
   // Main si occupa solo di far eseguire il programma e gestire l'interfaccia
       grafica.
   // Non e' interessante dal punto di vista teorico. Non e' pienamente commentato.

      super(s);
      f = 700;
      canvas = new MyJPanel(f);
      this.add(canvas);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
      this.pack();
      scn = new Scene(0.1);

      scn.makeSphere(300, new Point(0,0, 1300), 50, 0.8, 0.3, 0.5);
        scn.makeQuad(new Point(0,-450,1600), new Point(0,600,0),new Point(711,0,0),
            .8,.8,.8);
        scn.makeSphere(150, new Point(-200, 150, 800), 6, 0.2, 0.2, 0.8);

      this.setVisible(true);
      this.addKeyListener(new KeyListener() {
           public Point v;
        /* le prossime istruzioni permettono di spostare il punto di visuale
            tramite i tasti di freccia e backspace/delete
        */
        @Override public void keyPressed(KeyEvent e) {
             if(e.getKeyCode() == KeyEvent.VK_UP) v = new Point(0,-10,0);
             if(e.getKeyCode() == KeyEvent.VK_DOWN) v = new Point(0,10,0);
             if(e.getKeyCode() == KeyEvent.VK_LEFT) v = new Point(-10,0,0);
             if(e.getKeyCode() == KeyEvent.VK_RIGHT) v = new Point(10,0,0);

             if(e.getKeyCode() == KeyEvent.VK_BACK_SPACE) v = new Point(0,0,-10);
             if(e.getKeyCode() == KeyEvent.VK_SPACE) v = new Point(0,0,10);

             for(int i=0; i<scn.pts.size(); i++){
              scn.pts.get(i).shift(v.per(-1));
             }
             repaint();
         }
         @Override public void keyTyped(KeyEvent e) {}
         @Override public void keyReleased(KeyEvent e){}
      });

   }

   public static void main(String args[]){

      RenderingEngine R = new RenderingEngine(720, 480, 700);
      Main C = new Main("Rendered");

   }
}
```

```
class MyJPanel extends JPanel{
   public ArrayList<Point> GList = new ArrayList<Point>();
   public ArrayList<Point> PList = new ArrayList<Point>();
   public Point p1 = new Point(300,300,20);
   public float r, g, b;
   public static int f,j,i,h,w;
```

```
   public MyJPanel(int f){
      super();
      this.f = f;
      w = 720;
      h = 480;
      this.setPreferredSize(new Dimension(w, h));
      this.setVisible(true);
      //PList.add(p1);
   }

    @Override public void paintComponent(Graphics gr){
      super.paintComponent(gr);
      gr.setColor(new Color(0,0,0));
      gr.fillRect(0, 0, this.getWidth(), this.getHeight());
      RenderingEngine.render(Main.scn);
      for(i=0; i<w; i++)
         for(j=0; j<h; j++){
            gr.setColor(RenderingEngine.pixelMatrix[i][j]);
            gr.drawLine(i,j,i,j);
         }
   }
}

public class Point{

  // cooridnate del punto nello spazio
  public double x;
  public double y;
  public double z;

  // colore
  public Point clr;

  // versore normale al punto
  public Point normal;

  public Point(){
    //costruttore
      this.set(0, 0, 0);
  }

    public Point(double a){
    //costruttore
      this.set(a, a, a);
  }

  public Point(double a, double b, double c){
    //costruttore da coordinate
```

```
    this.set(a, b, c);
    clr = new Point();
  }

  public Point(double q, double p, double d, double r, double g, double b){
    //costruttore con coordiante e colore
    this.set(q, p, d);
    clr = new Point(r,g,b);
  }

  public Point(Point P){
    // costruttore che copia un altro punto P
    this.set(P.x, P.y, P.z);
    clr = P.clr;
    normal = P.normal;
  }

  public void set(double a, double b, double c){
    // set delle coordinate e colore a grigio 70% circa
    x = a;
    y = b;
    z = c;
  }

  public void set(Point P){
    // set delle coordinate
    x = P.x;
    y = P.y;
    z = P.z;
  }

  public Point sum(Point P){
    // somma vettoriale
    return new Point(x + P.x, y + P.y, z + P.z);

  }

  public Point sum(double a){
    // somma vettoriale
    return new Point(x + a, y + a, z + a);

  }


  public void shift(Point v){
    // traslazione di vettore v
    x += v.x;
    y += v.y;
    z += v.z;
```

```
}

public void shift(double a, double b, double c){
  // traslazione da coordinate
  x += a;
  y += b;
  z += c;
}


public Point min(Point P){
  // differenza vettoriale
  return new Point(x - P.x, y - P.y, z - P.z);
}

public Point per(double l){
  // moltiplicazione per uno scalare l
  return new Point(x * l, y * l, z * l);
}

public Point star(Point P){
  return new Point(x*P.x, y*P.y, z*P.z);
}

public double dot(Point P){
  // prodotto scalare
  return x*P.x + y*P.y + z*P.z;
}

public Point vect(Point P){
  // prodotto vettoriale
 return new Point(this.y*P.z - this.z*P.y, this.z*P.x - this.x*P.z, this.x*P.y -
     this.y*P.x );
}

public Point to(Point P){
  // resituisce il vettore applicato che va "this" al punto P
  return P.min(this);
}

public double norm(){
  // norma euclidea
  return Math.sqrt(this.dot(this));
}

public double squareNorm(){
  // norma euclidea al quadrato
  return this.dot(this);
}
```

```
public Point normalize(){
  // noramlizzazione
  double n = this.norm();
  if(n == 0){
    return new Point(0,0,7);
  }
  return this.per( 1/n );
}

public Point normalizeL1(){
  // normalizzazione secondo la norma l1
  double n = Math.abs(x) + Math.abs(y) + Math.abs(z);
  if(n == 0) return this;
  return this.per(1/n);
}

public Point getNormal(){
  // getNormal() restituisce la normale, se esiste
  if(normal == null){
    System.out.println("Normal is null!:");
    return new Point(0,0,0);
  }
  return this.normal;
}

public void setNormal(Point v){
  // imposta normale al punto
  v = v.normalize();
  this.normal = v;
}

public static double clamp(double v, double min, double max){
  // clamp() costringe i valori di un double fra min e max
  if(v<min) v=min;
  if(v>max) v=max;
  return v;
}

public Point clamp(double min, double max){
  double xc = clamp(x, min, max);
  double yc = clamp(y, min, max);
  double zc = clamp(z, min, max);
  return new Point(xc, yc, zc);
}

public void print(){
 // stampa a schermo delle coordinate del punto
  System.out.println("p("+x+" "+y+" "+z+")");
```

```
}

public Color getColor(){
  Point c = (this.clr.per(255)).clamp(0, 255);
  return new Color(round(c.x), round(c.y), round(c.z));
}

public double getDistFrom(Point P){
  // distanza fra due punti
  Point V = this.to(P);
  return Math.sqrt(V.dot(V));
}

public Point project(double f){
  /* project() calcola la proiezione centrale di un punto sul piano z=0 rispetto
     al centro (0,0,-f)
   f rappresenta la distanza del viewplane dal punto di vista, in termini
      fotografici
   e' la lunghezza focale */

  double k = f/(f + z);
  Point projection = new Point(k*x, k*y, z); // risultato della matrice di
     proiezione, conservo la coordinata z dello spazio
  return projection;
}

public Point comb(double a, Point A, double b, Point B){
  // comb calcola la combinazione lineare di due vettori
  // usando i coefficienti a e b.
  return (A.per(a)).sum(B.per(b));
}

public static Point comb(double a, Point A, double b, Point B, double c, Point
    C){
  // comb calcola la combinazione lineare di tre vettori
  // usando i coefficienti a,b e c.
  Point p = new Point(a*A.x + b*B.x + c*C.x, a*A.y + b*B.y + c*C.y , a*A.z +
     b*B.z + c*C.z);
  return p;
}

public static Point average(Point P, Point Q){
  // punto medio fra due punti
  return (P.sum(Q)).per(0.5);
}

public static Point average(Point P, Point Q, Point U){
    // baricentro di tre punti
  return (P.sum(Q).sum(U)).per(1/3.0);
```

```
  }

  public static int round(double x){
    // round() arrotonda valori double all'intero piu' vicino
    // e' utilizzata per passare dalle corrdinate dello spazio (double)
    // alle coordinate della pixelMatrix (int)
    return (int) (Math.round(x));
  }

}
```

```
public class Triangle{
   public Point a;
   public Point b;
   public Point c;

   public Point center;
   public Point normal;

   public Triangle(Point x, Point y, Point z){
      a=x;
      b=y;
      c=z;

      if(a.normal != null && b.normal != null && c.normal != null)
         normal = Point.average(a.normal, b.normal, c.normal);
      else{
         this.makeNormal();
      }
      if(a != null && b != null && c != null)
         center = Point.average(a, b, c);
      if(a.clr != null && b.clr != null && c.clr != null)
         normal.clr = Point.average(a.clr, b.clr, c.clr);
   }

   public Point getNormal(){
      if(normal != null)
         return normal;
      else return this.makeNormal();

   }

   public Point makeNormal(){
      Point v = a.min(b);
      Point w = c.min(b);
      normal = w.vect(v).normalize();
      return normal;
   }
```

```
   public void print(){
      System.out.println("tr:");
      a.print();
      b.print();
      c.print();
   }

   public Point getCenter(){
      if(center != null)
         return center;
      Point cnt = Point.average(a,b,c);
      return cnt;
   }

   public Color getColor(){
      return normal.getColor();
   }

   public Triangle sum(Point p){
      return new Triangle(a.sum(p), b.sum(p), c.sum(p));
   }

   public void shift(Point v){
       // traslazione di vettore v
       a.shift(v);
       b.shift(v);
       c.shift(v);
       center = this.getCenter();
   }


}
```

```
public class Scene{
   public ArrayList<Point> pts; // Lista dei punti nella scena
   public ArrayList<Triangle> obj; // Lista dei triangoli nella scena

   public Scene(){
      pts = new ArrayList<Point>();
      obj = new ArrayList<Triangle>();

   }

   public Scene(ArrayList<Triangle> objects){
      obj = objects;
   }
```

```
public Triangle get(int i){
   return obj.get(i);
}

public void add(Point p){
   pts.add(p);
}

public void add(Triangle t){
   obj.add(t);
}

public void add(ArrayList<Triangle> list){
   for(int i=0; i<list.size(); i++)
      obj.add(list.get(i));
}

public int size(){
   return obj.size();
}


public void makeSphere(double r, Point c, int n, double R, double G, double B){
   /* makeSphere() genra una mesh sferica di triangonli
      la sfera ha raggio r, centro c ed e' divisa in n paralleli ed
      n meridiani. In tutto e' composta da n^2 triangoli.
      La sfera ha colore (R,G,B) */
   Point color = new Point(R,G,B);
   double x, y, z,a,b;
   double pi = Math.PI;
   int i,j;
   ArrayList<Point> tpts = new ArrayList<Point>();
   Triangle t;
   Point v1,v2,v3,p;
   for(i=0; i<n; i++){
      for(j=0; j<n; j++){
         x = r * Math.sin(i*pi/n) * Math.cos(2*j*pi/n) + c.x;
         y = r * Math.sin(i*pi/n) * Math.sin(2*j*pi/n) + c.y;
         z = -r * Math.cos(i*pi/n) + c.z;
         p = new Point(x,y,z);
         p.setNormal(c.to(p));
         this.add(p);
         tpts.add(p);
      }
   }

   for (i=0; i<n-1; i++) {
      for (j=0; j<n; j++) {
```

```
            t = new Triangle(tpts.get((i*n+j)%(n*n)), tpts.get(((i+1)*n+j)%(n*n)),
                tpts.get((i*n+j+1)%(n*n)));
            t.normal.clr = color;
            this.add(t);
            t = new Triangle(tpts.get(((i+1)*n+j)%(n*n)),
                tpts.get(((i+1)*n+j+1)%(n*n)), tpts.get((i*n+j+1)%(n*n)));
            t.normal.clr = color;
            this.add(t);
         }
      }
   }

   public void makeQuad(Point v0, Point l1, Point l2, double r, double g, double b){
      /* makeQuad genera un parallelogramma con vertice in v0, l1 e l2 come i due
          lati uscenti da v0,
      di colore (r,g,b)*/
      Point color = new Point(r,g,b);
      Triangle t;
      Point v1 = v0.sum(l1);
      Point v2 = v0.sum(l2);
      Point v3 = v1.sum(l2);
      pts.add(v0); pts.add(v1); pts.add(v2); pts.add(v3);
      t = new Triangle(v0, v1, v3);
      t.normal.clr = color;
      this.add(t);
      t = new Triangle(v0, v3, v2);
      t.normal.clr = color;
      this.add(t);
   }

}
```

```
public class RenderingEngine{
   public static Color[][] pixelMatrix;
   public static double[][] zBuffer;
   public static int f, w, h;
   public Point p1, p2, p3, p4,left,right;
   public static Point p[] = new Point[4];
   public static Point camera, view;

   public RenderingEngine(int width, int height, int focal){
      //Costruttore del renderer
      w = width;
      h = height;
      f = focal;
      view = new Point(0,0,f);
      pixelMatrix = new Color[w][h];
      zBuffer = new double[w][h];
```

```
}

public static void render(Scene scn){
   // render() esegue tutte le funzioni necessarie a fere il rendering della
       scena

   // inizializzo la matrice dei pixel con il colore dello sfondo
   // e quella dello z-buffer con la profonditÃ  massima
   for(int i=0; i<w; i++)
      for(int j=0; j<h; j++){
         pixelMatrix[i][j] = new Color(0,0,0);
         zBuffer[i][j] = 99999;
      }

   // per ogni triangolo della scena, calcolo il suo rendering
   for(int i=0; i<scn.size(); i++){
      render(scn.get(i), scn);
   }
}

public static void render(Triangle tr, Scene scn){
   // Questa funzione trova la proiezione del triangolo tr sul viewplane
   // e aggiorna i valori delle matrici pixelMatrix e zBuffer


   double x,y,z;
   Point n = new Point(tr.getNormal()); // normale del triangolo tr

   if(n.z>0) return; // backface culling

   //calcolo delle proiezioni dei tre vertici del triangolo sul view plane
   p[1] = tr.a.project(f);
   p[2] = tr.b.project(f);
   p[3] = tr.c.project(f);

   // eseguo scambi per far sÃŹ che p[1] sia il vertice piu' in alto sul
       viewplane (coordinata y piÃč bassa),
   // p[2] al centro e p[3] sia il piÃč basso
   if(p[1].y>p[2].y) swap(1,2);
   if(p[1].y>p[3].y) swap(1,3);
   if(p[2].y>p[3].y) swap(2,3);


   double zSlopeX = -n.x / n.z; // Ãš l'incremento di z per ogni spostamento di
       1 pixel lungo l'asse x
   double zSlopeY = -n.y / n.z; // Ãš l'incremento di z per ogni spostamento di
       1 pixel lungo l'asse y
```

```
   /* Adesso occorre fare un clipping del triangolo proiettato. Infatti la
       funzione scan() che disegna
      il triangolo su pixelMatrix funziona solo se uno dei lati ĂŠ parallelo
          all'asse x del view plane.
      Si opera dunque un taglio orizzionatle all'altezza del vertice p[2]
          ottentedo due triangol (uno
      superiore e uno inferiore), entrambi con un lato orizzontale (in comune)
   */

   // Caso banale: se due vertici hanno la stessa y, allora un lato ĂŠ
       orizzontale. Posso dunque eseguire
   // direttamente la funzione scan() e disegnare l'intero triangolo
   if(p[1].y == p[2].y){
      if(p[1].x>p[2].x) swap(1,2);
      scan(p[3], p[1], p[2], -1, zSlopeX, zSlopeY, tr);
      return;
   }

   // Nel caso generale devo trovare l'intersezione del triangolo proiettato sul
       viewplane (2D) con
   // la retta orizzontale passante per p[2]. Un punto d'interzezione ĂŠ
       ovviamente p[2], l'altro
   // si trova sul lato opposto e si ottiene tramite interpolazione lineare

   y = (p[2].y - p[1].y) / (p[1].y - p[3].y); //coefficiente di interpolazione
       lineare

   p[0] = new Point(y*(p[1].x - p[3].x) + p[1].x , p[2].y, y*(p[1].z - p[3].z) +
       p[1].z); // punto d'intersezione

   // Ora bisogna disegnare i due triangoli (ora entrambi con un lato
       orizzontale).
   // Opero degli scambi affinche' p[1],p[2],p[0] sia il triangolo superiore e
   // p[3],p[2],p[0] quello inferiore
   if(p[2].x>p[0].x) swap(2,0);

   // si possono ora disegnare le due porzioni di triangolo con la funzione
       scan()
   scan(p[1], p[2], p[0], 1, zSlopeX, zSlopeY, tr);
   scan(p[3], p[2], p[0], -1, zSlopeX, zSlopeY, tr);
}

public static int round(double x){
   // round() arrotonda valori double all'intero piu' vicino
   // e' utilizzata per passare dalle corrdinate dello spazio (double)
   // alle coordinate della pixelMatrix (int)
   return (int) (Math.round(x));
}
```

```
public static void scan(Point p1, Point p2, Point p3, int v, double zSlopeX,
    double zSlopeY, Triangle tr){
   /* scan() trova, con il calolo incrementale, tutti i pixel interni del
       triangolo p1,p2,p3 e per ognuno
      di essi richiama draw(), la funzione che aggiorna di fatto le matrici
         pixelMatrix e zBuffer.
      Il valore 'v' indica se si sta operando lo scan di un triangolo superiore
         (+1) o inferiore (-1) del clipping
   */

   // Se si considera la retta che passa lungo il lato sinistro del triangolo,
   //   ls rappresente l'incremento di x per ogni spostamento di 1 pixel lungo
       l'asse y del view plane
   // (dunque l'inverso del coefficiente angolare)
   double ls = (p1.x - p2.x) / (p1.y - p2.y);

   // rs rappresenta la stessa quantita' per il lato destro del triangolo
   double rs = (p1.x - p3.x) / (p1.y - p3.y);

   double x, y, z, xl, xr, zx;
   int i, j;
   Color c = tr.normal.getColor(); // ottengo il colore del triangolo (salvato
       come colore della normale)

   // Nel caso in cui v=+1, si sta facendo lo scan del triangolo superiore,
       dall'alto verso il basso
   // (dunque dal vertice in alto verso il lato orizzontale)
   if(v==1){
      z = p1.z;
      xl = xr = p1.x;
      for(y=p1.y; (y)<=p3.y; y++){
         // questo for() viene eseguito per ogni riga del triangolo
         zx = z;
         for(x=xl; x<=xr; x++){
            //questo for() viene eseguito per ogni pixel della stessa riga

            // con draw(), scrivo sulla matrice di pixel il valore del colore c
               nel punto x,y del viweplane
            // e aggiorno in tali coordinate anche il valore dello zBuffer
            draw(x, y, zx, c);

            // ottengo il nuovo valore di profondita' per la prossima iterazione
               (pixel di destra)
            zx += zSlopeX;
         }
         z += ls*zSlopeX + zSlopeY; // ottengo il nuovo valore di profondita'
             per la prossima iterazione (riga sottostante)
         xl += ls; // ottengo l'estremo sinistro dei valori di x per la prossima
             riga
```

```
            xr += rs; // ottengo l'estremo destro dei valori di x per la prossima
                riga
            // xl e xr sono gli estremi della riga per il ciclo interno, sono usati
                nella condizione
            // di permanenza del ciclo for() interno
         }
      }

      // Nel caso in cui v=-1, si sta facendo lo scan del triangolo inferiore,
          dall'alto verso il basso
      // (dunque dal lato orizzontale verso il vertice in basso)
      // le operazioni sono analoghe
      if(v==-1){
         z = p3.z;
         xl = p2.x;
         xr = p3.x;
         for(y=p3.y; (y)<=p1.y; y++){
            zx = z;
            for(x=xl; x<=xr; x++){
               draw(x, y, zx, c);
               zx += zSlopeX;
            }
            z += ls*zSlopeX;
            z += zSlopeY;
            xl += ls;
            xr += rs;
         }
      }
   }

   public static void draw(double xd, double yd, double z, Color c){
      // Se necessario, la funzione draw() aggiorna pixelMatrix con il colore c nel
          punto (xd,yd)
      // e inoltre aggiorna lo zBuffer nello stesso punto con il valore z

      // passo dalle coordinate dello spazio a quelle del viewplane (in pixel)
      int x = round(xd + w/2);
      int y = round(yd + h/2);

      /* Se le coordinate (x,y) del pixel sono contenute nel viewplane
       e la coordinata z di profondita' e' minore di quella gia' presente nello
       zBuffer, allora aggiorno i valori nelle due matrici con il colore e la
           profondita'
       passate come argomento*/
      if(x<w && x>=0 && y>=0 && y<h && z<zBuffer[x][y]){
         zBuffer[x][y] = z;
         pixelMatrix[x][y] = c;
      }
   }
```

```
   public static void swap(int i, int j){
      // swap() e' una funzione che scambia i nomi (puntatori) di due oggetti di
          tipo Point
      Point t;
      t = new Point(p[i]);
      p[i] = new Point(p[j]);
      p[j] = new Point(t);
   }
}
```

1.9. Appendice: accelerazione del Ray Tracing tramite $z-$buffer e suo codice Java

La fase che rallenta di più il Ray Tracing è il calcolo delle intersezioni di ciascun raggio proiettore con tutti i poligoni della scena. In compenso, rispetto allo $z-$buffer che non conserva informazione sulla disposizione spaziale dei poligoni proiettati sul viewplane, il Ray Tracing conserva questa informazione ed è quindi in grado di raffigurare effetti di illuminazione che dipendono dalla posizione delle sorgenti di luce e dell'osservatore rispetto al punto osservato, e quindi un maggior realismo. Ma possiamo ottenere lo stesso realismo saltando la suddetta fase onerosa, rimpiazzando il Ray Tracing con uno $z-$buffer (che, come osservato, ha un tempo di esecuzione che, per scene usuali, non cresce molto quando il numero di poligoni diventa elevato, e non richiede intersezioni di raggi con la scena).
Questa accelerazione del Ray Tracing tramite $z-$buffer si fa precalcolando i valori di illuminazione di ciascun poligono (ad esempio tramite i modelli di illuminazione di Lambert e di Phong, che verranno spiegati nel prossimo Capitolo, rispettivamente nelle Sezioni 2.2 e 3.2), e poi proiettiamo i poligoni sul viewplane tramite $z-$buffer: nel buffer dei poligoni viene allora memorizzato, per ciascun pixel, il valore precalcolato di illuminazione del poligono proiettato su quel pixel).

Il codice è una estensione semplice di quello presentato nella Sezione 1.9: per chiarezza, lo riportiamo interamente, anziché limitarci alle classi che sono state modificate. Anche questo codice è stato sviluppato da G. Nazzaro e comunicato in [**32**].
In questi codici, la scelta del linguaggio Java, invece che del più consueto linguaggio C++, è dovuta, oltre che alla maggiore facilità, anche al fatto che Java, essendo un linguaggio parzialmente interpretato invece che completamente compilato, produce applicazioni molto più lente di C++: ma nonostante questo, l'implementazione qui proposta. grazie alla velocizzazione tramite $z-$buffer ed all'uso di tessiture rotanti, è molto veloce, al punto da produrre una animazione fotorealistica in tempo reale. Per una implementazione in C++, si veda la successiva Appendice 5.9.

L'immagine di due sfere prodotta da questo renderer di Ray Tracing tramite $z-$buffer è in Figura 1.9.1.

```
import java.awt.*;
import java.util.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;

public class Main extends JFrame{
   public MyJPanel canvas;
   public int f;
   public static Scene scn;
   public static Main sv;

   public Main(String s){
   // Main si occupa solo di far eseguire il programma e gestire l'interfaccia
       grafica.
   // Non e' interessante dal punto di vista teorico. Non e' pienamente commentato.
```

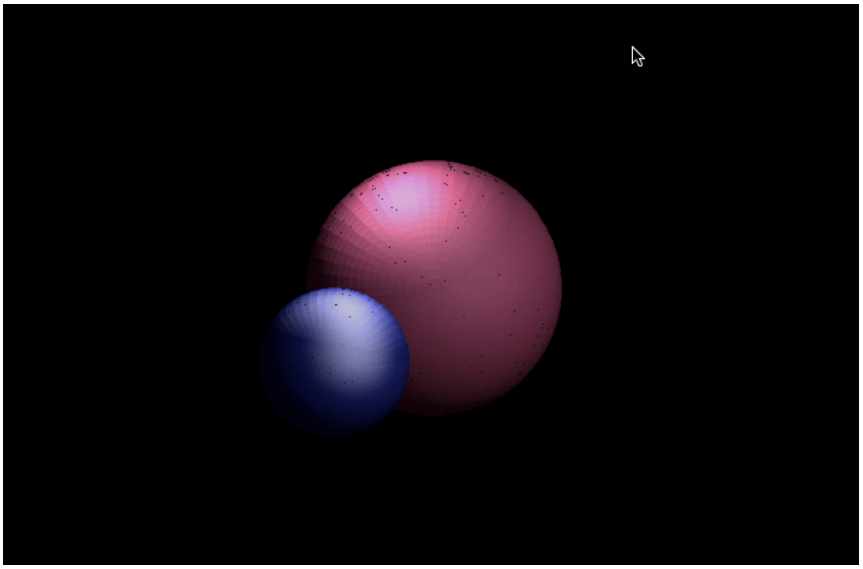


FIGURA 1.9.1. L'immagine di una scena che consiste di due sfere, prodotta dal renderer di Ray Tracing velocizzato tramite $z-$buffer della Sezione **??**. L'illuminazione di ciascun triangolo della modellazione a maglie delle sfere è precalcolata in base all'equazione di Phong per le superficie diffusive e semiriflettenti (Sezione 3.2) una volta assegnata la posizione di una sorgente di luce. Nel caso che si voglia realizzare una animazione, la posizione tridimensionale dei triangoli può cambiare ad ogni frame, e quindi i dati dell'equazione di Phong non vengono precalcolati una volta per tutte ma ricalcolati ogni volta, recuperando il valore di profondità dallo $z-$buffer. Il colore risultante di ciascun triangolo viene assegnato al buffer dei poligoni al momento della proiezione sul viewplane operata dallo $z-$buffer

```
super(s);
f = 700;
canvas = new MyJPanel(f);
this.add(canvas);
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
this.pack();
scn = new Scene(0.1);

scn.makeSphere(300, new Point(0,0, 1300), 50, 0.8, 0.3, 0.5, 0.1, 0.9);
/* il primo parametro e' il raggio della sfera;
l'argomento Point e' il centro: la variabile y
 e' l'altezza e cresce verso il basso,
z e' la profondita' spaziale e cresce all'allontanarsi
dal viewplane, che e' a z=0;
il parametro successivo e' il numero di ripartizioni
della latitudine e della longitudine nel creare
la modellazione a maglie triangolari della sfera;
i tre parametri successivi sono le coordinate R, G, B del colore, fra 0 e 1;
infine abbiamo il coefficiente di riflessione e quello di diffusione.
*/
```

```
scn.makeSphere(150, new Point(-200, 150, 1000), 50, 0.2, 0.2, 0.8, 0.8, 0.7);



final Light l1 = new Light(-300, -400, 1000, 150);
Light l2 = new Light(600, -400, 0, 350);
l2.l=l1.l=0.5;
l2.q=l1.q=0;
scn.lgt.add(l1);
scn.lgt.add(l2);

this.setVisible(true);
this.addKeyListener(new KeyListener() {
   public double d = 0.5;
   public double r = 0.5;
    public double diff = d;
      public double ref = r;
      public Point v;

   @Override public void keyPressed(KeyEvent e) {
       if(e.getKeyCode() == KeyEvent.VK_UP) v = new Point(0,-10,0);
       if(e.getKeyCode() == KeyEvent.VK_DOWN) v = new Point(0,10,0);
       if(e.getKeyCode() == KeyEvent.VK_LEFT) v = new Point(-10,0,0);
       if(e.getKeyCode() == KeyEvent.VK_RIGHT) v = new Point(10,0,0);

       if(e.getKeyCode() == KeyEvent.VK_BACK_SPACE) v = new Point(0,0,-10);
       if(e.getKeyCode() == KeyEvent.VK_SPACE) v = new Point(0,0,10);

       l1.shift(v);
      repaint();
    }

    @Override public void keyTyped(KeyEvent e) {
       v=new Point(0,0,0);
       if(e.getKeyChar() == 'w') v = new Point(0,-20,0);
       if(e.getKeyChar() == 's') v = new Point(0,20,0);
       if(e.getKeyChar() == 'a') v = new Point(-20,0,0);
       if(e.getKeyChar() == 'd') v = new Point(20,0,0);
       if(e.getKeyChar() == 'l') diff = -(diff - 0.5);
       if(e.getKeyChar() == 'p') ref = -(ref - 0.5);

     System.out.println(diff+" "+ref);
       for(int i=0; i<scn.size(); i++){
        scn.pts.get(i).shift(v.per(-1));
        scn.pts.get(i).kDif = diff;
        scn.pts.get(i).kRef = ref;
       }

       scn.lgt.get(0).shift(v.per(-1));
```

```
            repaint();

         }

          @Override public void keyReleased(KeyEvent e) {
         }
      });

   }

   public static void main(String args[]){

         RenderingEngine R = new RenderingEngine(720, 480, 700);
      Main C = new Main("Rendered");

   }
}



class MyJPanel extends JPanel{
   public ArrayList<Point> GList = new ArrayList<Point>();
   public ArrayList<Point> PList = new ArrayList<Point>();
   public Point p1 = new Point(300,300,20);
   public float r, g, b;
   public static int f,j,i,h,w;

   public MyJPanel(int f){
      super();
      this.f = f;
      w = 720;
      h = 480;
      this.setPreferredSize(new Dimension(w, h));
      this.setVisible(true);
      //PList.add(p1);
   }

    @Override public void paintComponent(Graphics gr){
      super.paintComponent(gr);
      gr.setColor(new Color(0,0,0));
      gr.fillRect(0, 0, this.getWidth(), this.getHeight());
      RenderingEngine.render(Main.scn);
      for(i=0; i<w; i++)
         for(j=0; j<h; j++){
            gr.setColor(RenderingEngine.pixelMatrix[i][j]);
            gr.drawLine(i,j,i,j);
         }
   }
```

```
}
```

```
public class Point{

  // coordinate del punto nello spazio
  public double x;
  public double y;
  public double z;

  // coefficiente diffusivo e riflessivo
  public double kRef,kDif;

  // colore
  public Point clr;

  // versore normale al punto
  public Point normal;

  public Point(){
    //costruttore
      this.set(0, 0, 0);
  }

    public Point(double a){
    //costruttore
      this.set(a, a, a);
  }

  public Point(double a, double b, double c){
    //costruttore da coordinate
    this.set(a, b, c);
    clr = new Point();
    clr.set(0.5, 0.2, 0.8);
    kRef = 0.3;
    kDif = 0.7;
  }

  public Point(double q, double p, double d, double r, double g, double b){
    //costruttore con coordiante e colore
    this.set(q, p, d);
    clr = new Point(r,g,b);
  }

  public Point(Point P){
    // costruttore che copia un altro punto P
    this.set(P.x, P.y, P.z);
    clr = P.clr;
    normal = P.normal;
```

```
}

public void set(double a, double b, double c){
  // set delle coordinate e colore a grigio 70% circa
  x = a;
  y = b;
  z = c;
}

public void set(Point P){
  // set delle coordinate
  x = P.x;
  y = P.y;
  z = P.z;
}

public Point sum(Point P){
  // somma vettoriale
  return new Point(x + P.x, y + P.y, z + P.z);

}

public Point sum(double a){
  // somma vettoriale
  return new Point(x + a, y + a, z + a);

}


public void shift(Point v){
  // traslazione di vettore v
  x += v.x;
  y += v.y;
  z += v.z;
}

public void shift(double a, double b, double c){
  // traslazione da coordinate
  x += a;
  y += b;
  z += c;
}


public Point min(Point P){
  // differenza vettoriale
  return new Point(x - P.x, y - P.y, z - P.z);
}
```

```
public Point per(double l){
  // moltiplicazione per uno scalare l
  return new Point(x * l, y * l, z * l);
}

public Point star(Point P){
  return new Point(x*P.x, y*P.y, z*P.z);
}

public double dot(Point P){
  // prodotto scalare
  return x*P.x + y*P.y + z*P.z;
}

public Point vect(Point P){
  // prodotto vettoriale
 return new Point(this.y*P.z - this.z*P.y, this.z*P.x - this.x*P.z, this.x*P.y -
     this.y*P.x );
}

public Point to(Point P){
  // resituisce il vettore applicato che va da "this" al punto P
  return P.min(this);
}

public double norm(){
  // norma euclidea
  return Math.sqrt(this.dot(this));
}

public double squareNorm(){
  // norma euclidea al quadrato
  return this.dot(this);
}

public Point normalize(){
  // noramlizzazione
  double n = this.norm();
  if(n == 0){
    return new Point(0,0,7);
  }
  return this.per( 1/n );
}

public Point normalizeL1(){
  // normalizzazione secondo la norma l1
  double n = Math.abs(x) + Math.abs(y) + Math.abs(z);
  if(n == 0) return this;
  return this.per(1/n);
```

```
}

public Point getNormal(){
  // getNormal() restituisce la normale, se esiste
  if(normal == null){
    System.out.println("Normal is null!:");
    return new Point(0,0,0);
  }
  return this.normal;
}

public void setNormal(Point v){
  // imposta normale al punto
  v = v.normalize();
  this.normal = v;
}

public void clamp(double min, double max){
 // clamp() costringe i valori del vettore fra min e max
  x = clamp(x, min, max);
  y = clamp(y, min, max);
  z = clamp(z, min, max);
}

public static double clamp(double v, double min, double max){
  // clamp() costringe i valori di un double fra min e max
  if(v<min) v=min;
  if(v>max) v=max;
  return v;
}

public void print(){
 // stampa a schermo delle coordinate del punto
  System.out.println("p("+x+" "+y+" "+z+")");
}



public double getDistFrom(Point P){
  // distanza fra due punti
  Point V = this.to(P);
  return Math.sqrt(V.dot(V));
}

public Point project(double f){
  /* project() calcola la proiezione centrale di un punto sul piano z=0 rispetto
     al centro (0,0,-f)
   f rappresenta la distanza del viewplane dal punto di vista, in termini
      fotografici
```

```
    e' la lunghezza focale */

    double k = f/(f + z);
    Point projection = new Point(k*x, k*y, z); // risultato della matrice di
        proiezione, conservo la coordinata z dello spazio
    projection.setNormal(this.normal);
    return projection;
  }

  public Point comb(double a, Point A, double b, Point B){
    // comb calcola la combinazione lineare di due vettori
    // usando i coefficienti a e b.
    return (A.per(a)).sum(B.per(b));
  }

  public static Point comb(double a, Point A, double b, Point B, double c, Point
      C){
    // comb calcola la combinazione lineare di tre vettori
    // usando i coefficienti a,b e c.
    Point p = new Point(a*A.x + b*B.x + c*C.x, a*A.y + b*B.y + c*C.y , a*A.z +
        b*B.z + c*C.z);
    return p;
  }

  public static Point average(Point P, Point Q){
    // punto medio fra due punti
    return (P.sum(Q)).per(0.5);
  }

  public static Point average(Point P, Point Q, Point U){
      // baricentro di tre punti
    return (P.sum(Q).sum(U)).per(1/3.0);
  }


}
```

```
public class Triangle{
   public Point a;
   public Point b;
   public Point c;

   public Point center;
   public Point normal;

   public Triangle(Point x, Point y, Point z){
      a=x;
      b=y;
```

```
      c=z;

      if(a.normal != null && b.normal != null && c.normal != null)
         normal = Point.average(a.normal, b.normal, c.normal);
      else
         this.makeNormal();
      if(a != null && b != null && c != null)
         center = Point.average(a, b, c);
      if(a.clr != null && b.clr != null && c.clr != null)
         normal.clr = Point.average(a.clr, b.clr, c.clr);
   }

   public Point getNormal(){
      if(normal != null)
         return normal;
      else return this.makeNormal();

   }

   public Point makeNormal(){
      Point v = a.min(b);
      Point w = c.min(b);
      normal = w.vect(v).normalize();
      return normal;
   }

   public void print(){
      System.out.println("tr:");
      a.print();
      b.print();
      c.print();
   }

   public Point getCenter(){
      if(center != null)
         return center;
      Point cnt = Point.average(a,b,c);
      return cnt;
   }

   public Triangle sum(Point p){
      return new Triangle(a.sum(p), b.sum(p), c.sum(p));
   }

   public void shift(Point v){
       // traslazione di vettore v
       a.shift(v);
       b.shift(v);
       c.shift(v);
```

```
        center = this.getCenter();
    }


}
```

```
public class Scene{
    public ArrayList<Point> pts; // Lista dei punti nella scena
    public ArrayList<Triangle> obj; // Lista dei triangoli nella scena
    public ArrayList<Light> lgt; // Lista delle luci nella scena
    public Point amb; // Colore della luce ambientale


    public Scene(Point ambientLight){
        amb = ambientLight;
        pts = new ArrayList<Point>();
        obj = new ArrayList<Triangle>();
        lgt = new ArrayList<Light>();

    }

    public Scene(double ambientLight){
        amb = new Point(ambientLight);
        pts = new ArrayList<Point>();
        obj = new ArrayList<Triangle>();
        lgt = new ArrayList<Light>();

    }

    public Scene(ArrayList<Triangle> objects, ArrayList<Light> lights, Point
         ambientLight){
        obj = objects;
        lgt = lights;
        amb = ambientLight;
    }

    public Triangle get(int i){
        return obj.get(i);
    }

    public void add(Point p){
        pts.add(p);
    }

    public void add(Triangle t){
        obj.add(t);
    }
```

```
   public void add(ArrayList<Triangle> list){
      for(int i=0; i<list.size(); i++)
         obj.add(list.get(i));
   }

   public void addLight(Light l){
      lgt.add(l);
   }


   public int size(){
      return obj.size();
   }
```

La seguente funzione makeSphere genera una sfera ottenuta da una maglia data da un array bidimensionale di triangoli, e ne precalcola il colore in base all'equazione di illuminazione di Phong (2.4.3)*. In alternativa, oppure obbligatoriamente nel caso si voglia realizzare una animazione nella quale i triangoli, le luci o l'osservatore si spostano, si può calcolare il colore solo del triangolo visibile attraverso ciascun pixel, utilizzando il valore di profondità dello z−buffer per calcolare le coordinate spaziali del punto osservato attraverso quel pixel e, grazie a queste, applicare l'equazione di illuminazione di Phong).*

```
public void makeSphere(double r, Point c, int n, double R, double G, double B,
    double ref, double dif){
      /* makeSphere() genera una mesh sferica di triangoli
         la sfera ha raggio r, centro c ed e' divisa in n paralleli ed
         n meridiani. In tutto e' composta da n^2 triangoli.
         La sfera ha colore (R,G,B) e coefficiente riflessivo e diffusivo
            rispettivamente pari a ref e dif. */


      // costruzione dei vertici della maglia:
      Point color = new Point(R,G,B);
      double x, y, z,a,b;
      double pi = Math.PI;
      int i,j;
      ArrayList<Point> tpts = new ArrayList<Point>();
      Triangle t;
      Point v1,v2,v3,p;
      for(i=0; i<n; i++){
         for(j=0; j<n; j++){
            x = r * Math.sin(i*pi/n) * Math.cos(2*j*pi/n) + c.x;
            y = r * Math.sin(i*pi/n) * Math.sin(2*j*pi/n) + c.y;
            z = -r * Math.cos(i*pi/n) + c.z;
            p = new Point(x,y,z);
            p.setNormal(c.to(p));
            this.add(p);
            tpts.add(p);
         }
```

```
    }


    // costruzione e colorazione dei triangoli:
    for (i=0; i<n-1; i++) {
       for (j=0; j<n; j++) {
          t = new Triangle(tpts.get((i*n+j)%(n*n)), tpts.get(((i+1)*n+j)%(n*n)),
              tpts.get((i*n+j+1)%(n*n)));
          t.normal.clr = color;
          t.center.kRef = ref;
          t.center.kDif = dif;
          this.add(t);
          t = new Triangle(tpts.get(((i+1)*n+j)%(n*n)),
              tpts.get(((i+1)*n+j+1)%(n*n)), tpts.get((i*n+j+1)%(n*n)));
          t.normal.clr = color;
          t.center.kRef = ref;
          t.center.kDif = dif;
          this.add(t);
       }
    }
  }

}


class Light extends Point{
  // sorgente luminosa puntiforme

  public double i; // intensit\‘{a} della sorgete luminosa
  public double q, l, c; // coefficienti di decadimento {quadratico, lineare e
      costante} */

  public Light(double x, double y, double z, double intensity){
   super(x, y, z);
   i = intensity;
   c=0;
   q = 1;
   l = 0;
  }

  public double getAtt(double r){
      // getAtt() restituisce l'attenuazione dovuta dalla distanza, usando i
          coefficienti
      // caratteristici della sorgete luminosa
    double att = (c + l*r + q*r*r);
    if(att<1)
        return 1;
    return 1/att;
  }
```

```
}
```

```
class RenderingEngine{
   public static Color[][] pixelMatrix;
   public static double[][] zBuffer;
   public static int f, w, h;
   public Point p1, p2, p3, p4,left,right;
   public static Point p[] = new Point[4];
   public static Point camera, view;

   public RenderingEngine(int width, int height, int focal){
      //Costruttore del renderer
      w = width;
      h = height;
      f = focal;
      view = new Point(0,0,f);
      pixelMatrix = new Color[w][h];
      zBuffer = new double[w][h];
   }

   public static void render(Scene scn){
      // render() esegue tutte le funzioni necessarie al rendering della scena

      // inizializzo la matrice dei pixel con il colore dello sfondo
      // e quella dello z-buffer con la profonditĂ  massima
      for(int i=0; i<w; i++)
         for(int j=0; j<h; j++){
            pixelMatrix[i][j] = new Color(0,0,0);
            zBuffer[i][j] = 99999;
         }

      // per ogni triangolo della scena, calcolo il suo rendering
      for(int i=0; i<scn.size(); i++){
         render(scn.get(i), scn);
      }
   }

   public static void render(Triangle tr, Scene scn){
      // Questa funzione trova la proiezione del triangolo tr sul viewplane
      // e aggiorna i valori delle matrici pixelMatrix e zBuffer


      double x,y,z;
      Point n = new Point(tr.getNormal()); // normale del triangolo tr

      if(n.z>0) return; // backface culling

      Color c = getShade(tr, scn);
```

```
//calcolo delle proiezioni dei tre vertici del triangolo sul view plane
p[1] = tr.a.project(f);
p[2] = tr.b.project(f);
p[3] = tr.c.project(f);

// eseguo scambi per far sÄŹ che p[1] sia il vertice piu' in alto sul
    viewplane (coordinata y piÄč bassa),
// p[2] al centro e p[3] sia il piÄč basso
if(p[1].y>p[2].y) swap(1,2);
if(p[1].y>p[3].y) swap(1,3);
if(p[2].y>p[3].y) swap(2,3);


double zSlopeX = -n.x / n.z; // Äš l'incremento di z per ogni spostamento di
    1 pixel lungo l'asse x
double zSlopeY = -n.y / n.z; // Äš l'incremento di z per ogni spostamento di
    1 pixel lungo l'asse y

/* Adesso occorre fare un clipping del triangolo proiettato. Infatti la
    funzione scan() che disegna
   il triangolo su pixelMatrix funziona solo se uno dei lati Äš parallelo
       all'asse x del view plane.
   Si opera dunque un taglio orizzionatle all'altezza del vertice p[2]
       ottentedo due triangol (uno
   superiore e uno inferiore), entrambi con un lato orizzontale (in comune)
*/

// Caso banale: se due vertici hanno la stessa y, allora un lato Äš
    orizzontale. Posso dunque eseguire
// direttamente la funzione scan() e disegnare l'intero triangolo
if(p[1].y == p[2].y){
   if(p[1].x>p[2].x) swap(1,2);
   scan(p[3], p[1], p[2], -1, zSlopeX, zSlopeY, tr);
   return;
}

// Nel caso generale devo trovare l'intersezione del triangolo proiettato sul
    viewplane (2D) con
// la retta orizzontale passante per p[2]. Un punto d'interzezione Äš
    ovviamente p[2], l'altro
// si trova sul lato opposto e si ottiene tramite interpolazione lineare

y = (p[2].y - p[1].y) / (p[1].y - p[3].y); //coefficiente di interpolazione
    lineare

p[0] = new Point(y*(p[1].x - p[3].x) + p[1].x , p[2].y, y*(p[1].z - p[3].z) +
    p[1].z); // punto d'intersezione
```

```
    // Ora bisogna disegnare i due triangoli proiettati sul viewplane (ora
       entrambi con un lato orizzontale).
    // Opero degli scambi affinche' p[1],p[2],p[0] sia il triangolo superiore e
    // p[3],p[2],p[0] quello inferiore
    if(p[2].x>p[0].x) swap(2,0);

    // si possono ora disegnare le due porzioni di triangolo con la funzione
       scan()
    scan(p[1], p[2], p[0], 1, zSlopeX, zSlopeY, tr);
    scan(p[3], p[2], p[0], -1, zSlopeX, zSlopeY, tr);
  }

  public static int round(double x){
    // round() arrotonda valori double all'intero piu' vicino
    // e' utilizzata per passare dalle coordinate dello spazio (double)
    // alle coordinate della pixelMatrix (int)
    return (int) (Math.round(x));
  }

  public static void scan(Point p1, Point p2, Point p3, int v, double zSlopeX,
     double zSlopeY, Triangle tr){
    /* scan() trova, con un procedimento incrementale, tutti i pixel interni del
       triangolo p1,p2,p3 e per ognuno
      di essi richiama draw(), la funzione che aggiorna di fatto le matrici
         pixelMatrix e zBuffer.
      Il valore 'v' indica se si sta operando lo scan di un triangolo superiore
         (+1) o inferiore (-1) del clipping
    */

    // Se si considera la retta che passa lungo il lato sinistro del triangolo,
    //   ls rappresente l'incremento di x per ogni spostamento di 1 pixel lungo
       l'asse y del view plane
    // (dunque l'inverso del coefficiente angolare)
    double ls = (p1.x - p2.x) / (p1.y - p2.y);

    // rs rappresenta la stessa quantit\'{a} per il lato destro del triangolo
    double rs = (p1.x - p3.x) / (p1.y - p3.y);

    double x, y, z, xl, xr, zx;
    int i, j;
    Color c = getShade(tr, MainZBuffer.scn); // trovo il valore d'illuminazione
       del triangolo nelle scena

    // Nel caso in cui v=+1, si sta facendo lo scan del triangolo superiore,
       dall'alto verso il basso
    // (dunque dal vertice in alto verso il lato orizzontale)
    if(v==1){
      z = p1.z;
      xl = xr = p1.x;
```

```
      for(y=p1.y; (y)<=p3.y; y++){
         // questo for() viene eseguito per ogni riga del triangolo
         zx = z;
         for(x=xl; x<=xr; x++){
            //questo for() viene eseguito per ogni pixel della stessa riga

            // con draw(), scrivo sulla matrice di pixel il valore del colore c
               nel punto x,y del viweplane
            // e aggiorno in tali coordinate anche il valore dello zBuffer
            draw(x, y, zx, c);

            // ottengo il nuovo valore di profondit\‘{a} per la prossima
               iterazione (pixel di destra)
            zx += zSlopeX;
         }
         z += ls*zSlopeX + zSlopeY; // ottengo il nuovo valore di profondit\‘{a}
             per la prossima iterazione (riga sottostante)
         xl += ls; // ottengo l’estremo sinistro dei valori di x per la prossima
             riga
         xr += rs; // ottengo l’estremo destro dei valori di x per la prossima
             riga
         // xl e xr sono gli estremi della riga per il ciclo interno, sono usati
             nella condizione
         // di permanenza del ciclo for() interno
      }
   }

   // Nel caso in cui v=-1, si sta facendo lo scan del triangolo inferiore,
       dall’alto verso il basso
   // (dunque dal lato orizzontale verso il vertice in basso)
   // le operazioni sono analoghe
   if(v==-1){
      z = p3.z;
      xl = p2.x;
      xr = p3.x;
      for(y=p3.y; (y)<=p1.y; y++){
         zx = z;
         for(x=xl; x<=xr; x++){
            draw(x, y, zx, c);
            zx += zSlopeX;
         }
         z += ls*zSlopeX;
         z += zSlopeY;
         xl += ls;
         xr += rs;
      }
   }
}
```

```
public static void draw(double xd, double yd, double z, Color c){
   // Se necessario, la funzione draw() aggiorna pixelMatrix con il colore c nel
       punto (xd,yd)
   // e inoltre aggiorna lo zBuffer nello stesso punto con il valore z

   // passo dalle coordinate dello spazio a quelle del viewplane (in pixel)
   int x = round(xd + w/2);
   int y = round(yd + h/2);

   /* Se le coordinate (x,y) del pixel sono contenute nel viewplane
    e la coordinata z di profondit\‘{a} e’ minore di quella gi\‘{a} presente
        nello
    zBuffer, allora aggiorno i valori nelle due matrici con il colore e la
        profondit\‘{a}
    passate in argomento*/
   if(x<w && x>=0 && y>=0 && y<h && z<zBuffer[x][y] && z>0){
      zBuffer[x][y] = z;
      pixelMatrix[x][y] = c;
   }
}

public static void swap(int i, int j){
   // swap() e’ una funzione che scambia i nomi (puntatori) di due oggetti di
       tipo Point
   Point t;
   t = new Point(p[i]);
   p[i] = new Point(p[j]);
   p[j] = new Point(t);
}


public static Color getShade(Triangle tr, Scene scn){
   // getShade() ottiene l’illuminazione del trianagolo tr nella scena,
       calcolando
   // il contributo ambientale, diffusivo e riflessivo
   Point N = tr.getNormal(); // ottengo la normale del triangolo
   Point p = tr.getCenter(); // ottengo il centro del triangolo
   return getShade(p, N, scn);
}
```

Il prossimo metodo getShade restituisce il colore dei triangoli in base al metodo di illuminazione di Lambert–Phong.

```
public static Color getShade(Point p, Point N, Scene scn){
   // getShade ottiene l’illuminazione del punto p nella scena, con normale N,
   // calcolando il contributo ambientale, diffusivo (Lambert) e riflessivo
       (Phong)
```

```
// Per il solo z-buffer, questa classe non serviva: lo z-buffer possiede come
    input
// il colore di ciascun triangolo, senza doverlo ricavare dai dati geometrici
    3D,
// ossia dalla posizione di osservatore, luci e versore normale.
// Ma qui passiamo allo z-buffer, per ciascun poligono, un colore appropriato:
// quello che si ottiene da questi dati geometrici applicando la equazione di
// illuminazione di Phong. Quindi questo metodo di z-buffer e' in realta' una
// accelerazione tramite z-buffer del Ray Tracing (non ricorsivo).

// inizializzazione di variabili
double intensity;
double d, att, diff=0,shadeDiff=0,shadeRef=0,ref=0;
Light light=null;
Point amb = scn.amb; // amb e' il colore della luce ambientale
Point shade;
Point clr = N.clr; // il colore del triangolo e' salvato nella sua normale
Point shadeDiffFinal = new Point(0,0,0);
Point shadeRefFinal = new Point(0,0,0);
Point L, V;
Point o = new Point(0,0,-f); // o rappresenta il centro di proiezione, ed e'
    dunque l'origine
                  // di tutti i raggi di vista


for(int i=0; i<scn.lgt.size(); i++){
   // per ogni luce della scena, calcolo il suo contributo d'illuminazione

   light = scn.lgt.get(i);

   // V e' il vettore che congiunge il centro del triangolo al centro di
       proiezione o
   V = p.to(o);

   // L e' il vettore che congiunge il centro del triangolo alla posizione
       della luce
   // (chiamato anche 'raggio d'ombra')
   L = p.to(light);

   // d e' la distanza del triangolo dalla luce
   d = L.norm();

   // att e' il coefficiente di attenuazione della luce dato dalla distanza d
   att = light.getAtt(d);

   if(L.dot(N)<0){
      // Se la superficie del triangolo non 'vede' la luce,
      // non calcolo affatto il contributo di questa sorgente e
      // salto direttamente alla prossima luce
```

```
            continue;
        }

        intensity = light.i * att; // intensity e' l'intensit\'{a} luminosa che
            raggiunge il punto

        L = L.normalize(); // normalizzo il raggio d'ombra
        V = V.normalize(); // normalizzo il raggio di vista
        // L e V sono ora versori

        /* diff e' il contributo d'illuminazione, diffusivo calcolato con la
            formula
           di Lambert: diff = <L,N>. Notare che a questo punto della funzione diff
               non
           puo' essere negativo per il controllo effettuato prima */
        diff = L.dot(N);

        shadeDiff += diff*intensity; // incremento il contributo diffusivo

        /* ref e' il contributo di luce riflettente calcolato con la formula
           di Phong: ref = 2 <N,L> <N,V> - <L,V> */
        ref = 2*N.dot(L)*N.dot(V) - L.dot(V);
        // se ref e' positivo, incremento l'illuminazione di ref, moltiplicata per
            il coefficiente di riflessione al punto
        if(ref>0) shadeRef += ref*ref*intensity;
    }

    shadeDiff *= p.kDif; // moltiplico diff per il coefficiente diffusivo del
        punto e quello d'attenuazione
    shadeRef *= p.kRef; // moltiplico diff per il coefficiente diffusivo del
        punto e quello d'attenuazione
    shadeDiffFinal = clr.per(shadeDiff);
    shadeRefFinal = new Point(shadeRef, shadeRef, shadeRef);

    shade = shadeRefFinal.sum(shadeDiffFinal);

    //amb.print();

    shade = shade.sum((amb.star(clr)).per(p.kDif)); // incremento del contributo
        ambientale
    // normalizzo a valori compresi fra 0 e 255
    shade = shade.per(255);
    // contengo l'illuminazione fra il valore massimo (255) e minimo (0)
    shade.clamp(0,255);
    //intShade = round(shade); // arrotondo all'intero piu' vicino per la
        codifica RGB

    return new Color(round(shade.x), round(shade.y), round(shade.z)); //
        restituisco l'illuminazione finale
```

```
    }
}
```

CAPITOLO 2

L'equazione dell'illuminazione

Ci proponiamo di stabilire, sulla base per ora di modelli empirici, l'intensità di luce nei punti visibili all'osservatore delle superficie che compongono la scena, sulla base del loro colore e del colore e della posizione delle sorgenti di luce e della posizione dell'osservatore. Considereremo varie componenti di illuminazione:

- Luce ambientale
- Illuminazione diffusa
- Attenuazione con la distanza
- Illuminazione speculare
- Illuminazione da sorgenti non puntiformi

Infine, considereremo modelli fisici di illuminazione.

2.1. Luce ambientale

Questa componente consiste di una luce di fondo presente nella scena e che illumina ciascun oggetto in maniera non direzionale, in conseguenza alla diffusione della luce causata dalle riflessioni multiple fra gli oggetti della scena. In seguito studieremo una modellazione matematica delle gradazioni di luce di queste componenti diffuse (radiosità), ma per ora ci limitiamo a considerare un modello molto semplificato in cui ad ogni materiale è associata una costante di riflessione ambientale k_a (nell'intervallo $0 < k_a < 1$) che misura quale percentuale della illuminazione ambientale viene diffusa da quel materiale. Per maggiore precisione potremmo far dipendere k_a anche dalla frequenza della luce incidente, ovvero dalla sua lunghezza d'onda λ, e quindi scrivere $k_{a,\lambda}$, ma, poiché in seguito studieremo modelli fisici precisi che chiariscono questa dipendenza, per ora spesso ignoreremo la dipendenze della costante di riflessione ambientale dalla lunghezza d'onda della luce. Sia $O_{d,\lambda}$ la distribuzione spettrale del colore della superficie osservata (quando la si illumina con luce bianca), e $I_{a,\lambda}$ quella della luce ambientale. Allora la componente ambientale dell'illuminazione è

$$I_\lambda = I_{a,\lambda} k_a O_{d,\lambda} \, .$$

2.2. Illuminazione diffusa: il modello di Lambert

Questa componente modella la diffusione della luce incidente che proviene da una data direzione $\boldsymbol{L}$ ($\boldsymbol{L}$ è il versore della direzione dal punto osservato sulla superficie alla posizione della sorgente di luce). Assumiamo per semplicità che ci sia una sola sorgente (se ce ne sono di più si devono sommare i corrispondenti contributi), e che essa sia puntiforme ed isotropa (cioè che l'intensità emessa non dipenda dalla direzione di emissione). Vogliamo modellare la diffusione da materiali opachi, come il gesso o il polistirolo. Il modello empirico che presentiamo, che si chiama modello di riflessione di Lambert, assume che l'intensità diffusa da una piccola porzione piana di superficie non dipenda dalla posizione dell'osservatore. Si assume

peraltro che la sorgente di luce e l'osservatore stiano entrambi all'esterno dell'oggetto racchiuso dalla superficie, cioè stiano nel semispazio frontale del punto osservato: questo equivale a dire che l'angolo formato dalla direzione dell'osservatore e la normale uscente sia minore di 90 gradi, ed altrettanto per l'angolo formato dalla direzione della luce e la normale uscente. In caso contrario, l'illuminazione di Lambert, ed anche tutte le altre forme di illuminazione, si pongono uguali a zero. Naturalmente, però, l'illuminazione deve dipendere dall'angolo θ formato dalla posizione della sorgente e la normale $\boldsymbol{N}$ alla superficie, perché un fascio di luce incidente con sezione, diciamo, A, illumina una porzione piana di superficie su un'area tanto più grande quanto più radente è la sua direzione $\boldsymbol{L}$ di provenienza, e precisamente pari a $A/\cos\theta$ (per il momento stiamo considerando fasci di raggi paralleli, come se la sorgente fosse all'infinito; in seguito tratteremo l'attenuazione dovuta alla distanza). Pertanto il fattore di attenuazione della luce incidente da una direzione $\boldsymbol{L}$ ad angolo θ rispetto alla direzione normale è

$$\mathrm{Att}(\theta) = \cos\theta = \langle \boldsymbol{L}, \boldsymbol{N} \rangle. \tag{2.2.1}$$

Quindi, se l'intensità emessa dalla sorgente puntiforme al variare della lunghezza d'onda λ è $I_{p,\lambda}$, la densità di energia luminosa sulla superficie si ottiene dal prodotto scalare $I_{p,\lambda}\,\langle \boldsymbol{L}, \boldsymbol{N} \rangle = I_{p,\lambda}\cos\theta$, e l'equazione dell'illuminazione combinata ambientale e diffusa diventa

$$I_\lambda = I_{a,\lambda} k_a O_{d,\lambda} + I_{p,\lambda} k_d O_{d,\lambda} \langle \boldsymbol{L}, \boldsymbol{N} \rangle,$$

dove k_d è un opportuno coefficiente di diffusione, con valore nell'intervallo $0 < k_d < 1$, e che può dipendere dalla lunghezza d'onda.
Si noti che il coefficiente di diffusione dipende solo dal materiale e non dal punto ivi osservato, ma l'intensità di luce diffusa varia da punto a punto, perché variano il versore normale $\boldsymbol{N}$ e la direzione $\boldsymbol{L}$ dal punto osservato alla sorgente.

2.3. Attenuazione con la distanza

L'esposizione precedente non tiene conto che i raggi di luce uscenti da una sorgente puntiforme a distanza d finita di distribuiscono in maniera isotropa, e quindi l'illuminazione si spande su una sfera di raggio $d = d_L$. Di conseguenza l'illuminazione che emana da una sorgente puntiforme (non quella ambientale!) decresce proporzionalmente all'area della sfera, cioè proporzionalmente a d^{-2}. Nel caso della luce del sole, la cui grande distanza rende irrilevanti le variazioni di distanza da punto a punto della scena, l'illuminazione è di intensità costante, ed i raggi solari sono paralleli. Ma per le sorgenti a distanza d_L finita dobbiamo moltiplicare il termine di Lambert per un fattore di attenuazione f_{att}:

$$I_\lambda = I_{a,\lambda} k_a O_{d,\lambda} + f_{\mathrm{att}} I_{p,\lambda} k_d O_{d,\lambda} \langle \boldsymbol{L}, \boldsymbol{N} \rangle,$$

dove $f_{\mathrm{att}} = c/d_L^2$. In realtà, però, questo termine crea contrasti troppo forti: ombre troppo scure e luci saturate. In parte questo è conseguenza del fatto che quasi nessuna sorgente di luce è puntiforme, e quelle non puntiformi causano, almeno a distanza comparabile con il loro diametro, un decadimento dell'illuminazione di ordine meno elevato di d^{-2}. Perciò miglioriamo il modello scegliendo un fattore di attenuazione che sia combinazione lineare di termini di grado 2, 1 e 0 (costante): $f_{\mathrm{att}} = 1/(c_0 + c_1 d_L + c_2 d_L^2)$. Ma dobbiamo comunque imporre che questo coefficiente sia di attenuazione e non di aumento, cioè non superiore a 1: quindi poniamo

$$f_{\mathrm{att}} = \min\left\{ \frac{1}{c_0 + c_1 d_L + c_2 d_L^2}\ , 1 \right\}.$$

2.3.1. Attenuazione atmosferica. Un altro effetto che può essere interessante generare è quello dell'attenuazione atmosferica, dovuta alla distanza non della luce ma dell'osservatore, per modellare fenomeni di foschia. Questo si realizza attenuando l'illuminazione progressivamente con la profondità z. Vogliamo modificare l'illuminazione interpolandola fra i seguenti due valori: quello precedentemente calcolato, I_λ , ed un valore I_0 di illuminazione minima dello sfondo che scegliamo e fissiamo. Per questo scopo si fissano un piano frontale $z = z_1$ ed un piano di fondo $z = z_0$, ovviamente con $z_0 < z_1$, e si scelgono due livelli di attenuazione atmosferica s_b e s_f, il primo per il piano di fondo ed il secondo per il piano frontale, con $s_b < s_f$; poi si interpola linearmente fra s_b e s_f per ottenere un valore s_z che cresce all'aumentare della profondità, cioè all'avvicinarsi all'osservatore (ricordiamo che l'asse z è orientato verso l'osservatore). Per valori di z inferiori alla profondità dello sfondo, cioè $z < z_0$, l'attenuazione rimane costantemente uguale a quella del piano di fondo, cioè s_b, mentre per $z > z_1$ l'attenuazione rimane uguale a quella del piano frontale, cioè s_f. Il calcolo di s_z è immediato, grazie all'equazione della retta che passa per i punti (z_0, s_b) e (z_1, s_f): si ottiene

$$s_z = s_b + (z - z_0)\,\frac{s_f - s_b}{z_1 - z_0}\ .$$

Una volta calcolato il fattore di attenuazione s_z, l'interpolazione attenuata si ottiene dall'interpolazione convessa fra l'illuminazione non attenuata e quella del piano di fondo:

$$I'_\lambda = s_z I_\lambda + (1 - s_z) I_0.$$

2.4. Riflessione speculare: il modello di Phong

La riflessione speculare è la riflessione della luce su una superficie che avviene nella direzione speculare (rispetto alla normale esterna alla superficie) alla direzione di incidenza. Per materiali non perfettamente speculari, la luce si riflette non solo nella direzione speculare a quella di incidenza, ma anche in direzioni ad essa vicine, generando un alone (highlight). Per alcune superficie, come quelle lucidate a cera o la plastica (nella quale i pigmenti di colore sono immersi all'interno di un materiale trasparente) questa luce che si riflette alla superficie ma non all'interno ha il colore della luce incidente. Per altri materiali, come i metalli, le leggi dell'ottica mostrano che il materiale può cambiare il colore della luce riflessa: vedremo in seguito un modello di riflessione basato sulle leggi fisiche.

2.4.1. La direzione del raggio riflesso. Per modellare la riflessione speculare in maniera euristica (cioè non basata su modelli fisici, per il momento), anzitutto calcoliamo la direzione della luce riflessa. La proiezione del versore $\boldsymbol{L}$ sul versore normale $\boldsymbol{N}$ vale $\cos\theta \boldsymbol{N}$, dove θ è l'angolo di incidenza. Perciò il vettore $\boldsymbol{S}$ da L al piede della proiezione vale $\boldsymbol{S} = \cos\theta\, \boldsymbol{N} - \boldsymbol{L}$. Quindi il versore della direzione riflessa, $\boldsymbol{R}$, vale

$$\boldsymbol{L} + 2\boldsymbol{S} = 2\cos\theta\, \boldsymbol{N} - \boldsymbol{L}.$$

D'altra parte, poiché $\boldsymbol{L}$ e $\boldsymbol{N}$ sono versori, cioè vettori di lunghezza 1, il coseno dell'angolo θ che essi formano è il loro prodotto scalare:

$$\cos\theta = \langle \boldsymbol{L}, \boldsymbol{N} \rangle.$$

Quindi si ottiene:

$$\boldsymbol{R} = 2\langle \boldsymbol{N}, \boldsymbol{L} \rangle \boldsymbol{N} - \boldsymbol{L} \tag{2.4.1}$$

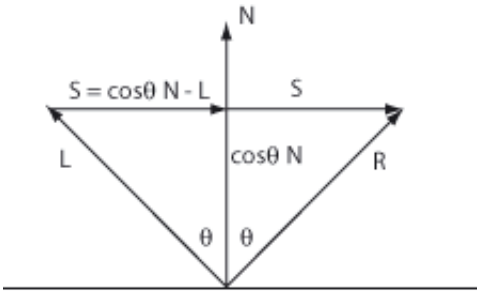


FIGURA 2.4.1. Il versore $\boldsymbol{R}$ è il riflesso speculare di $\boldsymbol{L}$

Ora consideriamo l'angolo α formato dal versore riflesso e dalla direzione V dell'osservatore. Questo angolo misura la deviazione angolare dalla direzione di massima intensità di riflessione.

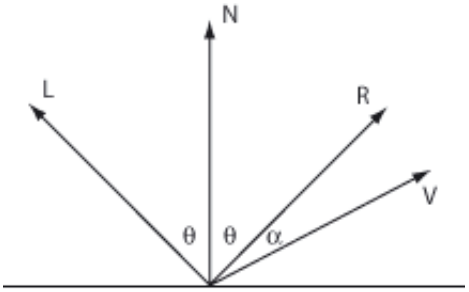


FIGURA 2.4.2. L'angolo α è la deviazione della direzione di osservazione $\boldsymbol{V}$ dal versore riflesso $\boldsymbol{R}$

Analogamente a prima, si ha

$$\cos\alpha = \langle \boldsymbol{R}, \boldsymbol{V} \rangle$$

e quindi

$$\cos\alpha = \langle \boldsymbol{R}, \boldsymbol{V} \rangle = 2\langle \boldsymbol{N}, \boldsymbol{L} \rangle \langle \boldsymbol{N}, \boldsymbol{V} \rangle - \langle \boldsymbol{L}, \boldsymbol{V} \rangle \tag{2.4.2}$$

Se l'angolo fra i versori R e V supera l'angolo retto, allora l'illuminazione data da questo modello diventerebbe negativa, il che non ha senso fisico: in tal caso si pone l'illuminazione uguale a zero (si vedano maggiori dettagli nella prossima Sottosezione 2.4.2).

2.4.2. Il modello di illuminazione riflessa di Phong. Questo modello, introdotto in **[34]**, descrive in maniera euristica la riflessione da materiali non perfettamente speculari, per i quali quindi si hanno componenti di luce riflessa non solo nella direzione speculare ma anche in direzioni vicine, con una deviazione α non nulla, le quali naturalmente diventano progressivamente più deboli. Il modello di Phong utilizza un decadimento rispetto ad α proporzionale a $\cos^n \alpha$: poiché $\cos 0 = 1$ ma $0 < \cos\alpha < 1$ per $0 < |\alpha| < \pi/2$, tutte le funzioni $\cos^n \alpha$ hanno valore 1 per $\alpha = 0$, ma tendono a zero quando n tende a infinito per α diverso da 0. Quindi, per n = 1, il decadimento è blando, e gli aloni sono alquanto diffusi, ma per

grandi valori di n il decadimento è rapido e l'alone è concentrato: scompare al superamento di una soglia piccola della deviazione α dalla direzione di massima riflessione speculare. Il valore di n da scegliere dipende solo dalla natura del materiale, e più precisamente dal suo livello di specularità. La frazione di energia luminosa riflessa può dipendere non solo da α ma anche dall'angolo di incidenza θ, quindi dovremmo introdurre un fattore di attenuazione della riflessione speculare $k(\theta)$ con $0 < k(\theta) < 1$: ma per semplicità assumiamo che il fattore k di attenuazione speculare sia indipendente da θ, e lo denotiamo con k_s. Ovviamente, come sempre, se $|\theta| > \pi/2$, la sorgente di luce sta dietro la superficie ed il valore dell'illuminazione è posto uguale a zero. A questo proposito, osserviamo che il modello di Phong si trova nell'imbarazzante necessità di decidere cosa fare quando $|\alpha| > \pi/2$, caso nel quale $\cos\alpha = \langle \boldsymbol{R}, \boldsymbol{V} \rangle < 0$. In questo caso l'equazione dell'illuminazione di Phong non può valere perché termini negativi per l'illuminazione non hanno senso fisico. Il modo più indolore di eliminare il problema è di porre il termine di Phong uguale a zero se $\langle \boldsymbol{R}, \boldsymbol{V} \rangle < 0$, cioè se $|\alpha| > \pi/2$. Si noti che questa modellazione non è fisicamente corretta, perché può succedere che l'angolo α fra la direzione di visuale e quella del raggio riflesso sia maggiore di $\pi/2$(si veda la figura 2.4.3), ma in tal caso nella realtà l'osservatore vede comunque un valore di illuminazione non nulla. Per risolvere questo aspetto in maniera fisicamente corretta bisogna utilizzare modelli più complessi, non euristici ma fisici, come faremo in seguito.

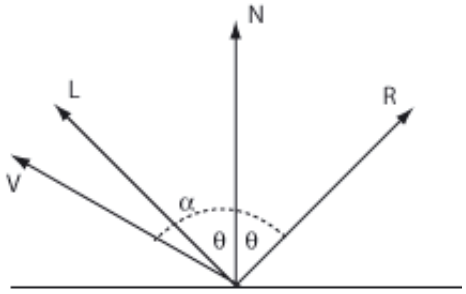


FIGURA 2.4.3. Un esempio in cui l'angolo di deviazione α è maggiore di 90 gradi

Ora l'equazione dell'illuminazione diventa, grazie a (2.4.2):

$$I_\lambda = I_{a,\lambda} k_a O_{d,\lambda} + f_{\text{att}} I_{p,\lambda} \left(k_d O_{d,\lambda} \langle \boldsymbol{L}, \boldsymbol{N} \rangle + k_s \langle \boldsymbol{R}, \boldsymbol{V} \rangle^n \right) \tag{2.4.3}$$

Si noti che la componente di illuminazione riflessa non ha un colore proprio, bensì quello della luce incidente. Quindi, per quanto osservato prima, questa equazione di illuminazione si presta a modellare la riflessione da superficie come la plastica o le superficie smaltate con cera o vernice a smalto. Se volessimo introdurre un colore speculare, potremmo modificare (2.4.3) inserendo un fattore $O_{s,\lambda}$ di distribuzione spettrale del colore riflesso:

$$I_\lambda = I_{a,\lambda} k_a O_{d,\lambda} + f_{\text{att}} I_{p,\lambda} \left(k_d O_{d,\lambda} \langle \boldsymbol{L}, \boldsymbol{N} \rangle + k_s O_{s,\lambda} \langle \boldsymbol{R}, \boldsymbol{V} \rangle^n \right) \tag{2.4.4}$$

Così facendo il modello tende a rappresentare ogni superficie riflettente come se fosse metallica invece che di plastica. In realtà, però, la resa del colore della riflessione ottenuta in questo modo non è molto accurata, perché è difficile trovare le distribuzioni spettrali adeguate: per renderla verosimile è ancora una volta indispensabile ricorrere a modelli fisici per la riflessione speculare e diffusa. Inoltre, il prodotto $I_{p,\lambda}\, O_{s,\lambda}$ vale identicamente zero quando i due fattori hanno supporti disgiunti, ossi sono colori consistenti di due range frequenze che

non si sovrappongono: in tal caso la precedente equazione implica che non c'è alcun contributo di riflessione, ma nella realtà fisica non è così. Infine, la luce radente riflessa ha certamente, in natura, lo stesso colore della luce incidente, e non il colore del materiale riflettente (ragione per cui, anche in questo modello euristico e non fisico, si preferisce usare (2.4.3) invece di (2.4.4)

2.5. Sorgenti di luce non puntiformi: riflettori di Warn

Come già accennato, normalmente le sorgenti di luce non sono puntiformi, neppure in via di approssimazione. Nelle riprese fotografiche e cinematografiche si usano riflettori, o banchi di luce, in cui la distribuzione angolare dell'emissione di luce non è isotropa. Un modo di estendere il modello di illuminazione di Phong per includere questi riflettori è stato introdotto da David Warn [**51**]. Al posto della sorgente puntiforme in direzione $\boldsymbol{L}$ si immagina di disporre un riflettore, modellato come una superficie piana illuminata da una nuova sorgente puntiforme disposta frontalmente ad esso, in direzione $\boldsymbol{L'}$; indichiamo con $I_{L',\lambda}$ la distribuzione spettrale della luce emessa da questo illuminatore.

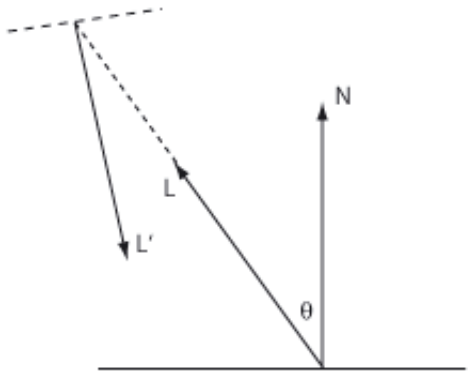


FIGURA 2.5.1. Un illuminatore non puntiforme è modellato come un pannello diffusivo (linea tratteggiata) ed una sorgente virtuale posta frontalmente rispetto ad esso (in direzione $\boldsymbol{L'}$)

Questa nuova sorgente non contribuisce all'illuminazione della scena altro che per il fatto di illuminare il riflettore, il quale a sua volta ne riflette la luce con distribuzione angolare in accordo al modello di illuminazione di Phong: ad un angolo γ di deviazione rispetto alla normale $\boldsymbol{L'}$ del riflettore, la luce emessa è $I_{L',\lambda} \cos^n \gamma$. D'altra parte, è immediato dalla Figura 2.5.1 che $\cos\gamma = \langle -\boldsymbol{L'}, \boldsymbol{L}\rangle$, e quindi la luce emanata dal riflettore è determinata dalla seguente equazione dell'illuminazione di Warn:

$$I_{L',\lambda} \langle -\boldsymbol{L'}, \boldsymbol{L}\rangle^n.$$

Si possono considerare due varianti del riflettore di Warn che modellano sorgenti di luce di uso comune in fotografia:

- il primo consiste dell'illuminatore con alette di focalizzazione (*flaps*). Le alette oscurano lateralmente il fascio di luce incanalandolo su una banda. Per modellare questo illuminatore basta porre uguale a zero l'illuminazione al di fuori di una banda prestabilita. Ad esempio, per incanalare la luce in una banda trasversale all'asse x, diciamo

da x_- a x_+, basta lasciare l'illuminazione come deriva dall'equazione di Warn nei punti (x, y, z) per i quali $x_- < x < x_+$, e porla uguale a zero per gli altri punti dello spazio (quelli esterni a questa banda);
- il secondo consiste nel modellare una luce spot in una direzione prefissata. A questo scopo basta annullare la luce al di fuori di un cono centrato in questa direzione, cioè per i punti dello spazio che eccedono una determinata deviazione rispetto all'asse determinato da questa direzione.

2.6. Modelli fisici di illuminazione

2.6.1. Microsfaccettature di una superficie. Il modello di Torrance e Sparrow, introdotto in [**45**] and [**46**], è un modello di superficie riflettenti basato su leggi fisiche. Blinn lo applicò alla Computer Graphics, paragonandolo al modello di Phong, in [**4**]. In seguito, Cook e Torrance [**12**] utilizzarono questo modello per ottenere la composizione spettrale della luce riflessa. Nel modello di Torrance–Sparrow la superficie è immaginata come un insieme di microscopiche sfaccettature che costituiscono riflettori perfettamente speculari. La distribuzione, la geometria di tali microsfaccettature e la direzione della luce (immaginata come emessa da una sorgente infinitamente distante, cosicché tutti i raggi si possono approssimare paralleli) determinano l'intensità e la direzione della riflessione speculare come una funzione dell'intensità I della sorgente luminosa puntiforme e del versore normale $\boldsymbol{N}$, del versore dal punto illuminato verso la posizione della sorgente di luce $\boldsymbol{L}$ e di quello verso la posizione dell'osservatore $\boldsymbol{V}$. Verifiche sperimentali hanno mostrato un'ottima corrispondenza con tale modello.
Il modello fa uso di una adeguata espressione per la *riflettività bidirezionale* speculare, ossia per l'intensità di luce che proviene da una direzione assegnata e viene riflessa specularmente in un'altra direzione assegnata. Rinviamo la definizione formale della riflettività bidirezionale al seguito, nello studio della Radiosità ed ancor meglio della Illuminazione Globale.
Abbiamo già incontrato un modello di illuminazione euristico e non fisico, il modello di illuminazione di Phong, che assume che questo coefficiente di riflettività bidirezionale sia proporzionale ad una opprtuna potenza del coseno fra la direzione di osservazione e la direzione speculare alla luce incidente. Questo termine deve essere moltiplicato per un coefficiente di riflettività che misura il grado di specularità della superficie riflettente, ovvero quanta percentuale della luce incidente viene riflessa: tale coefficiente dovrebbe dipendere dall'angolo di incidenza, ma spesso si assume costante indipendentemente dall'angolo (quindi una proprietà solo del materiale), per semplicità ed ancor più perché, in un modello euristico non basato su leggi fisiche, non si è in grado di ipotizzare una adeguata dipendenza dall'angolo.
Invece il modello di Torrance–Sparrow, che è basato su ipotesi fisiche, assume la seguente componente speculare della riflettività bidirezionale legata alla statistica della distribuzione angolare delle microsfaccettature (per una discussione lievemente più approfondita si veda la Sottosezione 7.7.9 in seguito):

$$\rho_s = \frac{F_\lambda}{\pi} \frac{DG}{\langle \boldsymbol{N}, \boldsymbol{V} \rangle \langle \boldsymbol{N}, \boldsymbol{L} \rangle} \tag{2.6.1}$$

Qui $\boldsymbol{N}$ è il versore normale alla superficie al punto di riflessione, $\boldsymbol{V}$ è il versore che individua la direzione dell'osservatore, ed i numeri F_λ, D e G, che calcoliamo nelle prossime Sezioni, rappresentano rispettivamente il termine di Fresnel delle leggi della riflessione dell'ottica fisica

(Sezione 2.6.4), il coefficiente di attenuazione dovuto alla distribuzione angolare delle microsfaccettature (Sezione 2.6.2), ed il fattore di attenuazione geometrica che misura la probabilità che la luce inviata verso una microsfaccettatura ad un'altra sia oscurata da un'altra che si interpone (Sezione 2.6.3). Il fattore π al denominatore è introdotto al fine di tenere in considerazione l'aumento dell'area, rispetto ad una superficie piana ideale, dovuto alla ruvidità. Infatti su una superficie perfettamente piana e speculare un fascio cilindrico di raggi paralleli di luce incidente da una data direzione fissa copre un'area circolare, ossia un disco; invece in presenza di microsfaccettature perfettamente speculari ma orientate diversamente fra loro con un gran numero di possibili angoli lo stesso fascio copre un'area maggiore, più vicina a quella dell'emisfero sotteso dal disco, perché le microsfaccettature tendono a disporsi in maniera non parallela ma distribuita al variare dell'angolo. Per una discussione pi'u dettagliata si veda il calcolo relativo alla normalizzazione dell'integrazione sulla semisfera frontale di un emettitore diffusivo nella Sottosezione7.4.1 in seguito. Infine, il fattore $\boldsymbol{N} \cdot \boldsymbol{V}$ al denominatore serve ad aumentare l'area della superficie illuminata per tener conto della sua proiezione nella direzione dell'osservatore, ed analogamente $\boldsymbol{N} \cdot \boldsymbol{L}$ è il fattore di proiezione nella direzione della luce: questi fattori misurano la attenuazione della riflessione bidirezionale dovuta al fatto che la luce incidente copre un'area maggiore della sezione del fascio (quindi dà luogo ad una densità di illuminazione minore), e l'osservatore vede una sezione di area minore dell'area illuminata, a causa dell'inclinazione rispetto alla normale sia dell'una sia dell'altro.

2.6.2. Funzione di distribuzione delle microsfaccettature. Poiché le microsfaccettature sono considerate perfettamente speculari, il modello tiene in considerazione solo quelle le cui normali giacciono lungo la direzione bisettrice fra le direzioni di incidenza e di uscita (ossia di visualizzazione): si tratta della direzione del vettore $\boldsymbol{H} = \frac{1}{2}(\boldsymbol{L} + \boldsymbol{V})$ (che, una volta normalizzato, si chiama lo *halfway vector*) . Solo una frazione D del totale delle microsfaccettatura ha tale orientamento. Torrance e Sparrow ipotizzano una funzione di distribuzione gaussiana. Altri lavori usano distribuzioni diverse: ad esempio Cook e Torrance usano una distribuzione più adeguata, la distribuzione di Beckmann, che per le superficie ruvide dà come risultato

$$D = \frac{1}{4m^2 \cos^4 \beta} e^{-[(\tan \beta)m]^2} \tag{2.6.2}$$

in cui β è l'angolo tra $\boldsymbol{N}$ ed $\boldsymbol{H}$, m è lo scarto quadratico medio della pendenza delle microsfaccettature (ossia m^2 è la varianza della distribuzione delle inclinazioni). Se m è piccolo allora l'inclinazione delle microsfaccettature varia poco rispetto alla normale della superficie e dunque la riflessione è molto a fuoco sulla direzione speculare (questo è il caso, per esempio, per uno specchio piano). Invece, per m grande l'inclinazione è elevata e la superficie ruvida diffonde in maniera uniforme la luce riflessa su tutto l'emisfero frontale (questo è il caso dei riflettori di Lambert, per esempio il gesso).

Per tenere conto di superficie con varie scale di ruvidità, si usa una combinazione convessa delle funzioni di distribuzione,

$$D = \sum_{j=1}^{n} w_j D(m_j) \tag{2.6.3}$$

dove la somma dei pesi w_j è pari a 1.

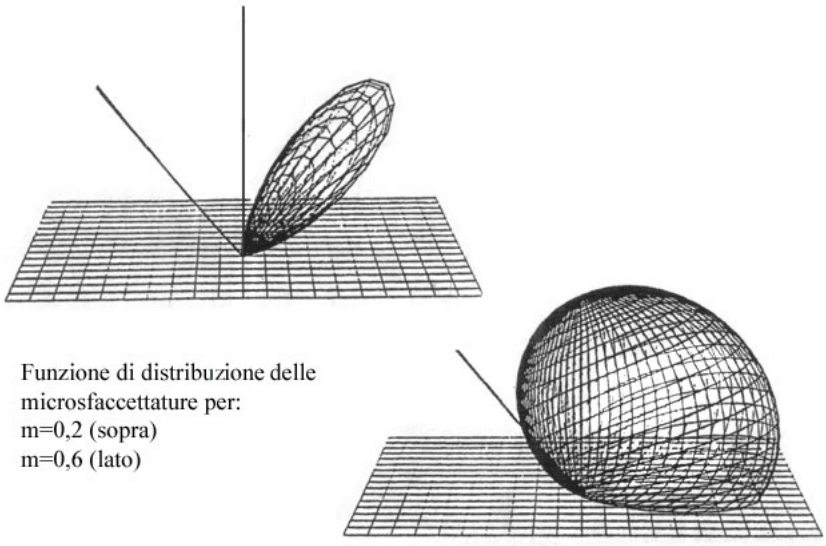


FIGURA 2.6.1. Diagramma polare della riflessione a seconda della funzione di distribuzione delle microsfaccettature

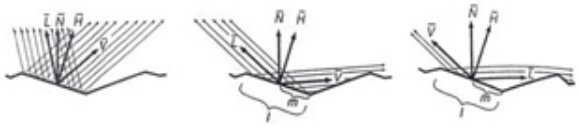


FIGURA 2.6.2. Due casi di ostruzione geometrica

2.6.3. Il fattore di attenuazione geometrica. Il modello tiene in considerazione il fatto che, rispetto alla sorgente o ad un'altra microsfaccettatura, ci siano microsfaccettature che ne coprono altre, occludendo la luce ad esse inviata dalla prima, o dalla sorgente. Questo fatto dà luogo ad un fattore di attenuazione geometrica G. Torrance e Sparrow, e Blinn, considerano tre differenti situazioni nel calcolo di G:

(a) La luce incidente sulla microsfaccettatura è totalmente riflessa e non colpisce più la stessa superficie.
(b) La microsfaccettatura è totalmente esposta alla luce incidente ma quella riflessa viene parzialmente o totalmente intercettata da altre microsfaccettature (questa luce intercettata ed a sua volta riflessa contribuisce alla riflessione diffusa).
(c) La microsfaccettatura è parzialmente schermata dalla luce incidente

Il fattore geometrico di attenuazione varia da 0 (occlusione totale) ad 1 (nessuna occlusione).

Nel primo caso, nel quale tutta la luce incidente viene riflessa, poniamo il fattore di attenuazione geometrica G uguale ad 1. In entrambi gli altri casi la quantità di luce intercettata da altre sfaccettature è pari a M/A, dove A è l'area totale della microsfaccettatura e M è l'area la cui luce riflessa è bloccata. Quindi $G_b = G_c = 1 - M/A$.

La proporzione di luce riflessa nel secondo caso è stata calcolata nel succitato articolo di Blinn ed il risultato è:

$$G_b = \frac{2\langle \boldsymbol{N}, \boldsymbol{H}\rangle \langle \boldsymbol{N}, \boldsymbol{V}\rangle}{\langle \boldsymbol{V}, \boldsymbol{H}\rangle} \tag{2.6.4}$$

Qui $\boldsymbol{L}$ è il versore che individua la direzione della sorgente ed $\boldsymbol{H} = 1/2(\boldsymbol{L} + \boldsymbol{V})$ è il versore normale della microsfaccettatura (prima non l'avevamo normalizzato perché facevamo solo riferimento alla sua direzione).

La proporzione di luce riflessa G_c nel terzo caso si ottiene in modo analogo: basta osservare che in effetti la situazione è identica alla precedente tranne per il fatto che ora la luce bloccata non è quella uscente nella direzione $\boldsymbol{V}$ dell'osservatore, bensì quella incidente, ovvero proveniente dalla direzione di $\boldsymbol{L}$. Quindi basta rimpiazzare al numeratore il fattore di attenuazione dato dalla proiezione nella direzione dell'osservatore, $\langle \boldsymbol{N}, \boldsymbol{V}\rangle$, con quello nella direzione della sorgente di luce, $\langle \boldsymbol{N}, \boldsymbol{L}\rangle$. Ovvero:

$$G_c = \frac{2\langle \boldsymbol{N}, \boldsymbol{H}\rangle \langle \boldsymbol{N}, \boldsymbol{L}\rangle}{\langle \boldsymbol{V}, \boldsymbol{H}\rangle} \tag{2.6.5}$$

Anche al denominatore sembrerebbe necessario sostituire $\boldsymbol{V}$ con $\boldsymbol{L}$, ma è pleonastico perché, per definizione di *halfway vector*, si ha $\langle \boldsymbol{V}, \boldsymbol{H}\rangle = \langle \boldsymbol{L}, \boldsymbol{H}\rangle$. Come valore di G. per semplicità di calcolo, possiamo prendere il valore minimo tra i tre valori così calcolati. Sarebbe assai più preciso, ma anche più complicato, calcolare la probabilità dei tre eventi (a), (b) e (c) e scegliere G come la media pesata dei tre fattori di attenuazione geometrica con i pesi dati da queste probabilità.

2.6.4. Le equazioni di Fresnel per la riflessione della luce. Ora introduciamo nel modello di illuminazione di Torrance–Sparrow la legge fisica che regola la distribuzione in frequenza della luce riflessa: la legge di Fresnel.
L'equazione di Fresnel per la luce non polarizzata determina la frazione di luce riflessa da una superficie dielettrica (ossia non metallica):

$$F_{\lambda\theta_i} = \frac{\tan^2(\theta_i - \theta_t)}{\tan^2(\theta_i + \theta_t)} = \frac{1}{2}\,\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)}\left(1 + \frac{\cos^2(\theta_i - \theta_t)}{\cos^2(\theta_i + \theta_t)}\right). \tag{2.6.6}$$

Qui θ_i l'angolo fra $\boldsymbol{L}$ e $\boldsymbol{N}$, e θ_t l'angolo di rifrazione; dalla legge di Snell per la rifrazione sappiamo che $\sin\theta_t/\sin\theta_i = \eta_{i\lambda}/\eta_{t\lambda}$, dove $\eta_{i\lambda}$ e $\eta_{t\lambda}$ sono gli indici di rifrazione dei due materiali, superficie riflettente ed ambiente circostante.
L'equazione di Fresnel può anche essere espressa come

$$F_\lambda = \frac{1}{2}\,\frac{(g-c)^2}{(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g+c)+1)^2}\right) \tag{2.6.7}$$

dove $c = \cos\theta_i$, $g^2 = \eta_\lambda^2 + c^2 - 1$ e $\eta_\lambda = \eta_{t\lambda}/\eta_{i\lambda}$.

Invece in un materiale conduttore, che attenua la luce, perché l'equazione (2.6.7) continui a valere è necessario sostituire l'indice di rifrazione del materiale con un nuovo indice di rifrazione a valori complessi $\breve{\eta}_\lambda$, definito così:

$$\breve{\eta}_\lambda = \eta_\lambda - ik_\lambda \tag{2.6.8}$$

(qui k_λ misura la frazione di luce assorbita per unità di lunghezza nell'attraversare il materiale rifrangente).
La parte immaginaria k_λ è il coefficiente di attenuazione della luce nel materiale: esso quindi misura la percentuale di attenuazione dell'intensità della luce a frequenza λ per unità di distanza attraversata. Per semplificare i calcoli di riflessione e rifrazione, nei materiali non dielettrici k_λ si assume sia zero. Questo elimina la componente immaginaria, e lascia un singolo valore η_λ reale. Per maggiori dettagli sull'equazione di Fresnel per materiali conduttori se veda la Sottosezione 7.7.8 in seguito.

2.6.5. Confronto fra i modelli di Torrance–Sparrow e di Phong. Per apprezzare la differenza del rendering fra i modelli di illuminazione di Torrance–Sparrow e di Phong è opportuno esaminare il caso di luce radente: in questo caso, in (2.6.7) abbiamo $\theta_i \approx \pi/2$, quindi $c \approx 0$, e quindi $F_\lambda = 1$ per tutte le frequenze λ. Uno sviluppo di Taylor rispetto a c mostra che la distribuzione dell'illuminazione riflessa ad angoli diversi da quello esattamente speculare decade molto più rapidamente con la deviazione laterale rispetto a tale direzione di quanto non avvenga se si usa il modello di Phong. Infatti, nel modello di Phong dobbiamo calcolare l'angolo fra il versore riflesso $\boldsymbol{R}$ e il versore del punto di osservazione $\boldsymbol{V}$. Ma se $\arccos(\langle \boldsymbol{L}, \boldsymbol{H}\rangle \approx \frac{\pi}{2}$, allora $\boldsymbol{L}$ è quasi perpendicolare all'*halfway vector* $\boldsymbol{H}$, e quindi $\boldsymbol{L}$ e $\boldsymbol{V}$ formano un angolo di circa π (poiché $0 = \langle \boldsymbol{H}, \boldsymbol{L}\rangle = 1/2(\langle \boldsymbol{L}, \boldsymbol{L}\rangle + \langle \boldsymbol{V}, \boldsymbol{L}\rangle) = 1/2(1 + \langle \boldsymbol{V}, \boldsymbol{L}\rangle)$ da cui $\langle \boldsymbol{V}, \boldsymbol{L}\rangle = -1$). In tal caso $\boldsymbol{V}$ e $\boldsymbol{R}$ sono quasi allineati, quindi il coseno $\langle \boldsymbol{V}, \boldsymbol{R}\rangle$ vale circa $\cos(\theta_i - \frac{\pi}{2}) \approx 1$. Allora lo sviluppo di Taylor rispetto a $c = \cos\theta_i$ del termine di Phong $\langle \boldsymbol{V} \cdot \boldsymbol{R}\rangle^n$ è dell'ordine di

$$\cos^n\left(\theta_i - \frac{\pi}{2}\right) \approx \left(1 - \frac{(\theta_i - \pi/2)^2}{2}\right) \approx 1 - \frac{n(\theta_i - \pi/2)^2}{2}\,, \tag{2.6.9}$$

che vale pur sempre 1 se $\theta_i = \frac{\pi}{2}$, ma decade quadraticamente al deviare da questo valore di θ_i (e quindi con lo stesso ordine ma comunque più velocemente del coseno, perché il primo termine che abbiamo trascurato nello sviluppo di Taylor in (2.6.9) ha segno positivo). Questo più elevato decadimento laterale è ben visibile nelle seguenti immagini tratte dal succitato articolo di Blinn, per angoli di incidenza (e quindi di riflessione) di 30^0 (Figura 2.6.3) e di 70^0 (Figura 2.6.4). In esse l'illuminazione è data da un fascio di raggi paralleli e si assume che il colore della luce speculare dipenda solo da quello della luce incidente (quindi si trascura una delle conseguenze del termine di Fresnel). Quelle che appaiono come superficie solide descrivono in realtà i diagrammi polari tridimensionali dell'intensità di luce riflessa alle varie direzioni (superficie più elongate rappresentano quindi distribuzioni angolari più concentrate). L'effetto che si ottiene è una maggiore concentrazione della luce riflessa quando la luce incidente è radente, come mostra la Figura 2.6.5.

2.6.6. La variazione del colore nella riflessione speculare. Abbiamo accennato che nell'articolo di Blinn si fa l'ipotesi che la lunghezza d'onda (ovvero il colore) della luce riflessa sia identica a quella della luce incidente. Ma l'equazione di Fresnel prevede che la luce riflessa

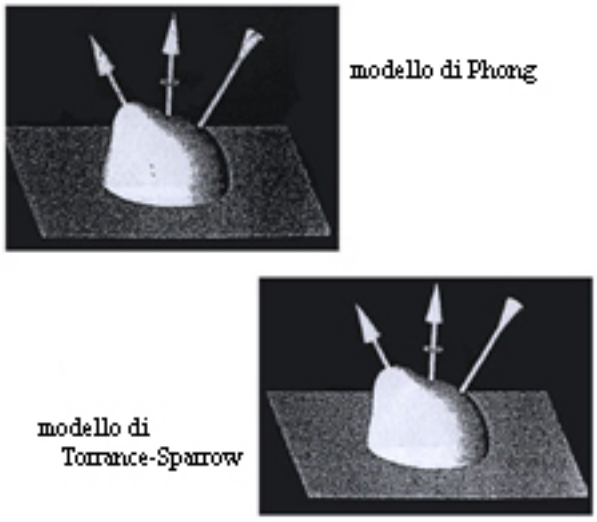


FIGURA 2.6.3. Paragone fra la distribuzione angolare della riflessione della luce incidente a 30^0 per i modelli di Phong e di Torrance–Sparrow

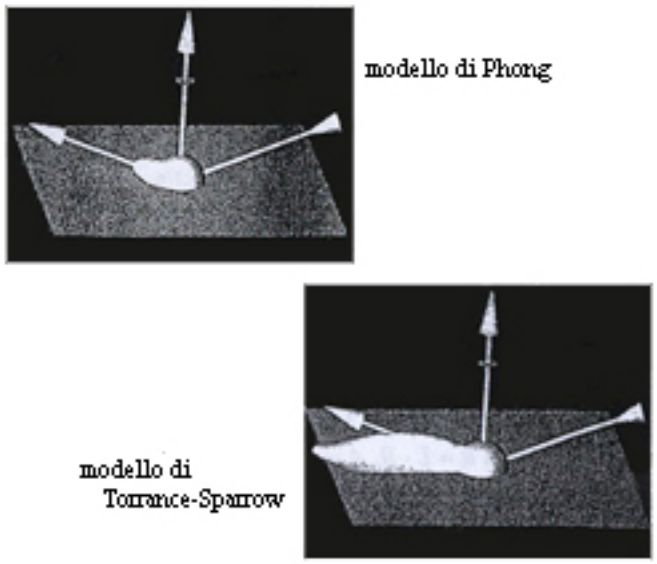


FIGURA 2.6.4. Paragone fra la distribuzione angolare della riflessione della luce incidente a 70^0 per i modelli di Phong e di Torrance–Sparrow

abbia distribuzione spettrale con una variazione continua della lunghezza d'onda. Pertanto un modello fisico più preciso deve tener conto di tale distribuzione, che dipende dai tre fattori seguenti (si riveda l'uguaglianza (2.6.6):

- la natura dei materiali (tramite i loro indici di rifrazione complessi);
- la lunghezza d'onda della luce incidente;
- l'angolo d'incidenza.

In funzione di tali parametri, si ha uno shift in frequenza nella luce riflessa rispetto a quella

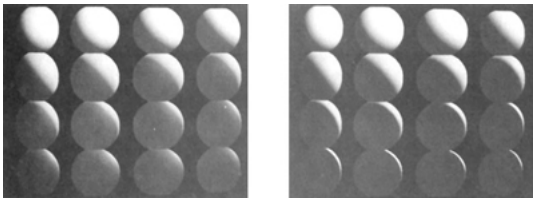


FIGURA 2.6.5. Figura 5. Paragone fra l'illuminazione di una sfera metallica per i modelli di Phong e di Torrance–Sparrow al variare della direzione della luce incidente

incidente: ovvero, la la luce riflessa viene *colorata* dal materiale su cui si riflette. Questa conseguenza della espressione (2.6.6) è illustrata nella Figura 2.6.6, che mostra l'andamento del termine di Fresnel in funzione della lunghezza d'onda e dell'angolo di incidenza per la riflessione su rame: il termine di Fresnel vale 1 indipendentemente da λ per angolo di incidenza zero (luce ed osservatore disposti in modo radente, l'illuminazione è del colore della sorgente), ma ne dipende per piccoli angoli di incidenza (in tal caso l'illuminazione è sbilanciata verso le lunghezze d'onda maggiori, ossia verso il rosso, il colore del rame).

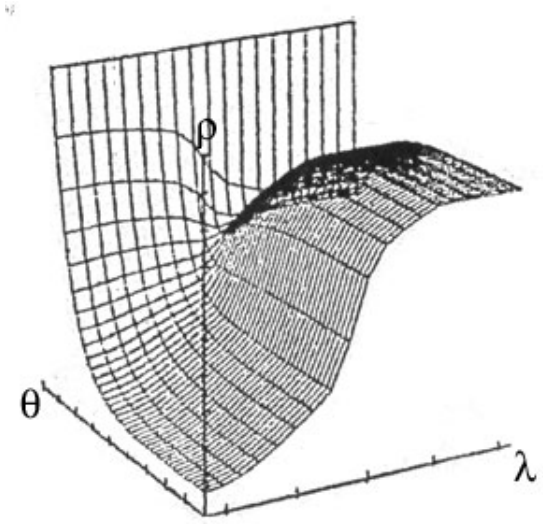


FIGURA 2.6.6. Valore del termine di Fresnel per uno specchio di rame, in funzione dell'angolo di incidenza θ rispetto allo *halfway vector* e della lunghezza d'onda λ. Per $\theta = \pi/2$ (luce radente) il termine di Fresnel vale costantemente 1 per ogni λ e la luce che arriva all'osservatore ha il colore della sorgente; per $\theta = 0$ (riflessione frontale), il termine di Fresnel è piccolo per lunghezze d'onda piccole (all'estremità blu-violetta dello spettro), e la luce riflessa ha una tinta rossastra

La colorazione della luce riflessa è stata introdotta in questo modello fisico di illuminazione nel succitato articolo di Cook e Torrance, i quali osservano i seguenti fatti.

Quando la luce incidente è perpendicolare alla microsfaccettaura, e quindi nella direzione di $\boldsymbol{H}$, allora $\theta_i = 0$, quindi $c = 1$ e $g = \eta_\lambda$. Sostituendo tali valori nell'equazione di Fresnel

(2.6.7), otteniamo il termine di Fresnel per $\theta_i = 0$:

$$F_{\lambda,0} = \left(\frac{\eta_\lambda - 1}{\eta_\lambda + 1}\right)^2 . \tag{2.6.10}$$

Invece, come abbiamo appena osservato alla fine della precedente Sezione 2.6.4, quando la luce incidente è radente allora $\theta_i = \pi/2$, quindi $c = 0$, ed il termine di Fresnel per $\theta_i = \pi/2$ è

$$F_{\lambda,\pi/2} = 1 . \tag{2.6.11}$$

Quindi, se la luce è normale alla microsfaccettatura, allora F_{λ_0}, e di conseguenza il coefficiente di riflettività speculare ρ_s, sono funzioni dell'indice di rifrazione della superficie, che a sua volta varia con la lunghezza d'onda della luce incidente.

Rivediamo invece il caso di luce radente. Come sempre, dobbiamo limitare l'attenzione ad una posizione di osservazione situata di fronte alla posizione della luce, cioè ad un angolo $\pi = 180^0$ rispetto a $\boldsymbol{L}$ (si tratta della configurazione nella quale i versori $\boldsymbol{L}$ della luce e $\boldsymbol{V}$ della direzione di osservazione sono entrambi tangenziali alla superficie ed opposti di segno, e quindi lo *halfway vector* $\boldsymbol{H} = 1/2(\boldsymbol{L}+\boldsymbol{V})$ coincide con il versore normale N, come già notato nella derivazione della identità (2.6.9)).
Allora $F_{\lambda_{\frac{\pi}{2}}}$, e di conseguenza ρ_s, valgono 1: quindi ρ_s non dipende da η_λ. In altre parole il colore della luce riflessa in questo caso è lo stesso della luce incidente e non dipende dalla natura fisica del materiale su di cui avviene la riflessione.
Però per ogni angolo di incidenza diverso da $\pi/2$ la riflettività speculare dipende dal coefficiente di rifrazione η_λ e quindi ha una distribuzione in frequenza, cioè un colore, diverso da quello della luce incidente
Per le superficie metalliche, sostanzialmente ogni riflessione avviene sulla superficie ed è speculare. Solo per angoli molto vicini a $\frac{\pi}{2}$ la riflettività speculare non viene influenzata dal materiale su di cui l'oggetto si riflette .
Si noti come in questo caso il coefficiente di riflessione speculare differisca da quello di Phong che è sempre indipendente dal colore dell'oggetto. Cook e Torrance osservano che il modello di Phong funziona bene per i materiali plastici, che difatti sono costituiti da pigmenti immersi in un colloide trasparente. Il colore della plastica è determinato dalla luce che penetra nel colloide trasparente ed interagisce con i pigmenti. Ad esempio, se la luce è bianca ed i pigmenti sono rossi, allora la plastica appare rossa. Invece la luce riflessa si riflette sulla superfice del colloide e non vi penetra dentro: perciò non raggiunge i pigmenti. Pertanto Il colore della luce riflessa non dipende dal colore del materiale (plastica rossa, verde od altro). In altre parole la luce riflessa non viene colorata dalla plastica, e quindi in modello di Phong, che è indipendente dal colore del materiale, in questo caso dà una resa fedele. D'altra parte questo vuol dire che gli oggetti resi con il modello di illuminazione di Phong sembrano tutti di plastica. Si veda, ad esempio, la Figura 5 , che confronta, a diversi angoli di incidenza della luce, il rendering di sfere metalliche nei modelli di illuminazione di Phong e di Torrance–Sparrow: nel caso di luce radente, il secondo modello rende molto meglio la tipica riflessione caratteristica dei metalli.

Se si conosce l'indice di rifrazione per diversi valori della lunghezza d'onda, allora lo si può direttamente utilizzare nell'equazione di Fresnel. Di solito non lo si conosce, ma il coefficiente di riflettività al variare della lunghezza d'onda è stato misurato per molti materiali nel caso incidenza perpendicolare ($\theta_i = 0$). In tal caso si puó ricavare il coefficiente di rifrazione η_λ

dall'equazione di Fresnel (2.6.7):

$$\eta_\lambda = \frac{1+\sqrt{F_{\lambda,0}}}{1-\sqrt{F_{\lambda,0}}} \tag{2.6.12}$$

Questo valore η_λ può allora essere utilizzato nell'equazione di Fresnel per ottenere $F_{\lambda\theta_i}$ per qualunque altro angolo di incidenza θ_i. Poiché questo calcolo è oneroso, Cook e Torrance lo semplificano calcolando un valor medio $\widetilde{F}_{\lambda\theta}$ del termine di Fresnel, corrispondente ad un valor medio $\widetilde{\eta}_\lambda$ dell'indice di rifrazione corrispondente alla media della riflettività perpendicolare. Il valore così calcolato viene utilizzato per interpolare tra il colore del materiale a $\theta_i = \pi/2$ ed a $\theta_i = 0$ per le frequenze corrispondenti per ciascuna delle primarie del modello di colore. A $\pi/2$ come si è visto $F_{\lambda_{\pi/2}} = 1$ e dunque il colore del materiale è quello della sorgente di luce. Pertanto il colore della luce riflessa a quest'angolo di incidenza è uguale a quello della luce incidente: utilizziamo il modello RGB, consideriamo, ad esempio, la componente rossa di tale colore e la chiamiamo $\text{Rosso}_{\pi/2}$. Chiamiamo poi Rosso_0 il valore della componente rossa del colore del materiale, ossia della luce riflessa frontalmente ($\theta_i = 0$). La componente Rosso_0 si ottiene nel modo seguente. Si calcolano prima le coordinate CIE della luce incidente. Se la luce incidente arriva frontalmente (cioè ad angolo 0), la luce riflessa che vediamo ha distribuzione spettrale (vale a dire in funzione della lunghezza d'onda) $F_{\lambda 0}P(\lambda)$, dove $P(\lambda)$ è la distribuzione spettrale della luce incidente. Allora le coordinate CIE si ottengono come nella Sottosezione 17.9 dell'Appendice 17:

$$X_0 = c\int_\infty^\infty F_{\lambda 0}\, P(\lambda)\, \overline{x}(\lambda)\, d\lambda$$

$$Y_0 = c\int_\infty^\infty F_{\lambda 0}\, P(\lambda)\, \overline{y}(\lambda)\, d\lambda$$

$$Z_0 = c\int_\infty^\infty F_{\lambda 0}\, P(\lambda)\, \overline{z}(\lambda)\, d\lambda$$

Qui $\overline{x}$, $\overline{y}$ e $\overline{z}$ sono le curve di corrispondenza ("color matching") del modello CIE, e c è la costante di normalizzazione che determina la scala in modo che la luce bianca più intensa corrisponda al 100÷ di intensità. Dalle coordinate CIE si risale alla componente rossa grazie alla regola di trasformazione CIE-RGB data dalla matrice J nella Sottosezione 17.16 dell'Appendice 17.

Più precisamente

$$\begin{pmatrix} \text{Rosso}_0 \\ \text{Verde}_0 \\ \text{Blu}_0 \end{pmatrix} = J^{-1} \begin{pmatrix} \overline{X}_0 \\ \overline{Y}_0 \\ \overline{Z}_0 \end{pmatrix}$$

Invece, come abbiamo osservato ripetutamente, a $\theta_i = \pi/2$ la luce riflessa è radente, $F_{\lambda_{\pi/2}}$=1, ed il colore della luce riflessa è lo stesso di quella incidente, ovvero la sua distribuzione spettrale rimane invariata: $F_{\lambda_{\pi/2}}P(\lambda) = P(\lambda)$

Perciò le coordinate CIE sono :

$$X_{\pi/2} = c\int_\infty^\infty P(\lambda)\, \overline{x}(\lambda)\, d\lambda$$

$$Y_{\pi/2} = c\int_\infty^\infty P(\lambda)\, \overline{y}(\lambda)\, d\lambda$$

$$Z_{\pi/2} = c \int_{\infty}^{\infty} P(\lambda)\, \overline{z}(\lambda)\, d\lambda$$

ed abbiamo

$$\begin{pmatrix} \text{Rosso}_{\pi/2} \\ \text{Verde}_{\pi/2} \\ \text{Blu}_{\pi/2} \end{pmatrix} = J^{-1} \begin{pmatrix} X_{\pi/2} \\ Y_{\pi/2} \\ Z_{\pi/2} \end{pmatrix}$$

Ora possiamo interpolare tra $\theta_i = 0$ e $\theta_i = \pi/2$. In questo modo si ottiene

$$\text{Rosso}_{\theta_i} = \text{Rosso}_0 + (\text{Rosso}_{\pi/2} + \text{Rosso}_0) \frac{\max(0, \widetilde{F}_{\theta_i} - \widetilde{F}_0)}{\widetilde{F}_{\theta_i} - \widetilde{F}_0} .$$

Ora sostituiamo $F_\lambda = F_{\lambda\theta_i}$ con Rosso_{θ_i} nell'espressione di Cook e Torrance (2.6.1)per la componente speculare del coefficiente di riflettività bidirezionale. Poiché questa approssimazione si basa su una media rispetto alla lunghezza d'onda, nell'equazione dell'illuminazione il termine speculare deve essere ora moltiplicato per un fattore di scala indipendente dalla lunghezza d'onda, invece che dalla distribuzione spettrale dell'intensità luminosa.

Tipicamente, sia per i materiali conduttori sia per i dielettrici, le componenti ambientale, diffusa e speculare sono del colore dell'oggetto. Per i materiali compositi, come la plastica, abbiamo già visto che le componenti diffusa e speculare sono di colore diverso. I metalli hanno una componente diffusa irrilevante, mentre quella speculare è del colore del metallo per angolo di incidenza 0^0 (luce frontale) e del colore della luce incidente per illuminazione radente (angolo di incidenza di 90^0 rispetto alla normale).

Polarizzazione della luce: l'equazione di Fresnel risulta valida solo per la luce non polarizzata: ma lo stato di polarizzazione della luce varia quando la luce viene riflessa da una superficie, ed il coefficiente di riflettività di una superficie varia in funzione allo stato di polarizzazione della luce incidente su di essa. Wolff e Kustlander hanno esteso il modello di Cook–Torrance per includere la luce polarizzata (la differenza diventa più evidente dopo due o più inter-riflessioni fra oggetti). In primo luogo consideriamo i materiali dielettrici: per ogni dielettrico esiste un angolo, detto angolo di Brewster, al quale la luce incidente viene completamente polarizzata quando riflessa o non viene affatto riflessa se la sua polarizzazione non è appropriata. Se la riflessione interoggetto di luce inizialmente non polarizzata avviene tra due superficie dielettriche e l'angolo di incidenza di ciascuna riflessione è pari all'angolo di Brewster ed il piano che contiene $\boldsymbol{N}$ ed $\boldsymbol{L}$ di una superficie è perpendicolare a quello dell'altra, allora la luce non viene riflessa specularmente dal secondo oggetto; variando angoli ed orientamenti otteniamo ancora un effetto di attenuazione, ma meno accentuato. In secondo luogo consideriamo i conduttori colorati (metalli come rame o oro, ad esempio): essi tendono a polarizzare la luce in modo diverso a differenti lunghezze d'onda, quindi se un conduttore colorato si riflette su una superficie dielettrica, la riflessione ha colore leggermente diverso rispetto a quello che si ottiene se non si considera la polarizzazione.

CAPITOLO 3

Ombreggiatura interpolata

Per rendere immagini realistiche non basta illuminare ciascuna superficie o poligono della scena con un colore fisso. Il colore deve cambiare da punto a punto, generando un'ombreggiatura (shading). La variazione di colore e luminosità è conseguenza del fatto che, quando cambiamo il punto osservato, cambia la geometria della scena, e piĂč precisamente le direzioni da questo punto alle luci ed al punto di osservazione. Solo nel caso in cui sia le luci sia l'osservatore sono collocati a distanza infinita i corrispondenti versori rimangono invariati, ed in tal caso si ottiene una colorazione fissa di ogni superficie o poligono, quindi un rendering poco realistico. In generale, invece, dovremmo ricalcolare la geometria punto per punto.

Per accelerare il rendering si usano metodi di ombreggiatura interpolata. I metodi di interpolazione richiedono solo informazioni ai vertici dei poligoni con cui si approssimano le superficie della scena: è solo necessario conoscere di partenza il versore normale alla superficie in ogni vertice della maglia poligonale. Questi dati si possono calcolare in maniera analitica se si conosce la descrizione analitica della superficie (ad esempio l'equazione che la determina); altrimenti essi vengono calcolati in maniera approssimata considerando, per ogni vertice, i versori normali dei poligoni che si toccano a quel vertice (i quali sono forniti dal modellatore oppure calcolati grazie al prodotto vettoriale dei vettori che formano due suoi lati consecutivi), poi calcolando la media aritmetica di tali versori (che di solito non ha piĂč norma 1), ed infine normalizzando. Ci sono due metodi standard che permettono un eccellente compromesso fra velocità ed accuratezza:

- Ombreggiatura interpolata di Gouraud (Sottosezione 3.1)
- Ombreggiatura interpolata di Phong (Sottosezione 3.2)

3.1. Il metodo di interpolazione di Gouraud

Il metodo di ombreggiatura interpolata di Gouraud **[20]** è il metodo più semplice e rapido per interpolare l'ombreggiatura. Esso interpola solo dati scalari, non vettoriali, e precisamente le intensità di illuminazione ad ogni vertice, calcolate grazie all'equazione dell'illuminazione del modello di Phong, a partire dalla direzione della sorgente (o delle sorgenti) di luce rispetto a quel vertice e dal suo versore normale (calcolato tramite operazioni di media sui poligoni che condividono quel vertice). In questo modo l'interpolazione all'interno di un poligono può essere calcolata in maniera incrementale al percorrere successivamente le varie righe di scansione della proiezione sul piano di visuale (scan conversion) del poligono, esattamente come per il calcolo della profondità nello z-buffer. In tal modo, se si vuole, il calcolo può essere eseguito nel piano (coordinate dell'osservatore), ed allora non richiede l'aggravio di lavoro dei calcoli nello spazio tridimensionale (coordinate dello spazio ambiente): in questo caso, però, esso diventa a precisione di immagine invece che di oggetto.

In ogni sua variante, questo metodo ottimizza la velocità e l'efficienza. Tuttavia, esso non produce sempre un rendering accurato, soprattutto in presenza di superficie molto speculari.

Supponiamo ad esempio che la scena contenga un grande poligono perfettamente speculare (o quasi), abbastanza vicino all'osservatore affinché ai suoi vertici le normali siano quasi perpendicolari alla direzione di osservazione. Allora i vertici sono al buio, ma all'interno del grande poligono ci può essere un punto in cui il versore normale è piĂč favorevole, e cioè a metà strada fra la direzione della luce e quello dell'osservatore. In tal caso in quel punto dovremmo vedere la riflessione speculare della sorgente di luce: ma il metodo di Gouraud interpola fra le luminosità dei vertici, e quindi restituisce bassa luminosità. Ad esempio, nel rendering tramite interpolazione di Gouraud a partire dai vertici del triangolo illuminato al centro della Figura 3.1.1, il triangolo apparirebbe nero.

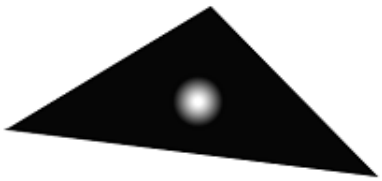


FIGURA 3.1.1. Questo triangolo, che assumiamo puramnete diffusivo, è illuminato da una sorgente di luce vicino al suo baricentro, e l'osservatore è vicino alla sorgente di luce. Quindi i versori della luce e dell'osservatore sono quasi allineati al versore normale al centro del triangolo, ma quasi perpendicolari ai tre vertici. Quindi i tre vertici, in base all'equazione di Lambert, sono scuri. Il metodo di Gouraud produrrebbe un rendering di questo triangolo completamento scuro ovunque, ed ometterebbe quindi gli effetti della sorgente di luce. Lo stesso accadrebbe per un triangolo riflettente, in base all'equazione di illuminazione di Phong

Per evitare questo errore dobbiamo interpolare non i valori della luminosità (scalari), bensì quelli dei versori normali (vettoriali), e poi rieseguire il calcolo dell'illuminazione ad ogni pixel della scan-conversion del poligono. Questo è precisamente ciò che si fa in un metodo di interpolazione piĂč accurato, il metodo di ombreggiatura di Phong.

3.2. Il metodo di interpolazione di Phong

A differenza del metodo di Gouraud, il metodo di ombreggiatura interpolata di Phong Bui-Tong **[34]** interpola non i valori dell'illuminazione (scalari), bensì i versori normali e della direzione della sorgente di luce (vettoriali), e quindi è piĂč accurato. La seguente figura illustra l'interpolazione, a partire dai vertici della modellazione poligonale, dei versori normale, della direzione della luce e della direzione dell'osservatore (Figura 3.2.2). In particolare, questo metodo di interpolazione si presta anche al rendering di superficie speculari, senza dar luogo alle patologie illustrate nella spiegazione del metodo di Gouraud. Per ogni pixel della scan conversion di un poligono della scena, il metodo di Phong applica l'equazione dell'illuminazione del modello di Phong, a partire dai vertici del poligono, dove vengono calcolati il versore di direzione della sorgente di luce, il versore normale alla superficie, ottenuto tramite operazioni di media sui poligoni che condividono quel vertice, ed il versore della direzione dell'osservatore.
In questo modo l'interpolazione all'interno di un poligono può essere calcolata in maniera incrementale al percorrere successivamente le varie righe di scansione della proiezione sul piano

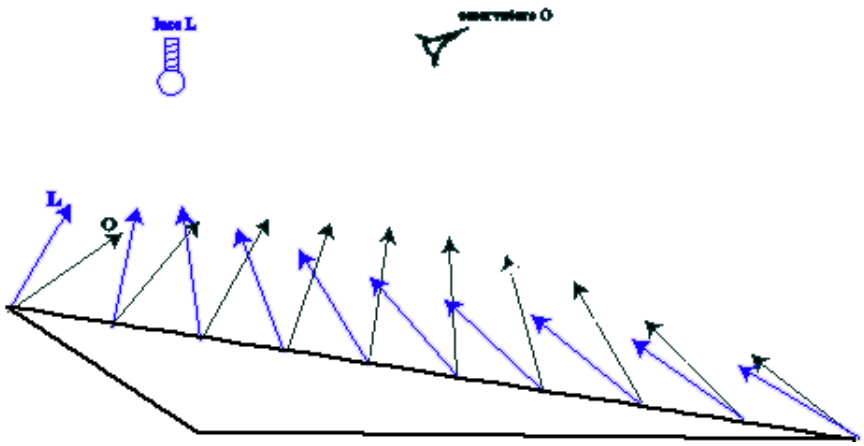


FIGURA 3.2.1. Interpolazione di Phong dei versori della luce e dell'osservatore, ed anche del versore normale, lungo un lato di un triangolo. La stessa procedura viene applicata agli altri due lati, e prolungata all'interno tramite intepolazione bilineare o bicubica

di visuale (scan conversion) del poligono, esattamente come per il calcolo della profondità nello z-buffer: però ora si devono interpolare tre vettori, e quindi le loro tre componenti, per un totale di nove interpolazioni: il metodo è quindi almeno nove volte piĂč lento di quello di Gouraud (inoltre le operazioni aritmetiche sono piĂč onerose, in quanto richiedono anche il calcolo di prodotti scalari, e non soltanto moltiplicazioni di numeri). In tal modo, come nel metodo di Gouraud, il calcolo può essere eseguito nel piano (ossia nelle coordinate dell'osservatore): però esso diventa a precisione di immagine invece che di oggetto.

3.2.1. Esercizio sul metodo di interpolazione di Phong ed i riflettori di Warn.

ESERCIZIO 3.2.1. Consideriamo un triangolo ombreggiato con il metodo di Phong. I vertici del triangolo sono $\boldsymbol{e_1} = (1,0,0)$, $\boldsymbol{e_2} = (0,1,0)$, $\boldsymbol{e_3} = (0,0,1)$. La sorgente (puntiforme) di luce di intensità 1 (a ciascuna frequenza monocromatica) è nel punto $(0,0,10)$, ma èdiretta verso un riflettore di Warn (Sezione **??**) che giace sul piano di equazione $z = 11$: quindi la sorgente di luce per la scena è il riflettore di Warn. Si suppongano nulli i termini di illuminazione ambientale e di Lambert, e si pongano uguale a 1 l'esponente di Phong del riflettore e del triangolo. L'osservatore è nel punto $\boldsymbol{o} = (5,1,0)$. Le maglie adiacenti al triangolo sono disposte in maniera tale che i vettori normali alla superficie (non normalizzati) di cui il triangolo fa parte (calcolati tramite medie tra i vettori perpendicolari a questo triangolo ed a quelli delle maglie adiacenti) siano rispettivamente $\boldsymbol{n_1} = (1,1,0)$, $\boldsymbol{n_2} = (1,0,1)$, $\boldsymbol{n_3} = (0,1,1)$. Quale è l'intensità di illuminazione che l'osservatore vede al baricentro del triangolo (cioè al punto ottenuto dalla combinazione convessa dei tre vertici con pesi uguali)?

Svolgimento.

Secondo il modello di ombreggiature di Warn l'intensità luminosa è data da :

$$I_\lambda \langle \boldsymbol{L}, -\boldsymbol{L'} \rangle^p = I_\lambda \cos^p \gamma,$$

dove I_λ è la intensità della sorgente che illumina il riflettore di Warn (qui posta ugiuale a 1 per ogni frequenza λ), $\boldsymbol{L}$ è il raggio dal punto osservato al riflettore, $\boldsymbol{L'}$ è il raggio normale al riflettore e passante per la sorgente di luce, γ è l'angolo compreso tra questi due angoli e p è

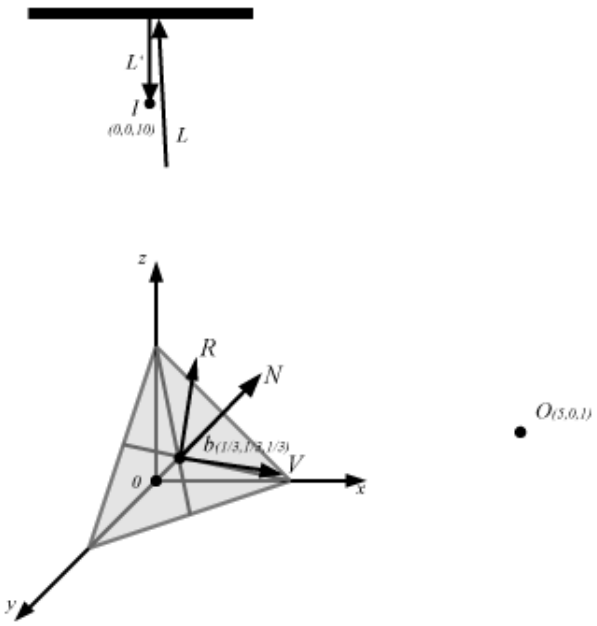


FIGURA 3.2.2. La posizione del triangolo e del riflettore di Warn nella scena dell'Esercizio 3.2.1

l'esponente di Phong.
Il baricentro del triangolo è il punto $\boldsymbol{b} = 1/3\boldsymbol{e_1} + 1/3\boldsymbol{e_2} + 1/3\boldsymbol{e_3} = (1/3, 1/3, 1/3)$.
Il versore $\boldsymbol{L'}$ dal riflettore alla sorgente di luce è $\boldsymbol{L'} = (0, 0, -1)$.
Calcoliamo il raggio $\boldsymbol{L}$ dal punto osservato $\boldsymbol{b}$ al riflettore:

$$\boldsymbol{L} = \frac{(0,0,11) - (\frac{1}{3},\frac{1}{3},\frac{1}{3})}{\sqrt{(-\frac{1}{3})^2 + (-\frac{1}{3})^2 + (\frac{32}{3})^2}} = \left(\frac{-1}{\sqrt{1026}}, \frac{-1}{\sqrt{1026}}, \frac{32}{\sqrt{1026}}\right).$$

Il versore $\boldsymbol{V}$ dal baricentro $\boldsymbol{b}$ all'osservatore in $\boldsymbol{o}$ è:

$$\boldsymbol{V} = \frac{(5,0,1) - (\frac{1}{3},\frac{1}{3},\frac{1}{3})}{\frac{1}{3}\sqrt{201}} = \left(\frac{14}{\sqrt{201}}, \frac{-1}{\sqrt{201}}, \frac{2}{\sqrt{201}}\right).$$

La normale al triangolo sul punto $\boldsymbol{b}$ col metodo di illuminazione di Phong viene calcolata per interpolazione con i vettori normali alla superficie di cui il triangolo fa parte: $\boldsymbol{n_1}$, $\boldsymbol{n_2}$ e $\boldsymbol{n_3}$. Otteniamo quindi:

$$\boldsymbol{N} = \frac{\boldsymbol{n_1} + \boldsymbol{n_2} + \boldsymbol{n_3}}{\|\boldsymbol{n_1} + \boldsymbol{n_2} + \boldsymbol{n_3}\|} = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right).$$

Adesso possiamo calcolare il versore $\boldsymbol{R}$ del raggio riflesso:

$$\boldsymbol{R} = 2\langle \boldsymbol{N}, \boldsymbol{L}\rangle \boldsymbol{N} - \boldsymbol{L} = \frac{20}{\sqrt{1026}}(1,1,1) - \left(\frac{-1}{\sqrt{1026}}, \frac{-1}{\sqrt{1026}}, \frac{32}{\sqrt{1026}}\right)$$
$$= \left(\frac{21}{\sqrt{1026}}, \frac{21}{\sqrt{1026}}, \frac{-12}{\sqrt{1026}}\right).$$

Dal momento che i termini di illuminazione ambientale e di Lambert sono nulli ed il coefficiente di Phong p vale 1, l'intensità luminosa finale risulta ora essere

$$\langle \boldsymbol{L}, -\boldsymbol{L}' \rangle \langle \boldsymbol{R}, \boldsymbol{V} \rangle = \frac{32}{\sqrt{1026}} \frac{249}{\sqrt{1026}\sqrt{201}} = \frac{7.76608187134503}{\sqrt{201}} .$$

□

3.3. Artefatti causati dagli algoritmi di ombreggiatura interpolata

Gli algoritmi di ombreggiatura interpolata non danno luogo sempre ad immagini realistiche: essi infatti introducono errori visibili. Ecco un elenco.

Profili poligonali. Uno di questi errori è il fatto che, interpolando a partire da poligoni, si ottengono sì gradazioni ben sfumate di ombreggiatura atte a simulare la rotondità delle superficie, ma i bordi esterni di ciascun oggetto rimangono poligonali, cioè con un profilo fatto di segmenti rettilinei. Questo problema è ineliminabile, ma il suo impatto visivo si può ridurre se si aumenta la risoluzione del modellatore, cioè se si prende una maglia poligonale piĂč fitta. In questo modo però aumenta la mole di calcoli.

Distorsione prospettica della profondità. Un altro errore di interpolazione è la distorsione prospettica della profondità. Immaginiamo di voler rendere un pavimento fatto di mattonelle quadrate, il cui colore sfuma dal bianco al nero al crescere della distanza dall'osservatore. Supponiamo che il pavimento. La linee che separano una riga di mattonelle dalla successiva si dispongono, nelle coordinate del monitor, in maniera piĂč fitta man mano che ci si avvicina al punto di fuga prospettico, a causa della divisione prospettica responsabile dello schiacciamento con la profondità. Supponiamo che il pavimento costituisca un unico lungo poligono della modellazione, che comincia a distanza di 100 metri e finisce a distanza di 1000 metri, in modo che l'ultima riga di mattonelle si proietti sul piano di visuale in punti molto vicini al punto di fuga (cioè che la sua proiezione sia un segmento orizzontale molto corto. La riga di metà profondità, a 450 metri, appare nella proiezione non a metà altezza, ma assai piĂč vicina alla riga piĂč alta, di nuovo a causa dell'infittirsi delle righe con la profondità. Ma l'algoritmo di ombreggiatura interpola il colore linearmente rispetto alle coordinate del piano di visuale, dalla riga piĂč alta a quella piĂč bassa, e quindi colloca il colore intermedio, il grigio medio di intensità 50%, esattamente sulla riga che sull'immagine proiettata appare a mezza altezza. L'errore è dovuto al fatto che nelle coordinate del piano di visuale la proiezione prospettica, a causa della proiezione prospettica, non è lineare rispetto alla distanza, mentre invece l'interpolazione lo è. La Figura 3.3.1 mostra, a sinistra, il pavimento reso per interpolazione a partire dai quattro vertici, ed a destra lo stesso pavimento con il colore che cambia proporzionalmente alla profondità nello spazio tridimensionale. Il segmento orizzontale interno rappresenta la linea del grigio intermedio (50% di luminosità).

Per ridurre questo tipo di distorsione occorre frazionare i poligoni (in questo caso il rettangolo del pavimento) in tanti poligoni piĂč piccoli (od almeno piĂč corti nella direzione della profondità).

Dipendenza dall'orientamento. Un altro errore tipico è la dipendenza dell'interpolazione dall'orientamento dei poligoni. Quando si ruota un poligono, un punto al suo interno, dopo la rotazione, può essere interpolato a partire da due lati diversi rispetto a prima, come nella Figura seguente:

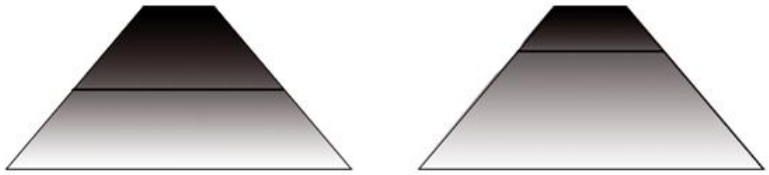


FIGURA 3.3.1. Distorsione prospettica dovuta all'interpolazione. Il rettangolo visto in prospettiva ha una tessitura che rappresenta una sfumatura lineare di toni di grigio. Nell'interpolazione, la linea di grigio medio viene disegnata al centro del rettangolo (disegno a sinistra), ma nella realtà, a causa della divisione propsettica, viene vista più in alto (disegno a destra)

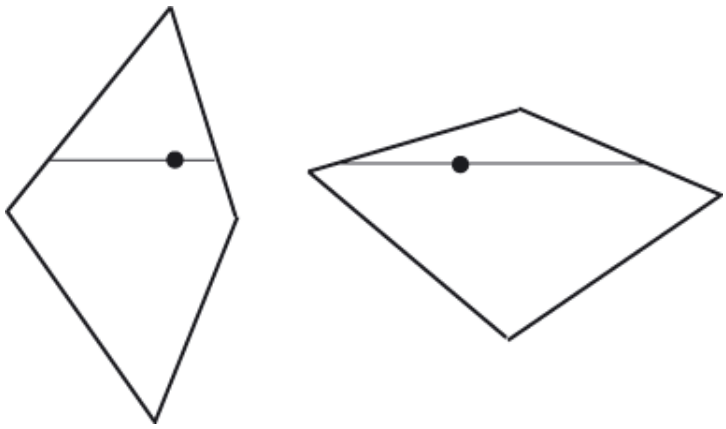


FIGURA 3.3.2. Per interpolare il colore di un punto occorre scrivere le sue coordinate come combinazioni convesse dei vertici del poligono a cui appartiene. Ma questa combinazione convessa non è unica, a meno che il poligono non sia un triangolo, perché per scrivere una combinazione convessa bastano tre vertici. Di solito si scelgono i primi tre vertici dall'alto al basso del viewport. In tal caso, se in una animazione il poligono ruota, il colore del punto considerato può cambiare improvvisamente quando la terna utilizzata cambia

In tal caso, il colore risultante può cambiare. Ad esempio, se nella disposizione a sinistra i tre vertici piĂč alti del quadrilatero sono neri ma il vertice in basso è bianco, il colore del punto evidenziato è una combinazione convessa del colore dei tre vertici neri, e quindi è nero. Invece, nella disposizione di destra, esso è una combinazione convessa di due vertici neri ed uno bianco, e quindi è grigio. Il problema nasce dal fatto che in questo caso determiniamo il colore come combinazione convessa dei tre vertici piĂč alti, ma i punti interni del quadrilatero non si esprimono in modo unico come combinazione convessa di tre dei suoi vertici. Però, se invece di un quadrilatero fossimo partiti con un triangolo, il problema non si sarebbe posto, perché i punti interni di un triangolo si esprimono in modo unico come combinazione convessa dei tre vertici. Quindi questo problema si risolve scindendo tutti i poligoni della modellazione in triangoli. Osserviamo che, se non risolto in questo modo, il problema ha un impatto visivo molto fastidioso nelle animazioni, dove accadrebbe che i punti interni di un oggetto ruotante possano cambiare colore per effetto della rotazione rispetto alle coordinate dell'osservatore. Si noti che questo problema capita non solo nell'interpolazione per l'ombreggiatura, ma anche

in tutti i tipi di interpolazione, ad esempio quella per il calcolo della profondit profondità nell'algoritmo di z-buffer o per la determinazione di aree nascoste nell'algoritmo di scan-line.

Vertici interni non condivisi. Un altro artefatto si ha quando un poligono possiede un vertice che separa due lati allineati, mentre il poligono adiacente vede quei due lati allineati come un unico lato, quindi senza il vertice intermedio. In tal caso, per il primo poligono quel vertice ha un colore assegnato dal modellatore, mentre per il secondo ha il colore interpolato a partire dai due vertici estremi dell'unico lato in cui giace. Quindi in generale il colore è diverso nei due casi, e questo fatto, a causa dell'interpolazione, porta ad una discontinuità del colore su tutto il segmento comune ai due poligoni, un artefatto di terribile impatto visivo perché distrugge la percezione della rotondità che l'ombreggiatura vuole simulare e la trasforma nella percezione di uno strappo. Per risolvere il problema occorre scindere il lato unico del secondo poligono in due nuovi lati allineati, che si congiungono al vertice non condiviso: in altre parole, occorre aggiungere tale vertice anche al database del secondo poligono.

Aliasing. Infine, un artefatto tipico nasce da problemi di risoluzione insufficiente, cioè di aliasing. Infatti può succedere che i versori normali che il procedimento di interpolazione calcola ai vertici siano poco adatti a rappresentare la superficie. Supponiamo ad esempio che la superficie abba la forma di una lamiera ondulata, diciamo di equazione $f(x, y) = \sin(px)$, e che la poligonale che la approssima consista di quadrilateri i cui vertici abbiano ascisse x intere, $x = 0, 1, 2, \ldots$. Allora i quadrilateri nello spazio hanno estensioni in x pari a p, ma la loro escursione in altezza y vale 2, perché in un semiperiodo la funzione seno ha escursione 2: piĂč precisamente, all'aumentare di x l'altezza dei quadrilateri alternativamente aumenta o diminuisce di 2 unità. Quindi i versori normali dei quadrilateri giacciono nel piano $\{x, y\}$ e puntano verso l'alto con inclinazioni di pendenza pari alternativamente a $2/p$ e $-2/p$. Quando l'algoritmo di interpolazione provvede a calcolare il valore delle normali ai vertici, esso prende la media di due consecutivi di tali versori, ed ottiene nuovi versori tutti diretti verticalmente verso l'alto, come se la superficie fosse orizzontale piatta. Questo problema, come sempre con l'aliasing, si riduce aumentando la risoluzione, cioè infittendo la partizione.

CAPITOLO 4

Ray tracing ricorsivo

4.1. Legge di Snell e versore trasmesso in una rifrazione

All'attraversare la superficie di confine fra due materiali (evidentemente semitrasparenti, visto che lasciano passare la luce), un raggio di luce si rifrange. Il raggio incidente (cioè quello che transita nel primo materiale) viene deviato in maniera da soddisfare la seguente relazione fra l'angolo θ_i che esso con la normale alla superficie e l'analogo angolo θ_t formato dal raggio rifratto, cioè trasmesso al secondo materiale (legge di Snell):

$$\eta_{i\lambda} \sin\theta_i = \eta_{t\lambda} \sin\theta_t.$$

Qui $\eta_{i\lambda}$ e $\eta_{t\lambda}$ sono i rispettivi indici di rifrazione dei due materiali, definiti come il rapporto tra la velocità della luce nel vuoto e nel materiale: essi variano al variare della lunghezza d'onda λ della luce. Il vuoto ha quindi indice di rifrazione 1, i materiali hanno indice di rifrazione più elevato.

È utile derivare dalla legge di Snell una formula per la direzione del raggio trasmesso, cioè per il suo versore che denotiamo con $\boldsymbol{T}$.

Per prima cosa, determiniamo il versore $\boldsymbol{M}$ perpendicolare al versore normale $\boldsymbol{N}$ che giace nel piano generato da $\boldsymbol{N}$ e dal versore della direzione di incidenza $\boldsymbol{I}$, e punta verso il lato opposto di $\boldsymbol{I}$ rispetto a $\boldsymbol{N}$. La proiezione di $\boldsymbol{I}$ lungo $\boldsymbol{N}$ vale $\boldsymbol{N}\cos\theta_i$. Perciò il vettore $\boldsymbol{N}\cos\theta_i - \boldsymbol{I} = \langle \boldsymbol{N}, \boldsymbol{I}\rangle \boldsymbol{N} - \boldsymbol{I}$ è perpendicolare a $\boldsymbol{N}$. Poiché $\boldsymbol{N}$ e $\boldsymbol{I}$ hanno norma 1 (sono versori!) e $\boldsymbol{N}\cos\theta_i - \boldsymbol{I}$ è ortogonale a $\boldsymbol{I}$, per il teorema di Pitagora la norma di $\boldsymbol{N}\cos\theta_i - \boldsymbol{I}$ vale $\sqrt{1-\cos^2\theta_i} = \sin\theta_i$ (il seno è positivo perché l'angolo θ_i è compreso fra 0 e $\pi/2$). Quindi, normalizzando, si ottiene

$$\boldsymbol{M} = \frac{\boldsymbol{N}\cos\theta_i - \boldsymbol{I}}{\sin\theta_i}.$$

Ora, per la legge di Snell, il versore del raggio trasmesso è

$$\boldsymbol{T} = \sin\theta_t \boldsymbol{M} - \cos\theta_t \boldsymbol{N} = \frac{\sin\theta_t}{\sin\theta_i}\,(\boldsymbol{N}\cos\theta_i - \boldsymbol{I}) - \cos\theta_t \boldsymbol{N}.$$

D'altra parte, sempre per la legge di Snell, si ha

$$\frac{\sin\theta_t}{\sin\theta_i} = \frac{\eta_{i\lambda}}{\eta_{t\lambda}}\,.$$

Quindi, ponendo $\eta_{\tau\lambda} := \eta_{i\lambda}/\eta_{t\lambda}$, otteniamo

$$\boldsymbol{T} = (\eta_{\tau\lambda}\cos\theta_i - \cos\theta_t)\boldsymbol{N} - \eta_{\tau\lambda}\boldsymbol{I}.$$

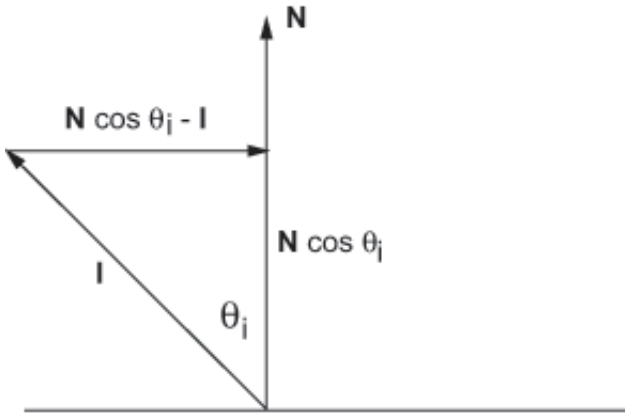


FIGURA 4.1.1. Il versore incidente

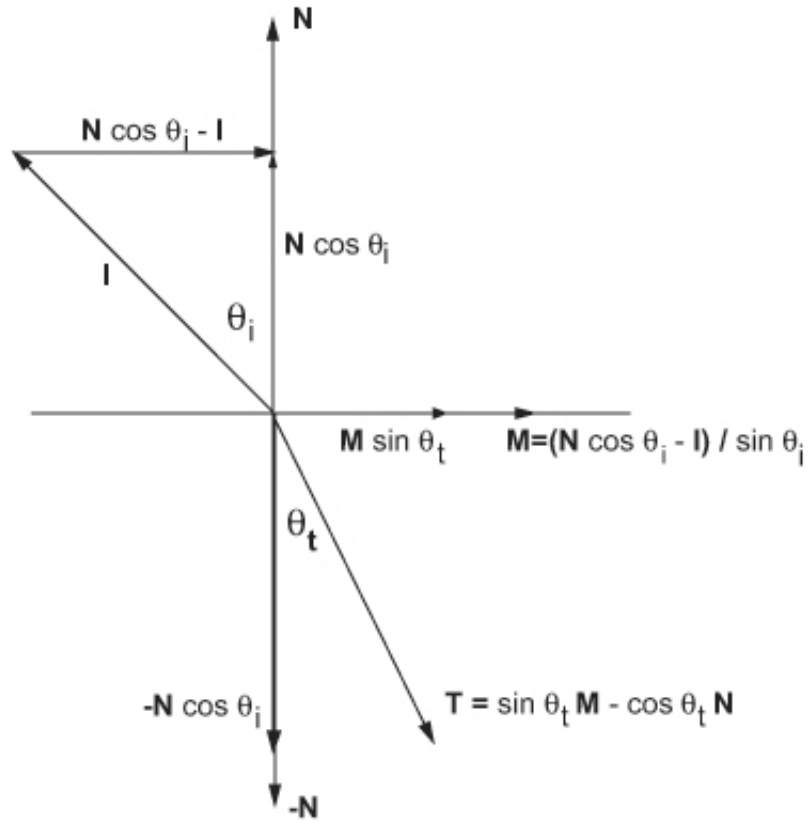


FIGURA 4.1.2. Il versore nella direzione della trasmissione rifrattiva

Ma $\cos\theta_i = \langle \boldsymbol{N}, \boldsymbol{I} \rangle$, e

$$\cos\theta_t = \sqrt{1-\sin^2\theta_t} = \sqrt{1-\eta_{\tau\lambda}^2 \sin^2\theta_i} = \sqrt{1-\eta_{\tau\lambda}^2(1-\cos^2\theta_i)}$$

$$= \sqrt{1-\eta_{\tau\lambda}^2(1-\langle \boldsymbol{N}, \boldsymbol{I} \rangle^2)}.$$

Pertanto possiamo scrivere $\boldsymbol{T}$ unicamente in termini di prodotti scalari e radici quadrate:

$$\boldsymbol{T} = \left(\eta_{\tau\lambda}\langle \boldsymbol{N}, \boldsymbol{I} > -\sqrt{1 - \eta_{\tau\lambda}^2(1 - \langle \boldsymbol{N}, \boldsymbol{I}\rangle^2)}\right)\boldsymbol{N} - \eta_{\tau\lambda}\boldsymbol{I}. \tag{4.1.1}$$

Si noti che i prodotti scalari sono eseguiti in brevissimo tempo dall'unità aritmetico-logica del processore di un elaboratore, perché richiedono solo somme e prodotti) e di radici quadrate, senza bisogno di più onerose funzioni trigonometriche:

4.1.1. Riflessione totale interna. Nelle figure precedenti abbiamo considerato un raggio di luce che passa da un mezzo ad indice di rifrazione basso (ad esempio l'aria) ad un mezzo con indice di rifrazione più elevato (ad esempio l'acqua o il vetro): dalla legge di Snell segue che il raggio rifratto si avvicina alla direzione della normale alla superficie di separazione dei due mezzi. Nel caso contrario, di un raggio di luce che passa dall'acqua all'aria, esso dovrebbe invece allontanarsi dalla normale. C'è però un'eccezione a questa regola. Consideriamo un raggio di luce nell'aria pressoché radente alla superficie di separazione con l'acqua: esso penetra nell'acqua con l'angolo più aperto possibile, cioè più lontano possibile dalla normale: quest'angolo, che viene chiamato angolo limite, dalla legge di Snell risulta valere $\arccos(\eta_{t\lambda}/\eta_{i\lambda})$. Consideriamo allora un altro raggio, che viaggia dall'acqua all'aria, ma con angolo superiore all'angolo limite. Per la reversibilità dei percorsi ottici, questo raggio non può essere trasmesso all'aria (ed in effetti, se lo fosse, dovrebbe avere angolo di trasmissione superiore a $\pi/2$, e quindi essere riflesso indietro nell'acqua!). In effetti, l'ottica prevede che un tale raggio sia interamente riflesso indietro al mezzo di provenienza, secondo le leggi della riflessione (riflessione totale). Di questo si deve tener conto nel rendering: in una scena subacquea un osservatore che guarda verso la superficie dell'acqua vede la luce e la scena dell'aria soprastante solo entro il cono di ampiezza pari all'angolo limite: al di fuori invece vede la riflessione interna della scena subacquea.

4.2. L'algoritmo di Ray Tracing ricorsivo

Ora estendiamo l'algoritmo di Ray Tracing per includere ombre, riflessioni e rifrazioni. Questa estensione è dovuta a Whitted [**52**] e Kay [**27**].
Questa variante dell'algoritmo di Ray Tracing determina il colore di un pixel tramite il colore del punto di intersezione fra il raggio di proiezione (cioè proveniente dal punto di osservazione) che passa per il centro di quel pixel ed il primo oggetto della scena che esso interseca, in base ai consueti modelli d'illuminazione (tipicamente l'equazione dell'illuminazione di Phong, ma si impiegano spesso anche modelli fisici di illuminazione), però iterando ricorsivamente la procedura per generare altri raggi a partire dal punto di intersezione: uno riflesso ed uno rifratto. Ciascuno di questi due raggi viene tracciato per determinare (se e) quale oggetto della scena interseca per primo: per tale oggetto, al nuovo punto di intersezione, si calcola l'intensità di luce (vista dal precedente punto di intersezione), data dall'equazione dell'illuminazione di Phong, considerando ora come direzione dell'osservatore quella individuata dalla direzione di provenienza del raggio, ossia quella del vecchio punto di intersezione inizialmente osservato. In altre parole, si considera il contributo all'illuminazione del punto inizialmente osservato dato dalla luce inviata verso di esso dall'oggetto più vicino nella direzione speculare rispetto all'osservatore (e nella direzione rifratta, se il punto giace su una superficie parzialmente trasparente). Ovviamente attenuiamo le rispettive illuminazioni moltiplicandole per il fattore (compreso fra 0 e 1) di riflettività o di rifrazione del materiale al punto originalmente osservato.

Poi si procede ricorsivamente: si considerano i due raggi generati ed ai due punti della scena che hanno toccato li si riflette e rifrange nuovamente, quindi ciascuno di essi genera due nuovi raggi, che vengono tracciati per trovare ulteriori punti di intersezione con la scena e sommarne i contributi di luce (che sono quindi contributi di due riflessioni o rifrazioni consecutive, ossia del secondo ordine), e così via.
Inoltre, per calcolare le ombre, tracciamo un altro raggio addizionale, dal punto di intersezione con l'oggetto, verso ciascuna delle sorgenti di luce (questi raggi si chiamano *raggi d'ombra*). Se uno di questi raggi d'ombra interseca un oggetto lungo la sua traiettoria, e quindi non raggiunge la sorgente di luce, allora il punto dell'oggetto da cui il raggio è partito è in ombra rispetto alla data sorgente di luce, ed in tal caso l'algoritmo di rendering ignora il contributo del raggio luminoso proveniente da quella sorgente.

Quale è il motivo di aggiungere un nuovo raggio riflesso ed uno rifratto? Per spiegare questo punto, consideriamo l'equazione dell'illuminazione di Phong. Essa determina, sia pure in via euristica e non fisica, l'intensità di luce nel punto osservato causata dalla luce ambientale, quella diffusa generata dalle sorgenti ed il bagliore riflesso specularmente a partire dalle sorgenti: però non tiene conto della luce che arriva sul punto osservato non direttamente dalle sorgenti, bensì riflessa da altri oggetti della scena (o meglio, ne tiene conto implicitamente nel termine di illuminazione ambientale, che però è impreciso perché non direzionale: il nostro modello di luce ambientale non dipende dalla posizione degli oggetti e delle luci, mentre la luce riflessa da un oggetto sugli altri ne risente). Tracciare dal punto osservato un raggio nuovo raggio nella direzione riflessa e sommare alla illuminazione del punto osservato il valore, opportunamente attenuato, della illuminazione del punto della scena colpito da questo nuovo raggio significa tenere conto della luce riflessa dall'oggetto colpito, ma solo di quella proveniente dalla direzione esattamente speculare rispetto alla direzione di osservazione. Riassumendo, il metodo aggiunge all'illuminazione diretta un ulteriore contributo dovuto alla riflessione da altri oggetti, ma solo quella proveniente dalla direzione di massima riflessione, trascurando contributi di riflessione in direzioni deflesse rispetto a quella perfettamente speculare. Pertanto il Ray Tracing ricorsivo, già al primo livello di ricorsività, accentua l'illuminazione diretta aggiungendovi l'effetto di una riflessione, ma solo sotto forma di bagliore intorno alla direzione speculare, ossia senza rendere gli effetti speculari veri e propri. Iterando una seconda volta il procedimento aggiungiamo alla luce proveniente dalla riflessione perfettamente speculare da parte di un altro oggetto l'ulteriore contributo di illuminazione che quel dato oggetto riceve a sua volta dalla direzione perfettamente speculare rispetto a quella in cui viene osservato (la direzione del primo raggio riflesso). Quindi l'iterazione ricorsiva, passo dopo passo, raffina e rende via via più preciso il rendering degli effetti delle riflessioni secondarie fra oggetti, ma sempre limitatamente alla specularità perfetta, non a quella alle varie deviazioni angolari (se si intendessero includere contributi di riflessioni non perfettamente speculari si dovrebbe utilizzare un metodo più sofisticato del Ray Tracing, ossia un metodo che genera raggi non solo nella direzione speculare ma anche in direzioni distribuite, possibilmente vicine a questa: ciò si può fare mediante il Ray Tracing ricorsivo stocastico, ovvero distribuito statisticamente). Analoga evidenziazione avviene per i contributi della interrifrazione fra oggetti. Perciò il Ray Tracing ricorsivo è un metodo ideale per evidenziare gli effetti perfettamente speculari e rifrattivi, e non per quelli diffusi: la variante di Ray Tracing ricorsivo stocastico permette un trattamento più accurato della riflessione e rifrazione con parziale diffusione, ma richiede un maggior numero di raggi a meno di non rassegnarsi ad elevati livelli di rumore. Un procedimento stocastico di illuminazione globale, che tiene conto di tutti i raggi riflessi e non solo

di quelli speculari, basato su una equazione integrale ricorsiva, è stato introdotto da Kajiya in [**25**].

Ora studiamo in maggior dettaglio i raggi secondari tracciati per effettuare il rendering della riflessione speculare e della trasparenza rifrattiva.

I raggi d'ombra, di riflessione e di rifrazione sono chiamati raggi secondari, per distinguerli dai raggi di proiezione primari provenienti dall'osservatore Se l'oggetto intersecato da un raggio riflette in modo speculare, allora viene generato un raggio di riflessione, diretto verso la direzione $\boldsymbol{R}$ speculare a quella di incidenza. Se l'oggetto è trasparente, e se non si verifica una riflessione totale interna, allora viene inviato un raggio di rifrazione nell'oggetto nella direzione $\boldsymbol{T}$ stabilita dalla legge di Snell sulla rifrazione.

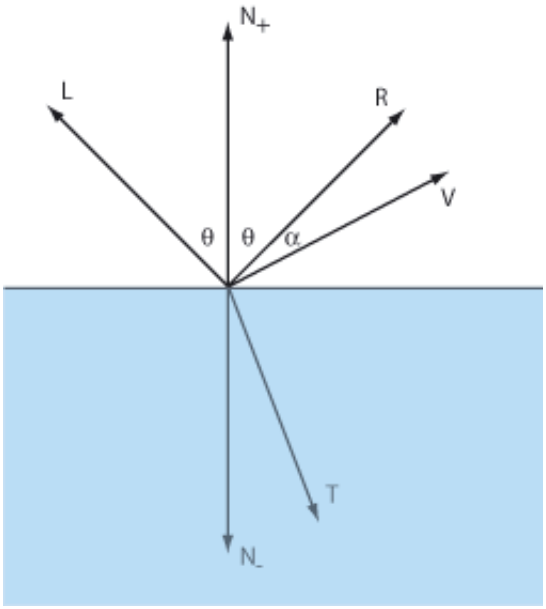


FIGURA 4.2.1. Versori del raggio proiettore, del raggio riflessi e del raggio trasmesso (rifratto), per l'applicazione del metodo di illuminazione di Phong nel ray Tracing ricorsivo

Inoltre tutti questi raggi di riflessione e rifrazione, qualora intersechino un nuovo oggetto, possono produrre in maniera ricorsiva raggi di ombra, riflessione e rifrazione. Così i raggi formano una struttura gerarchica ad albero. I nodi dell'albero sono i punti di intersezione, ed i segmenti sono i raggi. Si prefissa il numero massimo di generazioni (cioè di fasi di ricorrenza) che verranno considerate in quest'albero, in base ad una soglia prefissata di energia luminosa al di sotto della quale ci si deve necessariamente ridurre dopo un tale numero di generazioni, a causa della attenuazione complessiva dei contributi ricorsivi, causata, generazione dopo generazione, dalla moltiplicazione successiva dei coefficienti di riflessività e di trasmissibilità rifrattiva. Si può usare una stima uniforme, basata sul valore massimo di tali coefficienti fra i materiali della scena: alla n-sima generazione ciascun contributo non può superare questo valore elavato alla potenza n. Oppure si può optare per una scelta di numero massimo di generazioni a passo variabile, ossia, come si suol dire, *adattiva*, tenendo conto degli effettivi valori dei coefficienti di attenuazione per i materiali incontrati da ciascun raggio nel corso delle sue riflessioni o rifrazioni alle varie generazioni, e fermando la ricorrenza quando il risultato diventa inferiore ad una soglia prefissata.

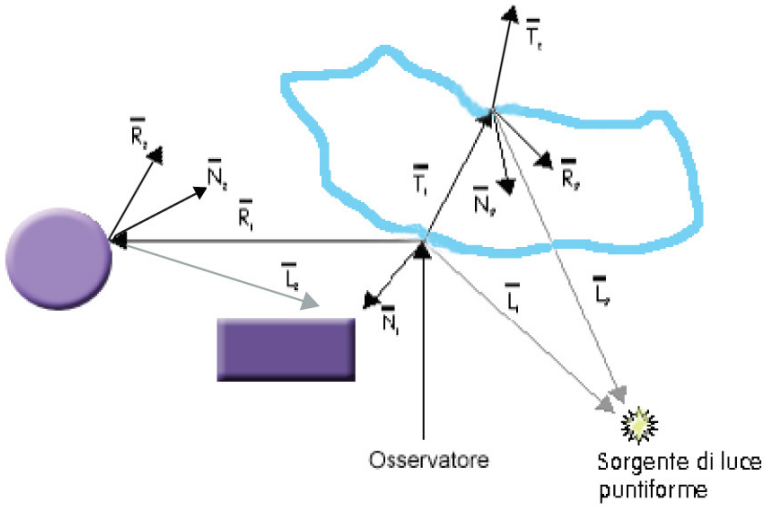


FIGURA 4.2.2. Raggi proiettori e loro raggi riflessi e rifratti, e corripondenti raggi d'ombra (ossia diretti verso le sorgenti di luce) in generazioni successive del ray Tracing ricorsivo

Nell'algoritmo di Whitted, un ramo dell'albero termina nei casi seguenti: se i raggi riflessi e rifratti non intersecano alcun oggetto, se si raggiunge il massimo numero predefinito di generazioni o se il sistema esaurisce la memoria. Per calcolare l'illuminazione di un pixel, i rami dell'albero che hanno inizio con il raggio proiettore che passa per il centro di quel pixel sono percorsi ricorsivamente dal basso in alto: a ciascun nodo (cioè punto di intersezione) l'intensità è determinata in base all'intensità dei suoi immediati discendenti. In tal modo, risalendo l'albero, ogni raggio accumula energia, e la somma di tutti i raggi derivati da quello che passa per il centro del pixel è l'illuminazione totale del pixel.

L'equazione dell'illuminazione usata da Whitted è la seguente equazione ricorsiva:

$$I_\lambda = I_{a,\lambda} k_a O_{d,\lambda} + \sum_{i=1}^{m} f_{\text{att}}(i) S_i I_{p,\lambda}(i) \left(k_d O_{d,\lambda} \langle \boldsymbol{L}, \boldsymbol{N} \rangle + k_s \langle \boldsymbol{R}, \boldsymbol{V} \rangle^n\right) + k_s I_{r,\lambda} + k_t I_{t,\lambda}$$

dove le sorgenti di luce sono m, e per la $i-$sima sorgente di luce:

- $I_{r,\lambda}$ è l'intensità del raggio riflesso, $I_{t,\lambda}$ quella del raggio trasmesso (rifratto), $I_{a,\lambda}$ è l'intensità della luce ambientale, $I_{p,\lambda}(i)$ è l'intensità della ($i-$sima) sorgente, ed i corrispondenti coefficienti O rappresentano la distribuzione spettrale del colore di queste forme di illuminazione al variare della frequenza λ;
- k_a e k_s sono i coefficienti di attenuazione per la luce ambientale e la luce riflessa specularmente al punto osservato in seguito ad illuminazione diretta dalla sorgente, k_t è il coefficiente di rifrazione al punto osservato: tutti questi coefficienti possono variare fra 0 e 1 (il coefficiente k_s è il coefficiente di riflessione speculare del materiale al punto in cui è stato generato il raggio riflesso, ed analogamente per k_t ed il raggio rifratto, ma naturalmente i contributi delle generazioni successive, $I_{s,\lambda}$ e $I_{t,\lambda}$, deve

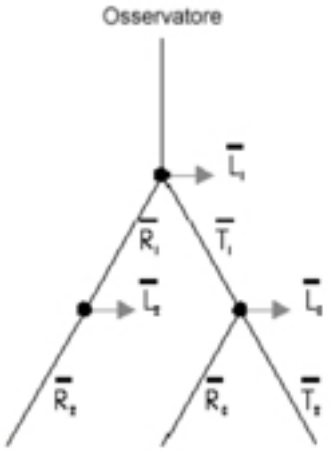


FIGURA 4.2.3. Albero genealogico dei raggi riflessi, ritratti e di ombra alle varie generazioni dell'algoritmo di Whitted

essere applicato il coefficiente di riflessione speculare e di rifrazione del punto di volta in volta colpito dai raggi riflessi di tali generazioni); la somma di tutti questi coefficienti di attenuazione dovrebbe valere 1, per la conservazione dell'energia, ma negli esempi ed esercizi trascureremo questa richiesta, lasciando al lettore il compito di rinormalizzare i risultati;

- per ogni raggio riflesso o rifratto, il versore del punto di osservazione cambia generazione dopo generazione: volta per volta è quello diretto verso il punto della scena in cui sono stati generati tali raggi;
- il valore di S_i è 1 oppure 0 a seconda del fatto, che la prima sorgente di luce sia visibile oppure no dal punto illuminato (cioè che il punto sia illuminato o in ombra). Invece che trattare S_i come una funzione a valori 0 e 1, il metodo la sostituisce con una funzione continua del coefficiente di trasmissione k_t degli altri oggetti precedentemente attraversati in modo che gli oggetti parzialmente trasparenti assorbano meno luce di quelli opachi.
- Per ogni punto della scena toccato da un raggio proiettore proveniente dall'osservatore, i termini residui $I_{r,\lambda}$ e $I_{t,\lambda}$ sono determinati da questa stessa equazione applicata questa volta ai punti della superficie più vicina che vengono toccati dai raggi riflessi e trasmessi (da qui la ricorsività del procedimento).

Per approssimare l'attenuazione dovuta alla distanza, per ogni raggio l'algoritmo di Whitted moltiplica il valore calcolato di I_λ per l'inverso della distanza percorsa dal raggio.

Questo algoritmo presenta alcuni problemi che presentiamo in una pagina separata.

4.2.1. Pseudocodice del Ray Tracing ricorsivo.

```
Variabili globali accessibili a tutte le procedure:
Array di punti dove sono le luci;
BRDF (es: Lambert, Phong, Cook-Torrance) per ogni triangolo della scena

Si sceglie un punto di osservazione (centro di proiezione)
```

```
          e la finestra;
Per ogni scan line nell'immagine {
Per ogni pixel nella scan line  {
Si determina il raggio che proviene dell'osservatore e che
   attraversa il pixel;
lo si traccia intersecandolo con la scena:
pixel=RT_trace (raggio, 1);
}
}

/*intersezione
RT_colore RT_trace (RT_raggio raggio, int generazione)  {
Se generazione <= generazione_massima
si determina l'intersezione piu' vicina tra il raggio
    ed un oggetto;
Se (un oggetto viene intersecato ed e' il piu' vicino all'osservatore)  {
si calcola la normale al punto di intersezione;
        restituisce RT_shading (raggio tracciato, oggetto intersecato piu'
        vicino, punto di intersezione, normale,
        generazione) ;
}
altrimenti
restituisce il valore dello sfondo;
}

/* Ombre, riflessioni e rifrazioni
/* commento: il parametro di tipo RT_raggio nella prossima procedura RT_shading
/* rappresenta un raggio d'ombra

}
RT_colore RT_shading ( RT_raggio raggio1, RT_oggetto oggetto1,
      RT_punto punto1,
      RT_normale normale1, int generazione)
\*
Traccia i raggi d'ombra dal punto alla direzione
        delle sorgenti di luce e restituisce l'illuminazione
        al punto osservato,
        calcolando anche il fattore di attenuazione RT_ombra dovuto ad
        oggetti opachi o semitrasparenti che il raggio attraversa prima
        di arrivare alla sorgente, e memorizzando tale fattore di
        attenuazione, per la sorgente i-sima, in una variabile
        CoeffOmbra_i di tipo RT_colore; poi traccia un raggio riflesso
        ed uno trasmesso de ne aggiungi i contributi, calcolati
        analogamente, in maniera ricorsiva.
        Ora spieghiamo questa aggiunta ricorsiva.
*\
{
RT_colore colore1;
RT_raggio  RaggioRifl, RaggioTrasm, RT_raggio Array RaggioOmbra_i ;
```

```
RT_colore  ColoreRifl, ColoreTrasm;

colore1= ambiente;
Per ogni sorgente di luce
\*
{   Sunto:
RaggioOmbra_i = raggio diretto da punto1 alla sorgente
     di luce numero i;
Se il prodotto scalare fra la normale e la
     direzione della luce e' positivo   {
si calcola  quanta luce e' bloccata dalle superficie
     intermedie opache o trasparenti,
    si calcola per ogni sorgente di luce il contributo di diffusione
         e quello speculare e si aggiungono a colore1,
         come spiegato qui di seguito
}
*\

Se la generazione \'{e} minore della generazione massima  {
    Per tutte le sorgenti di luce i = 1,..., m
    {
\* dal punto osservato a tale generazione si richiama RT_trace
passando a RT_shading tale punto come punto1
(e RT_trace a sua volta richiama RT_shading)
*\
RaggioOmbra_i=RT_Trace{raggio nella direzione della
 sorgente i, int generazione};
colore1 = colore1 + RT_colore RT_shading (RaggioOmbra_i,
generazione+1);
         {Commento: CoeffOmbra_i e' il fattore di attenuazione della
         luce della sorgente i dovuta ad oggetti opachi
         o semitrasparenti interposti fra il punto da
         illuminare e la sorgente i; il suo tipo di dati e' RT_colore}

Se l'oggetto e' riflettente   {
RaggioRifl=raggio nella direzione della
 riflessione speculare;
ColoreRifl=RT_trace (RaggioRifl, generazione+1);
    si scala ColoreRifl con il coefficiente di riflessione
        speculare del punto da cui parte il rgggio
        e con CoeffOmbra_i del punto dove arriva
        e lo si aggiunge a colore1;
}
Se l'oggetto e' trasparente   {
RaggioTrasm = raggio in direzione della rifrazione;
Se non si ha una riflessione totale interna   {
ColoreTrasm=RT_trace (RaggioTrasm, generazione +1);
        si scala ColoreTrasm  con il coefficiente di trasmissione
        con il coefficiente di riflessione
```

```
        speculare del punto da cui parte il raggio
        e con CoeffOmbra_i del punto dove arriva
        e lo si aggiunge a colore1;
   }
}
}
restituisce colore1;
}
```

4.2.2. Spiegazione dello pseudocodice dell'algoritmo del Ray Tracing Ricorsivo. Il metodo *RT_ trace* determina il punto di intersezione più vicino al punto di visuale di un raggio di proiezione con un oggetto della scena, e richiama il metodo *RT_ ombra* per determinare l'illuminazione in quel punto. Dapprima, *RT_ ombra* calcola la componente di luce ambientale dell'illuminazione, poi traccia raggi d'ombra verso ogni sorgente di luce per determinare se il punto di intersezione sia in ombra rispetto a qualcuna delle sorgenti, che quindi non dà apporto all'illuminazione. Poi il metodo verifica se l'oggetto intersecato è riflettente od opaco, e se è opaco o (semi)trasparente. Un oggetto opaco scherma totalmente la luce, mentre un oggetto riflettente lascia continuare il raggio incidente in un raggio riflesso, ed uno trasparente in un raggio rifratto: per tracciare questi eventuali nuovi raggi viene richiamato ricorsivamente su entrambi il metodo *RT_ trace*. La direzione del raggio riflesso dipende solo dalla direzione di incidenza e dalla direzione normale alla superficie; per calcolare la direzione del raggio rifratto sono necessari gli indici di rifrazione dei due materiali per cui il raggio incidente sta transitando, che vengono inclusi fra i dati di ogni raggio generato. *RT_ trace* costruisce l'albero del raggio proiettore, ma lo mantiene solo il tempo necessario a determinare il colore corrente del pixel.

4.3. Esercizi sul Ray Tracing ricorsivo

ESERCIZIO 4.3.1. La scena consiste dell'interno del tetraedro che ha per vertici l'origine ed i punti $(1,0,0),(0,1,0),(0,0,1)$. Le pareti interne del tetraedro sono bianche con coefficiente di Lambert e di riflessione speculare uguali a $1/2$ ed esponente di Phong 1. L'osservatore è nel vertice alto $(0,0,1)$, e la sorgente di luce nell'origine. L'osservatore guarda verso il punto della base $\boldsymbol{p} = (1/2\sqrt{2}, 1/2\sqrt{2}, 0)$. Si mostri che il Ray Tracing dà illuminazione nulla, e quindi è necessario arrivare almeno alla prima generazione di raggi riflessi nel Ray Tracing ricorsivo per avere un risultato realistico.

Suggerimento: La sorgente di luce è nel piano di base, e quindi la luce al punto $\boldsymbol{p}$ è radente, ossia la sua direzione di provenienza è perpendicolare al versore normale: quindi il contributo di Lambert è nullo. Inoltre, al punto $\boldsymbol{P}$ la direzione $\boldsymbol{R}$ della luce riflessa è quella del versore della bisettrice del primo quadrante del piano di base (uscente dall'origine), mentre la direzione $\boldsymbol{V}$ dell'osservatore (che si trova sul semiasse z positivo) ha componente sul piano di base anch'essa diretta lungo la bisettrice ma in senso opposto, verso l'origine: quindi il coseno dell'angolo fra $\boldsymbol{R}$ e $\boldsymbol{V}$ è negativo ed il contributo di Phong deve essere posto uguale a zero. □

ESERCIZIO 4.3.2. Una scena consiste di due sfere T_1 e T_2 con centro l'origine e raggio rispettivamente 1 e 7, dalle pareti perfettamente speculari (cioè con coefficiente di riflettività pari a 1), ed un quadrato rosso puro con coefficiente di riflessione $\frac{1}{2}$ sul piano $\{z = 2\}$, con i lati paralleli agli assi coordinati, con centro in $(0,0,2)$ e lato pari a 3. L'osservatore si trova al punto $\boldsymbol{o} = (1,0,2)$, la luce a $\boldsymbol{l} = (0,4,0)$ ed è bianca. Il coefficiente di diffusione di Lambert del quadrato è $\frac{1}{2}$, tutti gli altri sono zero.

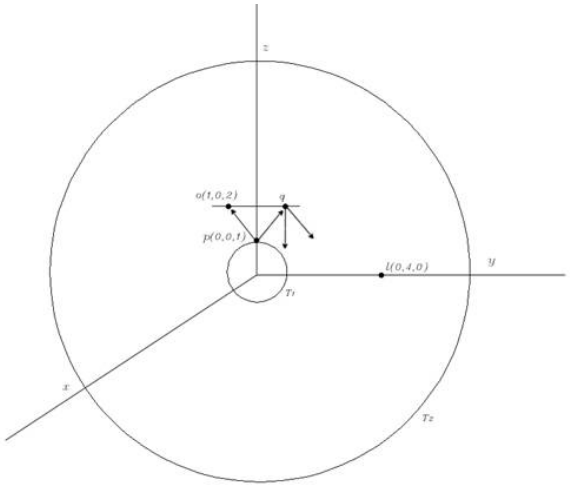


FIGURA 4.3.1. La scena dell'Esercizio 4.3.2

Si determini cosa vede l'osservatore quando guarda verso il punto (0,0,1); si ricavi il risultato mediante il procedimento di Ray Tracing ricorsivo, con due generazioni di tracciamento di raggi (oltre a quello originale). L'esponente di Phong di ciascun oggetto è 2.

Svolgimento.
Troviamo $\boldsymbol{V}_1$, il versore del raggio incidente in $\boldsymbol{p}$:

$$\boldsymbol{V}_1 = \frac{\boldsymbol{o} - p}{\|\boldsymbol{o} - p\|} = \frac{(1,0,2) - (0,0,1)}{\|(1,0,1\|} = \frac{(1,0,1)}{\sqrt{2}}$$

La normale in $\boldsymbol{p}$ è nella direzione del raggio:

$$\boldsymbol{N}_1 = (0,0,1)$$

Il raggio proiettore riflesso in $\boldsymbol{p}$ è dato da una espressione del tipo

$$\boldsymbol{R} = 2\langle \boldsymbol{N}, \boldsymbol{V} \rangle \boldsymbol{N} - \boldsymbol{V}$$

e quindi, poiché $\langle \boldsymbol{V}_1, \boldsymbol{N}_1 \rangle = \frac{1}{\sqrt{2}}$, troviamo

$$\boldsymbol{R}_1 = \frac{2}{\sqrt{2}}(0,0,1) - \left(\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right) = \left(-\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right)$$

Il punto $\boldsymbol{p}$ è in ombra perché la retta che congiunge il punto l (luce) con $\boldsymbol{p}$ interseca prima un altro punto della sfera. Quindi non bisogna calcolare il raggio d'ombra. Se non stessimo usando il Ray Tracing ricorsivo, il punto sarebbe stato in ombra e l'osservatore avrebbe visto un colore nero.

Prima generazione. Per calcolare il contributo della generazione successiva, spostiamo l'osservatore in $\boldsymbol{p}$. Troviamo il punto di intersezione $\boldsymbol{q}$ del raggio riflesso in $\boldsymbol{p}$ con il quadrato rosso.

$$\boldsymbol{R}(t) = p + t\boldsymbol{R}_1 = \begin{cases} x = -\frac{1}{\sqrt{2}}t \\ y = 0 \\ z = 1 + \frac{1}{\sqrt{2}}t \end{cases}$$

Mettiamo a sistema $\boldsymbol{p} + t\boldsymbol{R}_1$ con il piano $\{z = 2\}$:

$$\begin{cases} x = -\frac{1}{\sqrt{2}}t \\ y = 0 \\ 1 = \frac{1}{\sqrt{2}}t \end{cases}$$

quindi $t = \sqrt{2}$, e pertanto il punto osservato è $\boldsymbol{q} = (-1, 0, 2)$ Il versore normale è

$$\boldsymbol{N}_2 = -\boldsymbol{N}_1 = (0, 0, -1)$$

Troviamo $\boldsymbol{V}_2$, il versore della direzione del raggio proiettore di prima generazione incidente in $\boldsymbol{q}$:

$$\boldsymbol{V}_2 = \frac{p - q}{\|p - q\|} = \frac{(0, 0, 1) - (-1, 0, 2)}{\|(-1, 0, 1\|} = \frac{(1, 0, -1)}{\sqrt{2}} = -\boldsymbol{R}_1$$

Questo versore è quindi il nuovo versore dell'osservatore, che ora immaginiamo localizzato al punto $\boldsymbol{p}$. Troviamo la direzione del proiettore riflesso $\boldsymbol{R}_2$ al punto $\boldsymbol{q}$:

$$\begin{aligned} \boldsymbol{R}_2 &= 2\langle \boldsymbol{V}_2, \boldsymbol{N}_2 \rangle \boldsymbol{N}_2 - \boldsymbol{V}_2 \\ &= 2\left\langle \left(\frac{1}{\sqrt{2}}, 0, \frac{-1}{\sqrt{2}}\right), (0, 0, -1) \right\rangle (0, 0, -1) - \left(\frac{1}{\sqrt{2}}, 0, \frac{-1}{\sqrt{2}}\right) \\ &= \left(\frac{-1}{\sqrt{2}}, 0, \frac{-1}{\sqrt{2}}\right) \end{aligned}$$

Troviamo il versore della direzione della luce $\boldsymbol{L}_2$:

$$\boldsymbol{L}_2 = \frac{l - q}{\|l - q\|} = \frac{(0, 4, 0) - (-1, 0, 2)}{\|(1, 4, -2)\|} = \frac{(1, 4, -2)}{\sqrt{21}}$$

Infine, calcoliamo il versore $\boldsymbol{L'}_2$ riflesso speculare del versore della luce:

$$\begin{aligned} \boldsymbol{L'}_2 &= 2\langle \boldsymbol{L}_2, \boldsymbol{N}_2 \rangle \boldsymbol{N}_2 - \boldsymbol{L}_2 \\ &= 2\left\langle \frac{(1, 4, -2)}{\sqrt{21}}, (0, 0, -1) \right\rangle (0, 0, -1) - \frac{(1, 4, -2)}{\sqrt{21}} \\ &= \frac{(-1, -4, -2)}{\sqrt{21}} \end{aligned}$$

Poiché la luce è bianca, il quadrato è rosso ed i suoi coefficienti di diffusione e di riflessione sono $\frac{1}{2}$, il contributo del raggio di prima generazione alla luminosità è

$$\begin{aligned} \frac{1}{2}\langle \boldsymbol{N}_2, L_2 \rangle (1_R, 0_G, 0_B) + \frac{1}{2}\langle \boldsymbol{V}_2, \boldsymbol{L'}_2 \rangle^2 (1_R, 1_G, 1_B) & \\ = \frac{1}{\sqrt{21}}(1_R, 0_G, 0_B) + \frac{1}{84}(1_R, 1_G, 1_B) & \\ = (0.230_R,\, 0.012_G,\, 0.012_B) & \quad (4.3.1) \end{aligned}$$

(il primo termine è quello di Lambert ed il secondo quello di Phong).

Seconda generazione. Supponiamo che l'osservatore stia ora in $\boldsymbol{q}$. Troviamo il punto di intersezione w tra la retta r uscente da $\boldsymbol{q}$ in direzione $\boldsymbol{R}_2$ e la sfera T_2. L'equazione parametrica della retta è $r(t) = q + t\boldsymbol{R}_2$, ossia

$$\begin{cases} x = -1 - \frac{t}{\sqrt{2}} \\ y = 0 \\ z = 2 + \frac{t}{\sqrt{2}} \end{cases}$$

Intersecando con la sfera $x^2 + y^2 + z^2 = 49$ troviamo il parametro t del punto di intersezione:

$$(-1 - \frac{t}{\sqrt{2}})^2 + (2 - \frac{t}{\sqrt{2}})^2 = 49$$

$$1 + \frac{t^2}{2} + \frac{2t}{\sqrt{2}} + 4 + \frac{t^2}{\sqrt{2}} - 2\frac{2t}{\sqrt{2}} = 49$$

$$t^2 - \sqrt{2}t - 44$$

Le soluzioni sono:

$$t_{1,2} = \frac{1 \pm \sqrt{89}}{\sqrt{2}} \qquad \text{cioè} \quad t_1 = 7.4 \text{ e } t_2 = -5.25.$$

Un valore di t negativo corrisponde ad un'intersezione che avviene prima dell'inizio del raggio riflesso: quindi usiamo il valore positivo.
Il punto è: $w = (-6.2, 0, -3.2)$
Troviamo la normale in w:

$$\boldsymbol{N}_3 = \frac{0 - w}{\|0 - w\|} = \left(\frac{6.2}{7}\,,\, 0\,,\, \frac{3.2}{7}\right)$$

Troviamo $\boldsymbol{V}_3$, il versore del raggio proiettore di seconda generazione incidente in w (ossia il nuovo vettore della direzione dell'osservatore che si è spostato in $\boldsymbol{q}$):

$$\boldsymbol{V}_3 = \frac{q - w}{\|q - w\|} = \frac{(-1,\, 0,\, 2) - (-6.2,\, 0,\, 3.2)}{\|(5.2\,,\, 0,\, 5.2)\|} = \left(\frac{1}{\sqrt{2}}\,,\, 0,\, \frac{1}{\sqrt{2}}\right) = -\boldsymbol{R}_2$$

Troviamo il versore $\boldsymbol{L}_3$:

$$\boldsymbol{L}_3 = \frac{l - w}{\|l - w\|} = \frac{(0,\, 4,\, 0) - (-6.2,\, 0,\, 3.2)}{\|(6.2,\, 4,\, 3.2)\|} = \frac{(6.2\,,\, 4\,,\, 3.2)}{\sqrt{64.7}}$$

Troviamo la direzione $\boldsymbol{L'}_3$ riflessa speculare di $\boldsymbol{L}_3$ rispetto a $\boldsymbol{N}_3$:

$$\begin{aligned} \boldsymbol{L'}_3 &= 2\left\langle \frac{(6.2\,,\, 0,\, 3.2)}{7}\,,\, \frac{(6.2\,,\, 4\,,\, 3.2)}{\sqrt{64.7}} \right\rangle \left(\frac{6.2}{7}\,,\, 0,\, \frac{3.2}{7}\right) - \frac{(6.2,\, 4,\, 3.2)}{\sqrt{64.7}} \\ &= 2\,\frac{48.68}{7\sqrt{64.7}}\,\frac{(6.2,\, 0,\, 3.2)}{7} - \frac{(6.2,\, 4,\, 3.2)}{\sqrt{64.7}} \\ &= 0.247\,(6.2, 0, 3.2) - 0.124\,(6.2,\, 4,\, 3.2) = (0.763\,,\, 0.498\,,\, 0.394). \end{aligned}$$

Poiché la luce è bianca ed i coefficienti di diffusione e di riflessione della sfera esterna sono rispettivamente 0 e 1, ma il raggio proiettore si era riflesso al punto $\boldsymbol{q}$ che appartiene ad un materiale di coefficiente di riflessione uguale a 1, il contributo del raggio di seconda generazione alla luminosità è

$$\frac{1}{2}\langle \boldsymbol{V}_3, \boldsymbol{L'}_3\rangle^2 (1_R, 1_G, 1_B) = (0.335_R,\, 0.335_G,\, 0.335_B) \tag{4.3.2}$$

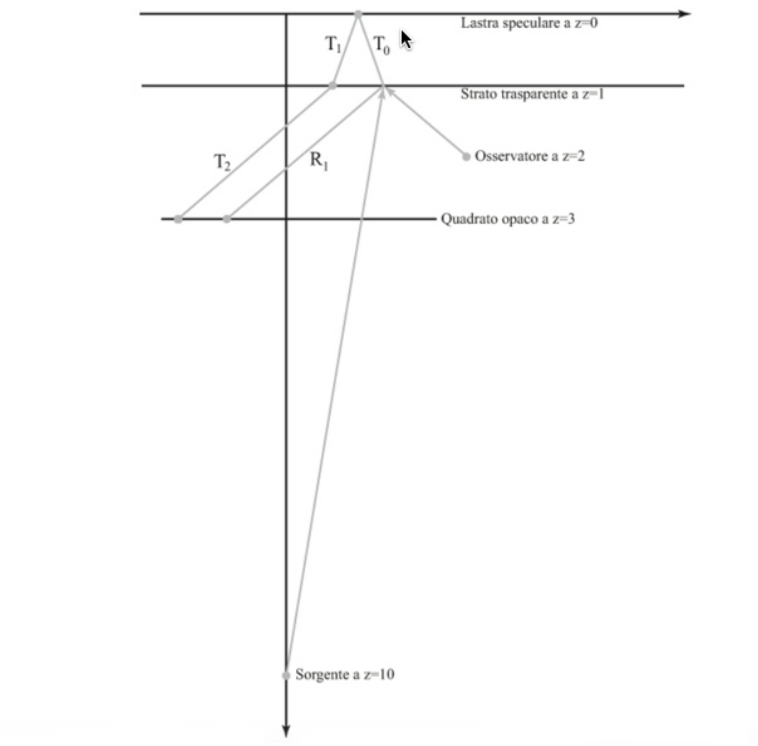


FIGURA 4.3.2. La scena dell'Esercizio 4.3.3

Infine calcoliamo l'intensità della luce che vede l'osservatore, sommando i risultati della prima e seconda generazione, ottenuti in (4.3.1) e (4.3.2):

$$\begin{aligned} I_p &= 0 + \Big[\frac{1}{2}\langle \boldsymbol{N}_2, \boldsymbol{L}_2\rangle(1_R, 0_G, 0_B) + \frac{1}{2}\langle \boldsymbol{V}_2, \boldsymbol{L'}_2\rangle^2(1_R, 1_G, 1_B)\Big] \\ &\quad + \frac{1}{2}\Big[\langle \boldsymbol{V}_3, \boldsymbol{L'}_3\rangle^2(1_R, 1_G, 1_B)\Big] \\ &= (0.313_R,\, 0.095_G,\, 0.095_B) + (0.335_R, 0.335_G, 0.335_B) \\ &= (0.648_R,\, 0.43_G,\, 0.43_B) \end{aligned}$$

□

ESERCIZIO 4.3.3. Un osservatore è collocato nel punto $\boldsymbol{o} = (2, 2, 2)$. Una sorgente isotropa di luce è collocata nel punto $\boldsymbol{l} = (0, 0, 10)$. L'osservatore guarda verso l'origine.
La scena si compone di una lastra metallica speculare sul piano $z = 0$, uno strato di materiale trasparente di coefficiente di rifrazione $\sqrt{2}$ che occupa la striscia $\{0 < z < 1\}$, un quadrato sul piano $\{z = 3\}$ di estensione $\{-1.4 \leqslant x \leqslant 1.4,\ -1.4 \leqslant y \leqslant 1.4\}$ ed una sfera con centro $(5, -5, 5)$ e raggio 2. Si segua il raggio di osservazione, si determini il punto $\boldsymbol{p}$ con la lastra trasparente, e da tale punto si traccino:

- il raggio riflesso $\boldsymbol{R}_1$
- il raggio rifratto $\boldsymbol{T}_0$
- il raggio $\boldsymbol{V}_{\text{sorg}}$ che si dirige verso la sorgente di luce.

Inoltre

(1) Per il terzo raggio, si determini se esso raggiunge la sorgente di luce oppure se prima interseca qualche altro oggetto della scena. Il punto $\boldsymbol{p}$ è in luce o in ombra?
(2) Per il primo raggio, si determini se esso interseca qualche altro oggetto della scena o si perde nello sfondo.
(3) Per il secondo raggio, si trovi il punto $\boldsymbol{q}$ in cui esso tocca la lastra speculare. Dal punto $\boldsymbol{q}$ si tracci il suo raggio riflesso $\boldsymbol{T}_1$, e si tracci un nuovo raggio diretto verso la luce . Questo nuovo raggio raggiunge la luce oppure prima interseca qualche altro oggetto? il punto $\boldsymbol{q}$ è in luce o in ombra?
(4) Si determini il punto s in cui il raggio $\boldsymbol{T}_1$ esce dalla lastra trasparente, e si segua il raggio rifratto (che esce quindi dalla lastra ed entra nell'aria). Tale raggio colpisce qualche altro oggetto? Dal punto s si tracci un nuovo raggio diretto verso la sorgente di luce. Tale raggio arriva alla sorgente di luce o viene fermato da qualche altro oggetto? Il punto s è in luce o in ombra?

Svolgimento. Per trovare il punto $\boldsymbol{p}$ trovo la retta passante per l'osservatore e con direzione $\boldsymbol{V}_1$.

$$\boldsymbol{V}_1 = \frac{\boldsymbol{0}-\boldsymbol{o}}{\|\boldsymbol{0}-\boldsymbol{o}\|} = \frac{(0,0,0)-(2,2,2)}{\|(-2,-2,-2\|} = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$$

$$\boldsymbol{R}(t) = \boldsymbol{o} + t\boldsymbol{V}_1 = \begin{cases} x = 2 + t\frac{-1}{\sqrt{3}} \\ y = 2 + t\frac{-1}{\sqrt{3}} \\ z = 2 + t\frac{-1}{\sqrt{3}} \end{cases}$$

Mettiamo a sistema l'equazione parametrica della retta $\boldsymbol{o} + t\boldsymbol{V}_1$ con quella della superficie dello strato trasparente $z = 1$ per trovare quale sia il punto osservatp $\boldsymbol{p}$:

$$\begin{cases} x - 2 = \frac{-t}{\sqrt{3}} \\ y - 2 = \frac{-t}{\sqrt{3}} \\ -1 = \frac{-t}{\sqrt{3}} \end{cases}$$

quindi $\boldsymbol{p} = (1,1,1)$.
Per trovare il versore del raggio riflesso $\boldsymbol{R}_1$, usiamo come sempre l'identità $\boldsymbol{R}_1 = 2 < \boldsymbol{N}_1, \boldsymbol{V}_p > \boldsymbol{N}_1 - \boldsymbol{V}_p$. Si ottiene:

$$\boldsymbol{V}_p = \frac{\boldsymbol{o}-p}{\|\boldsymbol{o}-p\|} = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right),$$

$$\boldsymbol{N}_1 = (0,0,1),$$

$$\langle \boldsymbol{N}_1, \boldsymbol{V}_p \rangle = \frac{1}{\sqrt{3}}$$

quindi

$$\boldsymbol{R}_1 = \frac{2}{\sqrt{3}}(0,0,1) - \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right) = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right).$$

Punto 2: troviamo la retta passante per $\boldsymbol{p}$ e con direzione $\boldsymbol{R}_1$.

$$\begin{cases} x = 1 - \frac{t}{\sqrt{3}} \\ y = 1 - \frac{t}{\sqrt{3}} \\ z = 1 + \frac{t}{\sqrt{3}} \end{cases}$$

Mettiamo a sistema con il quadrato ($z = 3$) :

$$\begin{cases} x = 1 - \frac{t}{\sqrt{3}} \\ y = 1 - \frac{t}{\sqrt{3}} \\ 2 = \frac{t}{\sqrt{3}} \end{cases}$$

quindi $\boldsymbol{p}_1 = (-1, -1, 3)$.
Per trovare il versore del raggio rifratto $\boldsymbol{T}_0$, usiamo l'identità

$$\boldsymbol{T}_0 = \left[\eta < \boldsymbol{N}_1, \boldsymbol{V}_p > -\sqrt{1 - \eta^2(1 - < \boldsymbol{N}_1, \boldsymbol{V}_p >)}\right]\boldsymbol{N}_1 - \eta \boldsymbol{V}$$

dove $\eta = \frac{1}{\sqrt{2}}$. Si trova:

$$\left[\frac{1}{\sqrt{2}}(\frac{1}{\sqrt{3}}) - \sqrt{1 - \frac{1}{2}(1 - \frac{1}{3})}\right](0, 0, 1) - \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right) = \left(-\frac{1}{\sqrt{6}}, -\frac{1}{\sqrt{6}}, -\frac{2}{\sqrt{6}}\right)$$

Ecco il vedrsore del raggio d'ombra $\boldsymbol{V}_{\text{sorg}}$ diretto verso la sorgente di luce:

$$\boldsymbol{V}_{\text{sorg}} = \frac{\boldsymbol{l} - p}{\|\boldsymbol{l} - p\|} = \frac{(0, 0, 10) - (1, 1, 1)}{\|(-1, -1, 9\|} = (\frac{-1}{\sqrt{83}}, \frac{-1}{\sqrt{83}}, \frac{9}{\sqrt{83}})$$

Punto 1: $\boldsymbol{V}_{\text{sorg}}$ raggiunge o interseca qualche oggetto?
Troviamo la retta passante per $\boldsymbol{p}$ e con direzione $\boldsymbol{V}_{\text{sorg}}$.

$$\begin{cases} x = 1 - \frac{t}{\sqrt{83}} \\ y = 1 - \frac{t}{\sqrt{83}} \\ z = 1 + \frac{9t}{\sqrt{83}} \end{cases}$$

Mettiamo a sistema con il quadrato ($z = 3$) :

$$\begin{cases} x = 1 - \frac{t}{\sqrt{83}} \\ y = 1 - \frac{t}{\sqrt{83}} \\ 2 = \frac{9t}{\sqrt{83}} \end{cases}$$

quindi $\boldsymbol{p}_2 = (\frac{7}{9}, \frac{7}{9}, 3)$.
Il punto appartiene al quadrato perché il quadrato ha estensione $\{-1.4 \leqslant x \leqslant 1.4, -1.4 \leqslant y \leqslant 1.4\}$.
Pertanto il punto $\boldsymbol{p}$ è in ombra.
Punto 3: determiniamo il punto $\boldsymbol{q}$.

Troviamo la retta passante per $\boldsymbol{p}$ e con direzione $\boldsymbol{T}_0$.

$$\begin{cases} x = 1 - \frac{t}{\sqrt{6}} \\ y = 1 - \frac{t}{\sqrt{6}} \\ z = 1 - \frac{2t}{\sqrt{6}} \end{cases}$$

Mettiamo a sistema con la lastra speculare ($z = 0$) :

$$\begin{cases} x = 1 - \frac{t}{\sqrt{6}} \\ y = 1 - \frac{t}{\sqrt{6}} \\ \frac{1}{2} = \frac{t}{\sqrt{6}} \end{cases}$$

quindi $\boldsymbol{q} = (\frac{1}{2}, \frac{1}{2}, 0)$
Per trovare il versore riflesso $\boldsymbol{T}_1$:

$$\boldsymbol{T}_1 = 2 < \boldsymbol{N}_2, \boldsymbol{T}_q > \boldsymbol{N}_2 - \boldsymbol{T}_q$$

$$\boldsymbol{T}_q = \frac{p-q}{\|p-q\|} = \left(\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}\right)$$

$$\boldsymbol{N}_2 = (0, 0, 1)$$

$$(\boldsymbol{N}_2, \boldsymbol{T}_q) = \frac{2}{\sqrt{6}}$$

quindi

$$\boldsymbol{T}_1 = \frac{4}{\sqrt{6}}(0, 0, 1) - \left(\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}\right) = \left(-\frac{1}{\sqrt{6}}, -\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}\right)$$

Il punto $\boldsymbol{q}$ in è ombra?

$$\boldsymbol{L} = \frac{(\boldsymbol{l} - q)}{\|\boldsymbol{l} - q\|} = \frac{(0, 0, 10) - \left(\frac{1}{2}, \frac{1}{2}, 0\right)}{\sqrt{\frac{201}{2}}} = \frac{\left(-\frac{1}{2}, -\frac{1}{2}, 10\right)}{\sqrt{\frac{201}{2}}}$$

$$\begin{cases} x = \frac{1}{2} + t\frac{-2}{\sqrt{201}} \\ y = \frac{1}{2} + t\frac{-2}{\sqrt{201}} \\ z = 0 + t\frac{10\sqrt{2}}{\sqrt{201}} \end{cases}$$

Mettiamo a sistema con il quadrato ($z = 3$) :

$$\begin{cases} x = \frac{1}{2} + t\frac{-2}{\sqrt{201}} \\ y = \frac{1}{2} + t\frac{-2}{\sqrt{201}} \\ t = \frac{3\sqrt{201}}{10\sqrt{2}} \end{cases}$$

quindi $\boldsymbol{p}_L = \left(\frac{5\sqrt{2}-6}{10\sqrt{2}}, \frac{5\sqrt{2}-6}{10\sqrt{2}}, 3\right) = (0.15, 0.15, 3)$. Il punto appartiene al quadrato, che ha estensione $\{-1.4 \leqslant x \leqslant 1.4,\ -1.4 \leqslant y \leqslant 1.4\}$.

Pertanto il punto $\boldsymbol{q}$ è in ombra.
Punto 4: Troviamo il punto s dato dall'intersezione di $\boldsymbol{T}_1$ con la lastra trasparente.
Retta passante per $\boldsymbol{q}$ con direzione $\boldsymbol{T}_1$:

$$\begin{cases} x = \frac{1}{2} - \frac{t}{\sqrt{6}} \\ y = \frac{1}{2} - \frac{t}{\sqrt{6}} \\ z = \frac{2t}{\sqrt{6}} \end{cases}$$

Mettiamo a sistema con la lastra trasparente ($z = 1$) :

$$\begin{cases} x = \frac{1}{2} - \frac{t}{\sqrt{6}} \\ y = \frac{1}{2} - \frac{t}{\sqrt{6}} \\ \frac{1}{2} = \frac{t}{\sqrt{6}} \end{cases}$$

quindi $s = (0, 0, 1)$.
Troviamo il versore del raggio rifratto $\boldsymbol{T}_2$:

$$\boldsymbol{T}_2 = \left[\eta < \boldsymbol{N}_3, \boldsymbol{T}_s > -\sqrt{1 - \eta^2(1 - < \boldsymbol{N}_3, \boldsymbol{T}_s >)}\right]\boldsymbol{N}_3 - \eta \boldsymbol{T}_s$$

dove
$\eta = \frac{\sqrt{2}}{1}$
$\boldsymbol{N}_3 = (0, 0, -1)$

$$\boldsymbol{T}_s = \frac{q - s}{\|q - s\|} = \frac{\left(\frac{1}{2}, \frac{1}{2}, 0\right) - (0, 0, 1)}{\frac{\sqrt{3}}{\sqrt{2}}} = \left(\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}, -\frac{2}{\sqrt{6}}\right)$$

$$(\boldsymbol{N}_3, \boldsymbol{T}_s) = \frac{2}{\sqrt{6}}$$

Quindi

$$\begin{aligned} \boldsymbol{T}_2 &= \left[\sqrt{2}(\frac{2}{\sqrt{6}}) - \sqrt{1 - 2(1 - \frac{4}{6})}\right](0, 0, -1) - \sqrt{2}\left(\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}, -\frac{2}{\sqrt{6}}\right) \\ &= \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right) \end{aligned}$$

Calcoliamo la retta passante per s e direzione $\boldsymbol{T}_2$. La sua espressione parametrica è

$$\begin{cases} x = -\frac{t}{\sqrt{3}} \\ y = -\frac{t}{\sqrt{3}} \\ z = \frac{2t}{\sqrt{3}} \end{cases}$$

Mettiamo a sistema con il quadrato ($z = 3$): dalla terza equazione si ricava $t = \dfrac{3\sqrt{3}}{2}$, e quindi

$$\begin{cases} x = -\frac{3}{2} \\ y = -\frac{3}{2} \\ z = 3 \end{cases}$$

quindi il punto di intersezione è $n = (-\frac{3}{2}\ ,\ -\frac{3}{2}\ ,\ 3)$, che non appartiene al quadrato, che ha estensione $\{-1.4 \leqslant x \leqslant 1.4,\ -1.4 \leqslant y \leqslant 1.4\}$. La retta non interseca nemmeno la sfera (avente centro in (5,-5,5) e raggio 2).
Troviamo la retta che passa per s e si dirige verso la sorgente. Il vettore direzionale è

$$\boldsymbol{L}_2 = \frac{\boldsymbol{L} - s}{\|\boldsymbol{L} - s\|} = \frac{(0, 0, 10) - (0, 0, 1)}{9} = (0, 0, 1)$$

e quindi la retta, in forma parametrica, è

$$\begin{cases} x = 0 \\ y = 0 \\ z = 1 + t \end{cases}$$

Mettiamo a sistema con il quadrato ($z = 3$): la soluzione è

$$\begin{cases} x = 0 \\ y = 0 \\ z = 3 \end{cases}$$

(ed inoltre dalla terza equazione si ricava $t = 2$, ma questo non ci serve). Quindi $s_L = (0, 0, 3)$ é il punto in cui la retta incontra il piano del quadrato. Questo punto appartiene al quadrato.
Pertanto il punto $\boldsymbol{q}$ è in ombra. □

Esercizio 4.3.4. Nella scena dell'Esercizio 4.3.3 si sposti la sorgente di luce al punto $(0, 10, 10)$. Inoltre, si assuma che la lastra speculare abbia coefficiente di riflettività 1, esponente di Phong 3 e coefficiente di Lambert 0, la sfera sia rossa con coefficiente di Lambert ? e tutti gli altri coefficienti zero, il quadrato sia nero ed opaco, lo strato trasparente abbia coefficiente di Lambert 0, coefficiente di riflettività $1/2$, coefficiente di trasmissione $1/3$ ed esponente di Phong 1. Come cambiano le risposte?

Per comodità del lettore, ricapitoliamo il problema. L'osservatore è collocato nel punto $\boldsymbol{o} = (2, 2, 2)$. Una sorgente isotropa di luce è collocata nel punto $\boldsymbol{l} = (0, 10, 10)$. L'osservatore guarda verso l'origine. La scena si compone di una lastra metallica speculare sul piano $\{z = 0\}$, uno strato di materiale trasparente di coefficiente di rifrazione $\sqrt{2}$ che occupa la striscia $\{0 < z < 1\}$, un quadrato sul piano $\{z = 3\}$ di estensione $\{-1.5 \leqslant x \leqslant 1.5,\ -1.5 \leqslant y \leqslant 1.5\}$, ed una sfera con centro $\boldsymbol{C} = (5, -5, 5)$ e raggio 2. Si segua il raggio di osservazione, si determini il punto osservato $\boldsymbol{P}$ sulla lastra trasparente, e da tale punto si traccino:

- il raggio riflesso $\boldsymbol{R_P}$;
- il raggio rifratto $\boldsymbol{T_P}$;
- il raggio d'ombra $\boldsymbol{V}_{\text{sorg}}$ che si dirige verso la sorgente di luce.

Poi:

1. Per il il raggio d'ombra $\boldsymbol{V}_{\text{sorg}}$, si determini se esso raggiunge la sorgente di luce oppure se prima interseca qualche altro oggetto della scena. Il punto $\boldsymbol{P}$ è in luce o in ombra? Se è in luce si calcoli il relativo contributo di Lambert e di Phong.
2. Per il raggio riflesso $\boldsymbol{R_P}$, si determini se esso interseca qualche altro oggetto della scena oppure si perde nello sfondo. Nel primo caso si calcoli il contributo di Lambert e Phong.
3. Per il raggio rifratto $\boldsymbol{T_P}$, si trovi il punto $\boldsymbol{Q}$ in cui esso tocca la lastra speculare, e se ne calcoli l'eventuale contributo dalla illuminazione. Dal punto $\boldsymbol{Q}$ si tracci il suo raggio riflesso $\boldsymbol{r_q}$, e si tracci un nuovo raggio diretto verso la luce. Questo nuovo raggio raggiunge la luce oppure prima interseca qualche altro oggetto? Il punto $\boldsymbol{Q}$ è in luce o in ombra?
4. Si determini il punto $\boldsymbol{S}$ in cui il raggio $\boldsymbol{r_q}$ esce dalla lastra trasparente, e si segua il raggio rifratto $\boldsymbol{t_s}$ (che esce quindi dalla lastra ed entra nell'aria). Tale raggio colpisce qualche altro oggetto? Dal punto $\boldsymbol{S}$ si tracci un nuovo raggio d'ombra, ossia diretto verso la sorgente di luce. Tale raggio arriva alla sorgente di luce o viene fermato da qualche altro oggetto? Il punto $\boldsymbol{S}$ è in luce o in ombra?
5. Si determini l'illuminazione vista dall'osservatore sommando i contributi di questi raggi (ciascuno pesato con il corrispondente coefficiente di riflessine o di rifrazione al punto di origine del rispettivo raggio).

Svolgimento. Riassumiamo la geometria della scena. Essa consiste di:

- una lastra metallica speculare a $z = 0$ com coefficiente diffusivo $k_d = 0$, coefficiente speculare $k_s = 1$, esponente di Phong $n_l = 3$;
- uno strato trasparente in $\{0 \leqslant z \leqslant 1\}$ con coefficiente diffusivo $k_d = 0$, coefficiente speculare $k_r = 1/4$, esponente di Phong $n_s = 1$, coefficiente di trasmissione $k_t = 1/2$, coefficiente di rifrazione di Snell uguale a 2;
- una sfera rossa : $S = \{(x-5)^2 + (y+5)^2 + (z-5)^2 = 4\}$ con $k_s = 0$, $k_d = 1/4$;
- un quadrato nero opaco $Q = \{z = 3,\ -3/2 \leqslant x \leqslant 3/2,\ -3/2 \leqslant y \leqslant 3/2\}$ con $k_s = 0$ e $k_d = 1$.

Il punto di osservazione è $\boldsymbol{o} = (2, 2, 2)$. La direzione di osservazione è $\boldsymbol{w} = (-2, -2, -2)$. La sorgente di luce è in $\boldsymbol{L} = (0, 10, 10)$.

Determiniamo il punto osservato tracciando la retta parametrica nella direzione $\boldsymbol{w}$ passante per il punto $\boldsymbol{o}$:

$$\boldsymbol{r_o}(t) = \begin{cases} x = 2 - 2t, \\ y = 2 - 2t, \\ z = 2 - 2t. \end{cases}$$

Ora intersechiamo questa retta con lo strato trasparente, ovvero con il piano di equazione $\{z = 1\}$: la soluzione si ha per $t = 1/2$, ed il punto osservato è $\boldsymbol{P} = (1, 1, 1)$.
Verifichiamo se il punto $\boldsymbol{P}$ è in luce o in ombra. Dobbiamo tracciare la retta $\boldsymbol{r_p}$ che passa per il punto $\boldsymbol{P}$ e ha per direzione il vettore $\boldsymbol{l_0}$ che da $\boldsymbol{P}$ si dirige verso la sorgente di luce, ossia $\boldsymbol{l_0} = \boldsymbol{L} - \boldsymbol{P} = (0, 10, 10) - (1, 1, 1) = (-1, 9, 9)$.

$$\boldsymbol{r_p}(t) = \begin{cases} x = 1 - t, \\ y = 1 + 9t, \\ z = 1 + 9t. \end{cases}$$

Verifichiamo se la retta $\boldsymbol{r_p}$ interseca qualche oggetto della scena.
Essa non può intersecare la sfera poiché il suo vettore direzione ha coordinata y positiva mentre la sfera ha centro in $y = -5$ e raggio 2, quindi il massimo valore della coordinata y dei punti della sfera è $y = -3$. Verifichiamo quindi solo l'intersezione di $\boldsymbol{R_P}$ con il quadrato posto in $z = 3$. L'intersezione di $\boldsymbol{T_P}$ col piano $\{z = 3\}$ si ha per $t = 2/9$. ed è nel punto $(7/9, 3, 3)$. La coordinata y del punto di intersezione è fuori dal range del quadrato Q, quindi questo punto non appartiene al quadrato. Pertanto il punto $\boldsymbol{P}$ non è oscurato da nessun altro oggetto, quindi è in luce. Questo risponde alla domanda (1).
Calcoliamo il valore al punto $\boldsymbol{p}$ dell'illuminazione data dall'equazione di Phong. A questo scopo dobbiamo contare il contributo di Lambert, che è nullo perché la lastra speculare dove si trova il punto $\boldsymbol{P}$ ha coefficiente diffusivo uguale a zero, ed il contributo dato dal raggio riflesso e dal raggio rifratto. Determiniamo quindi il raggio riflesso ed il raggio rifratto.
In base a (7.7.1), il raggio riflesso del raggio proiettore proveniente dal punto di osservazione, ossia il versore riflesso *della direzione dell'osservatore* è

$$\boldsymbol{R_0} = 2\langle \boldsymbol{N_0}, \boldsymbol{V_0} \rangle - \boldsymbol{V_0},$$

dove $\boldsymbol{N_0} = (0, 0, 1)$ e $\boldsymbol{V_o}$ è il versore della direzione dell'osservatore visto dal punto $\boldsymbol{p}$,

$$\boldsymbol{V_0} = \frac{\boldsymbol{o} - \boldsymbol{P}}{\|\boldsymbol{o} - \boldsymbol{P}\|} = \frac{(1, 1, 1)}{\|(1, 1, 1)\|} = \frac{1}{\sqrt{3}}\,(1, 1, 1).$$

Si ricava quindi

$$\boldsymbol{R_0} = \frac{2}{\sqrt{3}}\,(0, 0, 1) - \frac{1}{\sqrt{3}}\,(1, 1, 1) = \frac{1}{\sqrt{3}}\,(-1, -1, 1).$$

Il versore rifratto $\boldsymbol{T_0}$ è dato dalla formula (7.7.2), che riscriviamo qui:

$$\boldsymbol{T_0} = \left(\eta_{\tau\lambda}\langle \boldsymbol{N_0}, \boldsymbol{I_0}\rangle - \sqrt{1-\eta_{\tau\lambda}^2(1-\langle \boldsymbol{N_0}, \boldsymbol{I_0}\rangle^2)}\right)\boldsymbol{N_0} - \eta_{\tau\lambda}\boldsymbol{I_0},$$

dove il versore incidente $\boldsymbol{I_0}$ ora ovviamente è l'opposto del versore del raggio di proiezione dall'osservatore, ossia $\boldsymbol{I_0} = -\boldsymbol{V_0} = -\frac{1}{\sqrt{3}}(1,1,1)$. Qui $\boldsymbol{N_0}$ e $\boldsymbol{V_0}$ sono i versori appena calcolati, ed il rapporto $\eta_{\tau\lambda}$ fra i coefficienti di rifrazione di Snell (che per semplicità assumiamo indipendente dalla frequenza della luce) vale $1/\sqrt{2}$ (perché il raggio esce dall'aria ed entra nel materiale trasparente). Poiché $\langle \boldsymbol{N_0}, \boldsymbol{I_0}\rangle = -1/\sqrt{3}$, applicando questa formula si trova

$$\boldsymbol{T_0} = \left(-\frac{1}{\sqrt{6}} - \sqrt{1-\frac{1}{2}\left(1-\frac{1}{3}\right)}\right)(0,0,1) - \frac{1}{\sqrt{6}}(1,1,1) \tag{4.3.3}$$

$$= -\frac{3}{\sqrt{6}}\left((0,0,1) - \frac{1}{\sqrt{6}}(1,1,1)\right) = \frac{1}{\sqrt{6}}(3,3,-4). \tag{4.3.4}$$

Calcoliamo ora il contributo di illuminazione al primo rimbalzo, in base all'equazione di Phong (2.4.3). Il colore di diffusione (e certamente anche quello di riflessione speculare) del materiale trasparente si può assumere neutro, quindi con coordinate RGB uguali a 1: ossia, quantizzando la frequenza in componenti R, G e B, possiamo scrivere

$$O_{d,\lambda} = O_{s,\lambda} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Pertanto (2.4.3) diventa

$$I_p = \left(k_d\langle \boldsymbol{N_0}, \boldsymbol{L_0}\rangle + k_s\langle \boldsymbol{T_0}, \boldsymbol{L_0}\rangle + k_t\langle \boldsymbol{N_0}, \boldsymbol{L_0}\rangle\right)\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Il primo addendo è il contributo diffusivo di Lambert: ma questo contributo è nullo perché si è posto $k_d = 0$. Il secondo addendo è il termine speculare di Phong, ma poiché la sorgente di luce è al punto $(0,10,10)$ si ha $\boldsymbol{L_0} = \frac{1}{\sqrt{200}}(0,10,10) = \frac{1}{\sqrt{2}}(0,1,1)$ e $\langle \boldsymbol{R_0}, \boldsymbol{L_0}\rangle = 0$, quindi anche questo contributo è nullo. Si noti che il fatto che il contributo di luce riflessa sia nullo non è dovuto a motivi fisici, ma solo al fatto che nel modello euristico di Phong dobbiamo trascurare gli angoli di osservazione che portano ad un lobo negativo del coseno dell'angolo di deviazione fra il versore rifratto del raggio di proiezione dell'osservatore ed il versore della luce, come osservato nella Sottosezione 2.4.2 subito prima dell'equazione (2.4.3).
L'ultimo addendo, $k_t\langle \boldsymbol{T_0}, \boldsymbol{L_0}\rangle$, è il contributo di Phong del raggio rifratto che entra nel materiale traslucido. Di solito questo contributo non si considera: comunque qui è certamente nullo, perché il versore rifratto $\boldsymbol{T_0}$ forma un angolo maggiore di $\pi/2$ con il versore diretto verso la sorgente di luce, dal momento che $\langle \boldsymbol{T_0}, \boldsymbol{L_0}\rangle = \frac{1}{\sqrt{12}}\langle (3,3,-4),(0,1,1)\rangle < 0$.
Quindi l'illuminazione diretta nel punto $\boldsymbol{P}$ è nulla, anche se per motivi legati alle inesattezze fisiche del modello euristico di Phong.

Vediamo ora la prima generazione ricorsiva, ossia calcoliamo i contributi di illuminazione dati dai punti colpiti rispettivamente dal raggio riflesso e del raggio rifratto.
Tracciamo la retta del raggio riflesso il punto $\boldsymbol{P}$, quindi con versore direzionale $\boldsymbol{R_o}$, e vediamo se interseca qualche oggetto nella scena. La sua equazione parametrica è

$$\boldsymbol{r_p}(t) = \begin{cases} x = 1-t, \\ y = 1-t, \\ z = 1+t. \end{cases}$$

Controlliamo se c'è intersezione con il quadrato Q che giace sul piano $\{z = 3\}$. La retta interseca questo piano quando $1 + t = 3$, ossia $t = 2$. Il punto di intersezione con il piano quindi è $(-1, -1, 3)$, che sta nel quadrato Q. Quindi c'è intersezione con il quadrato nero, ed allora non ci può essere intersezione con la sfera rossa perché il quadrato giace ne piano $\{z = 3\}$ ma la sfera giace nel semispazio $\{z > 3\}$. Dovremmo calcolare il contributo di illuminazione al punto di intersezione, ma, dato che il quadrato è opaco e nero, sappiamo a priori che il suo contributo è nullo. Questo risponde alla domanda (2).

Seguiamo ora il raggio rifratto e vediamo in quale punto colpisce la lastra speculare che giace sul piano $\{z = 0\}$. Scriviamo l'equazione parametrica del raggio rifratto: è la retta che passa per $\boldsymbol{P}$ in direzione del versore rifratto $\boldsymbol{T_0}$ calcolato in (4.3.3), ossia:

$$\boldsymbol{t_p}(t) = \begin{cases} x = 1 + 3t, \\ y = 1 + 3t, \\ z = 1 - 4t. \end{cases}$$

La lastra trasparente termina al piano $\{z = 0\}$, quindi il punto di intersezione $\boldsymbol{Q}$ si trova al parametro t tale che $1 - 4t = 0$, ossia $t = 1/4$. Pertanto $\boldsymbol{Q} = \frac{1}{4}(7, 7, 0)$. Controlliamo se il punto $\boldsymbol{Q}$ è in luce tracciando la retta passante per $\boldsymbol{Q}$ in direzione

$$\boldsymbol{l_q} = \boldsymbol{L} - \boldsymbol{Q} = \left(-\frac{7}{4},\, 10 - \frac{7}{4},\, 10\right) = \left(-\frac{7}{4},\, \frac{33}{4},\, 10\right),$$

e vediamo se tale retta interseca il quadrato. Si tratta della retta

$$\boldsymbol{V_Q}(t) = \begin{cases} x = \frac{7}{4}\,(1 - t), \\ y = \frac{1}{4}\,(7 + 33t), \\ z = 10t. \end{cases}$$

Questa retta interseca il piano $\{z = 3\}$ nel punto $(49/40, 169/40, 3)$, che appartiene al quadrato Q. Quindi il punto $\boldsymbol{Q}$ è in ombra. Qui, per semplicità stiamo tracciando il raggio d'ombra $\boldsymbol{V_Q}$ come se uscisse dritto dallo strato trasparente: il calcolo della direzione giusta del raggio d'ombra è un problema inverso complicato, cui si accennerà nella successiva Sezione 4.4. In realtà il raggio giusto ne deve uscire deflettendo dalla normale come stabilito dalla legge di Snell, ma, dal momento che il coefficiente di rifrazione di Snell è maggiore per lo strato trasparente che per l'aria, in effetti questo raggio, quando lascia lo strato ed entra nell'aria, deflette ancora di più dalla direzione normale. Quindi il raggio d'ombra giusto dovrebbe essere diretto più vicino alla direzione normale, e tramite una faticosa interpolazione si può vedere che intersecherebbe comunque Q. Questo risponde alla domanda (3).

Ora tracciamo il raggio riflesso della retta $\boldsymbol{T_P}$ dopo la riflessione in $\boldsymbol{Q}$. Il vettore direzionale della retta $\boldsymbol{T_P}$ è $(3, 3, -4)$, ed il suo vettore riflesso rispetto al versore normale a $\boldsymbol{Q}$, ossia rispetto a $(0, 0, 1)$, è $(3, 3, 4)$. Quindi il raggio riflesso è $\boldsymbol{r_q} = \frac{1}{4}(7, 7, 0) + t(3, 3, 4)$, ovvero

$$\boldsymbol{r_q}(t) = \begin{cases} x = \frac{7}{4} + 3t, \\ y = \frac{7}{4} + 3t, \\ z = 4t, \end{cases}$$

che interseca la faccia superiore dello strato trasparente, sul piano $\{z = 1\}$, per $t = 1/4$, ossia in $\boldsymbol{S} = (5/2,\, 5/2,\, 1)$. Vediamo se il punto $\boldsymbol{S}$ è in luce. La retta da $\boldsymbol{S}$ alla posizione della luce

$\boldsymbol{L} = (0, 10, 10)$ ha vettore di direzione $\boldsymbol{L} - \boldsymbol{S} = (-5/2,\ 15/2,\ 9)$ ed equazione parametrica

$$\begin{cases} x = \frac{5}{2}(1-t), \\ y = \frac{5}{2}(1+3t), \\ z = 1+9t, \end{cases}$$

ed interseca il piano $\{z = 3\}$ su cui giace il quadrato Q per $t = 2/9$, ossia nel punto $(35/18,\ 25/6,\ 3)$, che non appartiene al quadrato Q. Questa retta non può intersecare neppure la sfera rossa con centro $\boldsymbol{C} = (5, -5,\ 5)$ e raggio 2, perché l'estensione nella direzione y di questa sfera è $-7 \leqslant y \leqslant -3$, mentre la retta, nel semispazio $\{z > 0\}$, ossia per $t > 0$, ha coordinata $y = \frac{5}{2}(1+3t) > 0$. Quindi il punto $\boldsymbol{S}$ è in luce. Ma si trova sulla superficie dello strato trasparente, il cui coefficiente di Lambert è zero. Si deve quindi calcolare solo il contributo di Phong al punto $\boldsymbol{S}$, prendendo come direzione dell'osservatore quella dal punto $\boldsymbol{Q}$ (dove ha origine la riflessione) vista dal punto $\boldsymbol{S}$, ovvero $\boldsymbol{Q} - \boldsymbol{S} = -\frac{1}{2}(5, 5, 2) - \frac{1}{4}(7, 7, 0) = \frac{1}{4}(3, 3, 4)$. Si osservi che, come ovvio a priori senza bisogno di calcoli, la direzione dell'osservatore è precisamente l'opposto di quella speculare rispetto al piano $\{z = 0\}$ della direzione del raggio rifratto $\boldsymbol{T_P}$. Ora, chiaramente la direzione dell'osservatore e quella della sorgente vista dal punto $\boldsymbol{S}$ sono sui due lati opposti della superficie a $\{z = 1\}$, quindi non c'è contributo speculare di Phong riflessivo, ma potrebbe esserci un contributo speculare rifrattivo.
Sappiamo già che, quando il raggio $\boldsymbol{r_q}$ esce dallo strato trasparente ed entra nell'aria, il suo versore rifratto si ottiene riflettendo rispetto alla normale $(0, 0, 1)$ il versore incidente dato dal raggio di proiezione originale da $\boldsymbol{o}$ a $\boldsymbol{P}$. Questo versore incidente era $\boldsymbol{I_0} = -\boldsymbol{V_0} = -\frac{1}{\sqrt{3}}(1, 1, 1)$. Perciò il versore rifratto in uscita dallo strato trasparente è $\boldsymbol{T_s} = \frac{1}{\sqrt{3}}(-1, -1, 1)$.
D'altra parte, la direzione della luce vista da $\boldsymbol{S}$ è $\boldsymbol{L} - \boldsymbol{S} = (-5/2,\ 15/2,\ 9)$. Il versore normale della posizione della luce vista da $\boldsymbol{S}$ è quindi

$$\boldsymbol{L_s} = \sqrt{\frac{2}{287}}\,(-5/2,\ 15/2,\ 9).$$

Poiché l'esponente di Phong dello strato trasparente vale 1, il contributo di Phong del raggio rifratto è

$$\langle \boldsymbol{L_s}, \boldsymbol{T_s} \rangle = \frac{2}{287\sqrt{3}} \langle (-1, -1, 1), (-5/2,\ 15/2,\ 9) \rangle = \frac{8}{287\sqrt{3}}\ .$$

Questo contributo è positivo. Moltiplicandolo per il coefficiente k_t di trasmissione dallo strato trasparente all'aria si ottiene il valore dell'illuminazione (non occorre verificare se il contributo se si ha riflessione totale interna, il che annullerebbe questo contributo, perché non si ha riflessione totale, visto che abbiamo calcolato il versore rifratto ed esso punta verso l'esterno dello strato trasparente). Il raggio rifratto $\boldsymbol{T_S}$ ha direzione $\boldsymbol{T_s} = \frac{1}{\sqrt{3}}(-1, -1, 1)$, e la sua equazione parametrica è

$$\boldsymbol{T_S}(u) = \boldsymbol{S} + u\boldsymbol{T_S} = (\frac{5}{2}, \frac{5}{2}, 1) + \frac{u}{\sqrt{3}}(-1, -1, 1),$$

ossia, cambiando variabile $u \mapsto v = u/\sqrt{3}$,

$$\boldsymbol{T_S}(v) = \left(\frac{5}{2} - v, \frac{5}{2} - v, 1 + v\right).$$

Questo raggio interseca il piano $\{z = 3\}$ nel punto $(1/2, 1/2, 3)$, che appartiene al quadrato nero Q: quindi, di nuovo, esso non interseca la sfera rossa ed il contributo di illuminazione al punto di intersezione è zero. Questo completa la risposta alla domanda (4). Lasciamo al lettore il compito di sommare i risultati (ciascuno pesato con il corrispondente coefficiente di riflessine o di rifrazione al punto di origine del rispettivo raggio) e rispondere alla domanda (5). □

ESERCIZIO 4.3.5. Una scena si compone di una sfera con centro $(5, 0, 4)$ e raggio 1, bianca, con coefficiente di riflettività 0 e di Lambert 1, ed uno specchio ideale piano quadrato di lato 1 con centro l'origine e disposto nel piano che contiene l'asse y (in particolare l'origine) ed inclinato di un angolo θ rispetto all'asse x (ossia la sua normale forma un angolo θ con il versore $(1, 0, 0)$), che può variare nel range $\left[-\frac{\pi}{2}\ ,\ \frac{\pi}{2}\right]$. Il coefficiente di riflettività dello specchio è 1 e quello di Lambert 0; l'esponente di Phong è infinito perché lo specchio è ideale. Non c'è luce ambientale. C'è una sorgente di luce direzionale (ossia che emana raggi paralleli) disposta all'infinito nella direzione dell'asse x negativo, ossia a $(-\infty, 0, 0)$, di luce bianca di intensità 1. L'osservatore e' nel punto $(1, 0, 0)$ e guarda verso l'origine.
Per quale angolo l'osservatore vede la massima intensità di luce? Quanto vale questa intensità massima?

Svolgimento. Nel modello di Lambert l'intensità di luce ad un qualsiasi punto non dipende dalla posizione dell'osservatore, purché ovviamente quel punto sia visibile dall'osservatore. Invece, l'intensità di luce è data dal coseno dell'angolo di inclinazione dei raggi di luce rispetto alla normale alla superficie illuminata. Perciò il punto più luminoso della sfera è quello frontale rispetto alla luce, ossia il suo punto nella direzione della sorgente, in altre parole il suo polo nella direzione dell'asse x negativo. Si tratta del punto $(4, 0, 4)$, poichè il centro della sfera è in $(5, 0, 4)$ ed il raggio è 1. Notiamo che questo punto si trova nella bisettrice del piano xz. Dobbiamo quindi disporre lo specchio in modo tale che l'osservatore, che si trova sull'asse x, guardando verso l'origine (quindi verso l'asse x negativo) veda punti della bisettrice del piano xz. L'angolo fra l'osservatore e la bisettrice è di $\frac{\pi}{4}$: quindi l'inclinazione dello specchio deve essere della metà di questo angolo, ossia di $\frac{\pi}{8}$. □

ESERCIZIO 4.3.6. Una scena consiste di una sfera metallica completamente riflettente, con centro l'origine e raggio 1. L'osservatore è nel punto $(2, 0, 0)$, la sorgente di luce (di colore bianco) in $(0, 2, 0)$. In che punto della sfera l'osservatore vede riflessa la sorgente di luce? Si specifichi tale punto in coordinate cartesiane ed anche tramite gli angoli di Eulero. □

ESERCIZIO 4.3.7. La stessa scena dell'Esercizio precedente ora viene circondata da un cubo opaco con i lati paralleli agli assi, centrato nell'origine, di raggio 3, con coefficiente di riflettività ed esponente di Phong uguali a 1. La faccia alta è blu, le facce laterali sono verdi, la faccia bassa è rossa: tutti i colori sono puri alla massima luminosità. La sfera metallica è puramente speculare: non diffonde, riflette specularmente (si può porre il coefficiente di diffusione 0, l'esponente di Phong infinito ed il coefficiente di riflettività 1). Utilizzando il Ray Tracing ricorsivo a due generazioni, si determini:

- che colore l'osservatore vede riflesso se guarda in direzione dell'asse x verso la sfera;
- che colore l'osservatore vede riflesso se guarda verso il punto $(1/2, 1/2, 0)$.

□

ESERCIZIO 4.3.8. Un triangolo T nello spazio ha vertici in $(1, 1, 0)$, $(2, 2, 3)$, $(1, 2, u)$, dove u è un parametro reale. C'è una sorgente di luce in $\boldsymbol{L} = (-1, -1, 1)$. Il piano $\{z = 0\}$ è speculare. Un raggio di luce parte dal punto $\boldsymbol{l}$ e si muove verso l'origine: quando raggiunge l'origine si riflette, e la sua nuova direzione di movimento ovviamente si ottiene cambiando il segno della sola componente z della direzione che il raggio aveva prima della riflessione. Per qualche valore del parametro u, il raggio riflesso colpisce in triangolo T? Se sì, dove?

Svolgimento. . La direzione del raggio che va verso l'origine è data dal vettore $-\boldsymbol{L} = (1, 1, -1)$. Il raggio riflesso si propaga dall'origine in direzione $\boldsymbol{R} = (1, 1, 1)$. Il triangolo giace in un piano $ax + by + cz = d$, i cui coefficienti sono tali che il piano contiene i tre vertici di T, ossia soddisfano

le identità

$$\begin{aligned} a + b &= d, \\ 2a + 2b + 3c &= d, \\ a + 2b + uc &= d. \end{aligned}$$

Sostituendo la seconda identità nella prima troviamo $d + 3c = 0$; sostituendola nella terza abbiamo $b + uc = 0$. Quindi $c = -d/3$ e $b = ud/3$. Pertanto, dalla prima delle tre identità sopra, si ha $a = (1 - u/3)d$. Siamo liberi di dilatare dello stesso fattore tutti e quattro i coefficienti (questo non fa cambiare il piano), quindi prendiamo $d = 1$. Allora i coefficienti del piano di T sono $a = 1 - u/3, b = u/3, c = -1/3$. Quindi il piano ha equazione $(1 - u/3)x + u/3y - 1/3z = 1$ (per verifica, si controlli direttamente che i tre vertici soddisfano questa equazione).
La retta uscente dall?origine in direzione $\boldsymbol{R}$ ha equazioni cartesiane x= $y = z$ e quindi equazioni parametriche $x = t$, $y = t$, $z = t$. Osserviamo che $(1 - u/3) + u/3 - 1/3 = 2/3$. Quindi la retta interseca il piano quando il parametro t soddisfa $1 = (1 - u/3)t + u/3t - 1/3t = 2/3t$, ossia $t = 3/2$. Il punto di intersezione è $\boldsymbol{P} = (3/2, 3/2, 3/2)$. Il punto $\boldsymbol{P}$ giace dentro T o fuori?
Proiettiamo perpendicolarmente T sul piano $\{x, y\}$, ossia poniamo $z = 0$. Il punto $\boldsymbol{P}$ giace in T se e solo se la proiezione di $\boldsymbol{P}$ su questo piano giace nella proiezione di T (perché il triangolo è convesso). La proiezione di T è il triangolo nel piano con vertici in $(1, 1), (2, 2), (1, 2)$. La proiezione di $\boldsymbol{P}$ è il punto $(3/2, 3/2)$. Questo punto è esattamente il punto medio del segmento che congiunge i vertici $(1, 1)$ e $(2, 2)$, quindi il punto medio di uno dei lati della proiezione di T, pertanto esso giace nella proiezione di T. Concludiamo quindi che $\boldsymbol{P}$ giace in T. □

Esercizio 4.3.9. Un triangolo T nello spazio ha vertici in (1,0,0), (0,1,0), (0,0,1). Esso ha coefficiente di diffusione pari a 1/2 e coefficiente di riflessione di luce ambientale pari a 1/2, ma il coefficiente di riflessione speculare è 0; è colorato con un gradiente lineare che ha colore rosso puro nel vertice (1,0,0), verde puro in (0,1,0) e blu puro in (0,0,1). C'è una sorgente di luce in $(-1, -1, 1)$. Il piano $\{z = 0\}$ è una superficie bianca parzialmente speculare con coefficiente di riflessione speculare 4/5, coefficiente di diffusione di Lambert 1/5, coefficiente di riflessione per la luce ambientale 1/10 ed esponente di Phong 2. L'osservatore si trova nel punto $(-1, -1, 2)$ e guarda verso l'origine. La luce ambientale, di intensità 1, è bianca.

1. Che colore vede l'osservatore se si tiene conto solo della luce ambientale?
2. Che colore vede l'osservatore se nel calcolo si usa il Ray Tracing ricorsivo a due generazioni (due generazioni di raggi riflessi)?

Svolgimento. La direzione del raggio riflesso che si propaga dall'origine è, come nel problema precedente, $\boldsymbol{R} = (1, 1, 1)$. Invece ora il triangolo giace nel piano $x + y + z = 1$: il punto $\boldsymbol{P}$ di intersezione è quindi $\boldsymbol{P} = (1/3, 1/3, 1/3)$.
Ovviamente $\boldsymbol{P}$ giace dentro il triangolo, perché questo triangolo sottende l'intero ottante positivo. Osserviamo che i vertici del triangolo sono i tre vettori canonici di base $\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3$, ed il punto $\boldsymbol{P}$ è la loro combinazione convessa $(\boldsymbol{e}_1, +\boldsymbol{e}_2 + \boldsymbol{e}_)/3$ (in effetti, è il baricentro del triangolo). Osserviamo che la normale al triangolo T che punta verso il semipiano che contiene la sorgente di luce è la stessa in tutti i punti del triangolo e coincide con uno dei due versori normali al piano del triangolo, e precisamente con

$$\boldsymbol{N}_{\boldsymbol{P}} = \frac{1}{\sqrt{3}}\,(-1, -1, -1).$$

1. Se si tiene conto solo della luce ambientale e l?osservatore guarda verso l'origine, vede una superficie bianca con fattore di riflessone alla luce ambientale 1/10, e quindi vede un colore grigio di intensità 1/10.

2. Il contributo all?illuminazione dato dal raggio iniziale è 1/10 di grigio per la luce ambiente (come visto al punto precedente), più il contributo di Lambert, più il contributo di Phong. Poiché il coefficiente di diffusione del materiale del piano di base è 1/5 ed il versore L che individua la sorgente di luce è $(-1,-1,1)/\sqrt{3}$, il contributo di Lambert è grigio di intensità

$$\frac{1}{5}\,\langle \boldsymbol{L}, \boldsymbol{N}\rangle = \frac{1}{5\sqrt{3}} = 0.1156.$$

Come nel problema precedente, il versore riflesso è $\boldsymbol{R} = (1,1,1)/\sqrt{3}$, mentre il versore $\boldsymbol{V}$ che individua l?osservatore è il normalizzato di $(-1,-1,2)$, ossia $\boldsymbol{V} = (-1,-1,2)/\sqrt{6}$. Poiché il coefficiente di riflessione speculare è 0.8 e l?esponente di Phong 2, il contributo del termine di Phong per il raggio originale è grigio di intensità $0.8\ \langle \boldsymbol{R}, \boldsymbol{N}\rangle^2 = 0.8(\sqrt{2}/3)^2 = 1.6/9 = 0.1778$. Quindi il contributo totale del raggio originale è $1/10 + 0.1156 + 0.1778 = 0.3333$.

Per calcolare il contributo del raggio riflesso di prima generazione, dobbiamo considerare il punto $\boldsymbol{P}$ del triangolo T che esso colpisce ed il suo colore, e dobbiamo aggiungere per questo punto i contributi di Lambert e di Phong, attenuati tramite il coefficiente di riflettività del raggio riflesso, ossia il coefficiente di riflettività speculare del materiale che l?ha generato, che vale $4/5 = 0.8$. Si potrebbe aggiungere a questi due contributi anche quello dovuto alla luce ambientale, ma di norma la luce ambientale è considerata una luce di sottofondo tenue ed uguale per tutti i materiali, e quindi non la si aggiunge in maniera ricorsiva ai contributi speculari calcolati nelle generazioni successive.
Determiniamo anzitutto il colore del triangolo T al punto $\boldsymbol{P}$. Per linearità, il colore di $\boldsymbol{P}$, in ciascun canale, è la corrispondente combinazione convessa dei colori dei vertici. Quindi, nel canale rosso, il colore di $\boldsymbol{P}$ ha intensità 1/3, perché un vertice in questo canale ha intensità 1 e gli altri due hanno intensità zero. Analogamente nei canali verde e blu: quindi il colore di $\boldsymbol{P}$ è (1/3, 1/3, 1/3), ossia grigio di intensità 1/3.
Il coefficiente di diffusione di Lambert del triangolo T vale 1/2.
Rispetto al punto $\boldsymbol{P}$ la direzione della luce si ottiene normalizzando il vettore $\boldsymbol{L} - \boldsymbol{P}$: indichiamo con $\boldsymbol{L_P}$ il vettore normalizzato, che vale

$$\boldsymbol{L_P} = \frac{1}{\sqrt{51}}\,(-5,-5,-1).$$

Analogamente, la direzione dell'osservatore è il versore $\boldsymbol{V_P}$ ottenuto normalizzando $\boldsymbol{V} - \boldsymbol{P}$, e vale

$$\boldsymbol{V_P} = \frac{1}{\sqrt{51}}\,(-5,-5,1).$$

Ne segue che il contributo di Lambert del primo raggio riflesso generato (da moltiplicare poi per il fattore di riflessione speculare 0.8 della superficie sul piano $\{z = 0\}$ che ha generato tale raggio riflesso) è $1/2\langle \boldsymbol{L_P}, \boldsymbol{V_P}\rangle$ moltiplicato per il colore del punto $\boldsymbol{P}$. In ciascun canale tale colore ha intensità 1/3, e quindi il contributo di Lambert dovuto a tale raggio è

$$0.8\,\frac{1}{3}\,\frac{1}{2}\,\frac{1}{\sqrt{51}}\,(25+25-1) = \frac{2}{15}\,\frac{49}{\sqrt{51}} \approx 0.9148.$$

Poiché il coefficiente di riflessione speculare del triangolo vale 0, il primo raggio riflesso non produce alcun contributo di riflessione speculare (termine di Phong), ed inoltre esso non si riflette in un nuovo raggio riflesso. Ne segue che l?illuminazione totale del Ray Tracing ricorsivo a due generazioni vale $0.3333 + 0.9148 = 1.2481$ (naturalmente, dovremo poi riscalare questi valori di illuminazione dividendo per il massimo valore di illuminazione fra tutti i pixel).

□

ESERCIZIO 4.3.10. Si consideri una ciotola K di forma emisferica e di raggio 1, posata sull'origine, ossia tangente all'origine al piano $\{z = 0\}$ e contenuta nel semispazio superiore $\{z \geqslant 0\}$: quindi il bordo della ciotola è un cerchio J sul piano $\{z = 1\}$. Sia ϕ l'angolo di latitudine visto dal centro di K, orientato in modo che il bordo J corrisponda a $\phi = \pi/2$ ed il polo sud (ossia l'origine) a $\phi = 0$. La ciotola è colorata con un gradiente di rosso di intensità uguale a $\cos\phi$, ha coefficiente di diffusione di Lambert e di riflessione entrambi $1/2$, esponente di Phong 1 ed è riempita di acqua (coefficiente di rifrazione 1.33) fino a metà della sua altezza. Un osservatore sta nel punto $\boldsymbol{V} = (1, 1, 3/2)$ e guarda verso il punto (0,0,1/2). Una lampada puntiforme che emette luce bianca alla massima intensità in maniera isotropa è collocata al punto (0,0,1).

1. Se si pongono uguali a 0 il coefficiente di riflessione, di luce ambientale e di Lambert e ad 1 il coefficiente di trasmissione dell'acqua, che colore vede l'osservatore, nel modello del Ray Tracing con equazione di illuminazione di Phong? (Ovviamente, si tracci il raggio proiettore finché non tocca la ciotola, ma senza generare a quel punto ulteriori raggi riflessi e rifratti: però, per arrivare alla ciotola, il proiettore deve venire rifratto una volta).
2. Ora poniamo il coefficiente di riflessione della ciotola pari a $1/2$, quello dell'acqua pari a $1/10$ e quello di trasmissione dell'acqua $9/10$. Immaginiamo che la scena sia completata da uno sfondo nero non riflettente che la avvolge completamente, e che non ci sia luce ambientale. Rieseguiamo il calcolo con il Ray Tracing ricorsivo, con due generazioni di raggi. Come cambia la risposta?

Svolgimento. Per la parte (1), basta rammentare che il versore del raggio rifratto è

$$\boldsymbol{T} = (\eta_{\tau\lambda}\langle \boldsymbol{N}, \boldsymbol{I}\rangle - \sqrt{1 - \eta_{\tau\lambda}^2(1 - \langle \boldsymbol{N}, \boldsymbol{I}\rangle^2}\boldsymbol{N} - \eta_{\tau\lambda}\boldsymbol{I}, \tag{4.3.5}$$

dove ora $\eta_{\tau\lambda} = 1/1.33 = 3/4 = 0.75$, ed il versore incidente $\boldsymbol{I}$ vale, nel caso presente,

$$\boldsymbol{I} = \frac{1}{\sqrt{3}}\,(1, 1, 1).$$

Da questi dati si calcola subito il versore rifratto in (4.3.5):

$$\boldsymbol{T} = -\left(\frac{\sqrt{3}}{4}\,,\,\frac{\sqrt{3}}{4}\,,\,\frac{\sqrt{5}}{2\sqrt{2}}\right).$$

Il raggio proiettore colpisce la superficie dell?acqua al punto $\boldsymbol{Q} = (0, 0, 1/2)$, e da lì continua, senza perdita di energia, lungo la semiretta

$$\boldsymbol{R}(t) = \boldsymbol{Q} + t\boldsymbol{T} = \begin{cases} -\frac{\sqrt{3}}{4} \\ -\frac{\sqrt{3}}{4} \\ \frac{1}{2}\left(1 - \sqrt{\frac{5}{2}}\,t\right) \end{cases}$$

con $t > 0$.
L'emisfero ha equazione $x^2 + y^2 + (z - 1)^2 = 1$, con $z \geqslant 1$. Si vede facilmente che allora il punto di intersezione è dato dall'equazione

$$t^2 + \frac{\sqrt{5}}{2\sqrt{2}}\,t + \frac{1}{4} = 1,$$

la cui soluzione è con $t > 0$ è

$$t_0 = \frac{\sqrt{5}}{4\sqrt{2}} + \sqrt{\frac{5}{16} + \frac{3}{4}} \approx 0.6355.$$

Questo ci fornisce il punto di intersezione $\boldsymbol{R}(t_0) \approx (-0.2752, -0.2752, 0.1847)$. L'altezza dell'intersezione è quindi circa il 18.5% dell?altezza totale dell'emisfero, ossia una discesa rispetto al centro della

sfera di circa 81.5%. Questo corrisponde ad un angolo ϕ con coseno uguale a 0.185. Poiché l'emisfero è colorato con una sfumatura di rosso data da $\cos\phi$, l'intensità del colore al punto osservato è 0.185 del rosso puro, ossia rosso al 18.5% di luminosità. A questo dato bisogna applicare l'equazione di illuminazione di Phong, ricordando però che dopo la generazione del raggio rifratto il calcolo va fatto come se l'osservatore si spostasse nel punto dove viene generato tale raggio, quindi in (0,0,1/2): in altre parole, la direzione del versore $\boldsymbol{V}$ dell'osservatore, per il calcolo dell'equazione di Phong al punto $\boldsymbol{R}(t_0)$, è l'opposto del nuovo vettore incidente, ossia

$$\boldsymbol{V} = \left(\frac{\sqrt{3}}{4}, \frac{\sqrt{3}}{4}, \frac{\sqrt{5}}{2\sqrt{2}}\right) = (0.4330,\ 0.4330,\ 0.7906).$$

Inoltre, la luce è collocata nel centro dell'emisfero, e quindi il versore della direzione della luce coincide con il versore radiale $\boldsymbol{N}$, che è proporzionale a $\boldsymbol{A} = (0,0,1) - \boldsymbol{R}(t_0) = (0.2752,\ 0.2752,\ 0.8153)$. Si ha quindi $\|A\| = 0.903429$, e normalizzando si trova

$$\boldsymbol{N} = \boldsymbol{L} = (0.3046,\ 0.3046,\ 0.9025).$$

□

4.4. Malfunzionamenti inerenti al Ray Tracing ricorsivo

Il Ray Tracing ricorsivo è molto sensibile agli errori di arrotondamento. Infatti le coordinate del punto in cui il raggio principale interseca un oggetto sono usate come origine di un nuovo raggio d'ombra, che quindi in quel punto ha parametro t con valore 0. Se un arrotondamento numerico dà a t un valore piccolo ma negativo, l'algoritmo di Ray Tracing ricorsivo deduce che l'oggetto interferisce col nuovo raggio d'ombra, oscurando sé stesso (cioè proiettando ombra su sé stesso). In tal caso vengono generate ombre sbagliate sull'oggetto. Non si può risolvere il problema evitando di eseguire il test di intersezione fra un oggetto e i raggi di ombra da esso generati, perchè ci sono oggetti che possono proiettare davvero ombre su sé stessi: gli oggetti concavi. Bisogna invece predefinire una soglia di tolleranza ed eseguire il test di intersezione solo per i valori di t inferiori a quella soglia (in valore assoluto).

La Figura 4.4.1 illustra un problema del Ray Tracing ricorsivo dovuto alla rifrazione.

Il raggio d'ombra $\boldsymbol{L_3}$ (generato quando il raggio di luce $\boldsymbol{T_1}$ che sta attraversando l'oggetto solido semitrasparente sulla destra esce da questo oggetto) viene inviato in direzione della sorgente di luce. Però questo raggio esce anch'esso dal solido semitrasparente prima di arrivare alla sorgente. Nella realtà (cioè nel modello fisico della visibilità della sorgente) all'uscita dall'oggetto anche il raggio $\boldsymbol{L_3}$ dovrebbe subire una deviazione dovuta alla rifrazione. Il metodo ignora questa deviazione perchè al momento in cui $\boldsymbol{L_3}$ viene generato si conosce la direzione della sorgente di luce, ma non la direzione verso cui inviare il raggio per farlo arrivare alla sorgente di luce tenendo conto della successiva rifrazione all'uscita del solido (questa successione di raggio d'ombra interno al solido nella direzione giusta seguito da raggio rifratto esterno che colpisce esattamente la sorgente di luce è illustrata in rosso nella Figura). Quest'ultima avviene con un angolo che dipende dall'angolo che il raggio forma con la frontiera a quel punto. Per calcolarla esattamente si potrebbero usare alcuni accorgimenti sul campionamento come inviare non un raggio d'ombra, ma due raggi, tracciarli entrambi ed interpolare o procedere per bisezioni successive dell'angolo fra i due raggi (questo è un procedimento possibile ma oneroso) o sull'algoritmo stesso. Invece si utilizza un Ray Tracing a ritroso, cioè a partire dalle sorgenti luminose.

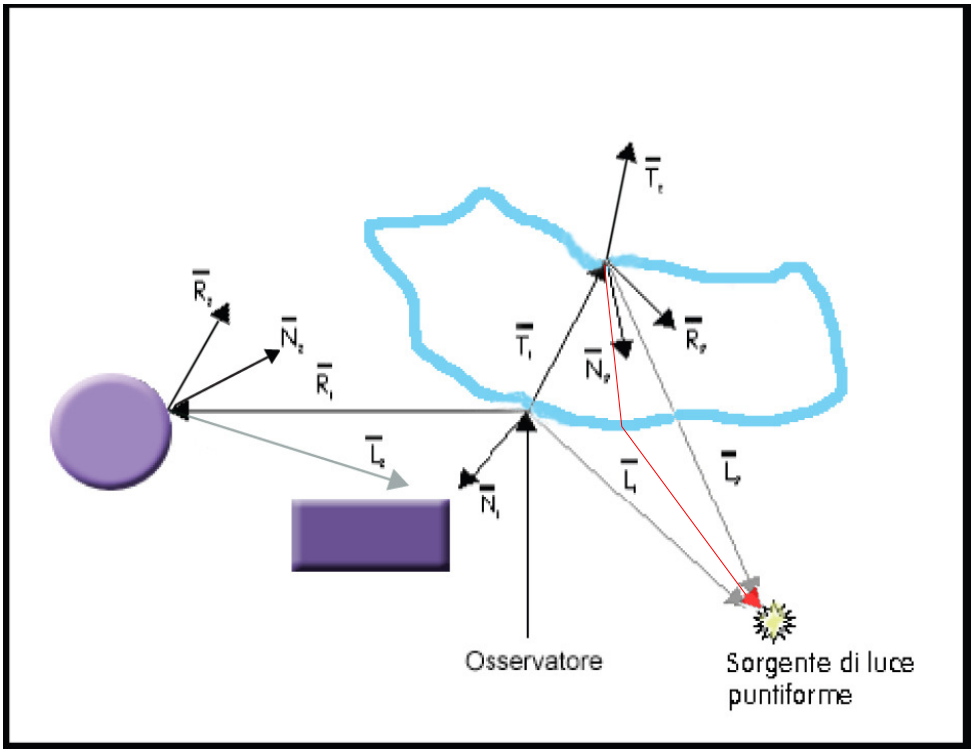


FIGURA 4.4.1. Il raggio d'ombra in colore rosso è quello diretto verso la sorgente di luce seguendo il percorso vero della luce nel corso delle rifrazioni, non il percorso rettilineo

CAPITOLO 5

Uso di mappe per accelerare il Ray Tracing

5.1. Mappa di rilievo

La mappa di rilievo [**5**] è un metodo per simulare il rilievo di una superficie, ad esempio la sua rugosità. È analoga alla mappa di tessitura, ma le informazioni mappate non rappresentano colore bensì profondità. Una mappa di tessitura che proietta sulla superficie una immagine di rugosità non sarebbe adeguata per il rendering, perché la rugosità modifica la direzione del vettore normale punto per punto, e quindi produce un effetto di ombreggiatura che varia da punto a punto con la posizione della sorgente di luce e dell'osservatore: invece una tessitura proiettata sulla superficie mostrerebbe la colorazione indotta dall'immagine proiettata, che non varia più quale che sia il punto osservato quando si spostano luce o osservatore. Inoltre, la direzione da cui proviene la luce nell'immagine proiettata non è generalmente la stessa che nella scena tridimensionale, e questo genererebbe una aberrazione visibile.
Si deve quindi risolvere il problema di calcolare la direzione del versore normale, o almeno approssimarla, dopo aver applicato la mappa di rilievo.

Una superficie tridimensionale è parametrizzata da due parametri reali s, t. La superficie di cui stiamo facendo il rendering sta nella scena tridimensionale: lo spazio tridimensionale ha coordinate x, y, z. Individuiamo un punto $\boldsymbol{p}$ sulla superficie con tre mappe, dalle coordinate s, t alle coordinate x, y, z. Quindi scriviamo $\boldsymbol{p} = \big(x(s,t),\, y(s,t),\, z(s,t)\big)$. Assumiamo che questa parametrizzazione sia di classe C^2, ossia derivabile due volte con continuità rispetto ad entrambe le variabili, ed in particolare differenziabile (ovvero deve esistere il piano tangente a ciascun punto). Supponiamo di tenere fisso $t = t_0$ e far variare s. Allora $\boldsymbol{p}$ si muove lungo una curva sulla superficie che si chiama curva di livello $t = t_0$. Fissato un qualsiasi valore di s, diciamo $s = s_0$, il vettore tangente a questa curva di livello al punto $(s_0,\, t_0)$ è la derivata parziale $\frac{\partial \boldsymbol{p}}{\partial s}\,(s_0\,,\, t_0)$, che per semplicità denotiamo con $D_1\boldsymbol{p}(s_0, t_0)$. Analogamente, per lo stesso punto di parametri $(s_0\,,\, t_0)$ della superficie passa una curva di livello $s =$ costante, in questo caso $s = s_0$: si tratta della curva parametrizzata da t variabile e s fisso $(s = s_0)$, il cui vettore tangente a $(s_0\,,\, t_0)$ è $D_2\boldsymbol{p}(s_0\,,\, t_0)$. Allora un vettore normale $\boldsymbol{n}$ alla superficie al punto di parametri $(s_0\,,\, t_0)$ è il prodotto vettore dei due vettori tangenti, poiché deve essere ortogonale ad entrambi:

$$\boldsymbol{n} = D_1\boldsymbol{p}(s_0\,,\, t_0) \times D_2\boldsymbol{p}(s_0\,,\, t_0) \tag{5.1.1}$$

(che in altri riferimenti bibliografici è indicato con il simbolo $\wedge$ invece che $\times$).

Assumiamo che il prodotto vettore a secondo membro di (5.1.1) non si annulli (questo equivale a richiedere che lo Jacobiano della mappa $\boldsymbol{p} : \mathbb{R}^2 \mapsto \mathbb{R}^3$ sia non nullo a ciascun punto $(s_0\,,\, t_0)$, ossia che la rappresentazione parametrica sia localmente invertibile; osserviamo che nell'esempio che stiamo per dare, quello della parametrizzazione di una sfera tramite gli angoli di Eulero (latitudine e longitudine) questa ipotesi vale ovunque tranne che ai poli Nord e Sud, e quindi questi due punti devono essere omessi dalla parametrizzazione). Il versore normale ovviamente è $\boldsymbol{n}/\|\boldsymbol{n}\|$.

Ora introduciamo una mappa di rilievo: ad ogni punto (s,t) nel dominio dei parametri il punto corrispondente sulla superficie viene elevato, cioè spostato perpendicolarmente al piano tangente, ossia nella direzione del versore normale, di uno spostamento di grandezza $b(s,t)$. La funzione b si

chiama appunto mappa di rilievo. Perciò la nuova posizione della superficie ai parametri $(s_0\,,\,t_0)$ è

$$\boldsymbol{p'}(s_0\,,\,t_0) = \boldsymbol{p}(s_0\,,\,t_0) + b(s_0\,,\,t_0)\,\frac{\boldsymbol{n}(s_0\,,\,t_0)}{\|\boldsymbol{n}(s_0\,,\,t_0)\|}\,. \tag{5.1.2}$$

Ora è necessario calcolare il nuovo versore normale per utilizzarlo nella equazione dell'illuminazione. Quale è il suo valore? Una buona approssimazione si ottiene grazie allo sviluppo di Taylor del primo ordine (ossia al teorema del valor medio di Lagrange) dopo aver sostituito l'identità (5.1.2) in (5.1.1). Il prossimo Corollario stabilisce questo valore.

Corollario 5.1.1 (Formula di Blinn).

$$\boldsymbol{n'} \approx \boldsymbol{n} + \frac{D_1 b \cdot (\boldsymbol{n} \times D_2\boldsymbol{p}) - D_2 b \cdot (\boldsymbol{n} \times D_1\boldsymbol{p})}{\|\boldsymbol{n}\|}$$

(nel senso del primo ordine di sviluppo di Taylor rispetto alle vafiabili s e t) purché la mappa di rilievo b sia piccola e cambi lentamente, ossia b, D_1b e D_2b siano uniformemente piccole.

Dimostrazione. Derivando (5.1.2) rispetto a s e t otteniamo:

$$D_1\boldsymbol{p'}(s_0\,,\,t_0) = D_1\boldsymbol{p}(s_0\,,\,t_0) + D_1 b(s_0\,,\,t_0)\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|} + b(s_0\,,\,t_0)D_1\left(\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}\right),$$

$$D_2\boldsymbol{p'}(s_0\,,\,t_0) = D_2\boldsymbol{p}(s_0\,,\,t_0) + D_2 b(s_0\,,\,t_0)\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|} + b(s_0\,,\,t_0)D_2\left(\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}\right).$$

Poiché la parametrizzazione è di classe C^2, il vettore $\boldsymbol{n} = D_1\boldsymbol{p}(s_0\,,\,t_0) \times D_2\boldsymbol{p}(s_0\,,\,t_0)$ è derivabile con continuità rispetto alle variabili s e t, e quindi lo stesso succede per $\boldsymbol{n}/\|\boldsymbol{n}\|$ perché, per ipotesi, il denominatore non si annulla. Questo vettore giace nella sfera unitaria, la quale è compatta. Pertanto le derivate $D_1(\boldsymbol{n}/\|\boldsymbol{n}\|)$ e $D_2(\boldsymbol{n}/\|\boldsymbol{n}\|)$ sono continue su un compatto, quindi limitate. Inoltre, naturalmente, lo spostamento b deve essere piccolo. Combinando queste due osservazioni vediamo che l'ultimo termine al lato di destra delle equazioni precedenti può essere trascurato. Pertanto

$$\begin{aligned}
\boldsymbol{n'}(s_0\,,\,t_0) &= D_1\boldsymbol{p'}(s_0\,,\,t_0) \times D_2\boldsymbol{p'}(s_0\,,\,t_0) \\
&= D_1\boldsymbol{p}(s_0\,,\,t_0) \times D_2\boldsymbol{p}(s_0\,,\,t_0) \\
&+ D_1 b(s_0\,,\,t_0)\frac{\boldsymbol{n}}{\|\boldsymbol{n}\|} \times D_2\boldsymbol{p}(s_0\,,\,t_0) \\
&+ D_2 b(s_0\,,\,t_0)D_1\boldsymbol{p}(s_0\,,\,t_0) \times \frac{\boldsymbol{n}}{\|\boldsymbol{n}\|} \\
&+ D_1 b(s_0\,,\,t_0)D_2 b(s_0\,,\,t_0)\frac{\boldsymbol{n} \times \boldsymbol{n}}{\|n\|^2}\,.
\end{aligned}$$

Ovviamente l'ultimo termine sul lato destro si annulla, ed in base a (5.1.2) il primo vale $\boldsymbol{n}$. Quindi

$$\begin{aligned}
\boldsymbol{n'} - \boldsymbol{n} &= \frac{D_1 b \cdot (\boldsymbol{n} \times D_2\boldsymbol{p}) + D_2 b \cdot (D_1\boldsymbol{p} \times \boldsymbol{n})}{\|\boldsymbol{n}\|} \\
&= \frac{D_1 b \cdot (\boldsymbol{n} \times D_2\boldsymbol{p}) - D_2 b \cdot (\boldsymbol{n} \times D_1\boldsymbol{p})]}{\|\boldsymbol{n}\|}\,.
\end{aligned}$$

□

Nota 5.1.2. *Nella identità* (5.1.2) *e nella formula di Blinn del Corollario 5.1.1 tutti i vettori sono calcolati allo stesso punto della superficie, parametrizzato da* $(s_0\,,\,t_0)$. *Abbiamo implicitamente*

supposto che la mappa di rilievo b sia parametrizzata dalle stesse coordinate s, t con le quali abbiamo parametrizzato la superficie. In realtà la mappa di rilievo di solito usa sue coordinate specifiche u, v, come accade per la mappa di tessitura: b viene memorizzata e richiamata come una immagine rettangolare con proprie coordinate (ed in tal modo può essere usata come tessitura per diverse superficie della scena, o per scene diverse). Quando b viene mappata sulla superficie, viene stabilita una corrispondenza biunivoca fra le coordinate intrinseche u, v di b e s, t della superficie: cioè si definisce una mappa biunivoca

$$u = u(s,t), \qquad v = v(s,t). \tag{5.1.3}$$

Implicitamente, quindi, nel supporre che b nella formula di Blinn sia parametrizzato dalle stesse coordinate s, t della superficie equivale ad identificare le sue coordinate u, v con s, t tramite (5.1.3)*. Pertanto il modo esplicito di scrivere la formula di Blinn è:*

$$\begin{aligned}
\boldsymbol{n'}(s_0\,,\,t_0) &= \boldsymbol{n}(s_0\,,\,t_0) \\
&\quad + \frac{D_1 b(u(s_0\,,\,t_0),\,v(s_0\,,\,t_0))(\boldsymbol{n}(s_0\,,\,t_0) \times D_2\boldsymbol{p}(s_0\,,\,t_0))}{\|\boldsymbol{n}(s_0\,,\,t_0)\|} \\
&\quad - \frac{D_2 b(u(s_0\,,\,t_0),\,v(s_0\,,\,t_0))\boldsymbol{n}(s_0\,,\,t_0) \times D_1\boldsymbol{p}(s_0\,,\,t_0)}{\|\boldsymbol{n}(s_0\,,\,t_0)\|} \\
&= \boldsymbol{n}(s_0\,,\,t_0) \\
&\quad + \frac{D_1 b(u(s_0\,,\,t_0),\,v(s_0\,,\,t_0))(\boldsymbol{n}(s_0\,,\,t_0) \times D_2\boldsymbol{p}(s_0\,,\,t_0))}{\|\boldsymbol{n}(s_0\,,\,t_0)\|} \\
&\quad - \frac{D_2 b(u(s_0\,,\,t_0),\,v(s_0\,,\,t_0))(\boldsymbol{n}(s_0\,,\,t_0) \times D_1\boldsymbol{p}(s_0\,,\,t_0))}{\|\boldsymbol{n}(s_0\,,\,t_0)\|}\,.
\end{aligned} \tag{5.1.4}$$

5.2. Richiami su parametrizzazione di superficie, integrali di superficie ed aree

Facondo uso della notazione della precedente Sezione 5.1, vogliamo ora esprimere il calcolo dell'area di una superficie $\boldsymbol{p}(s,t)$ in termini di un fattore locale di ingrandimento delle aree sotto la mappa $\boldsymbol{p}$ che descrive la superficie. In maniera analoga si definisce, più in generale, l'integrale di una funzione (a valori scalari o anche vettoriali) lungo la superficie.

La norma del vettore normale $\boldsymbol{n} = D_1\boldsymbol{p}(s_0\,,\,t_0) \times D_2\boldsymbol{p}(s_0\,,\,t_0)$ è data da $\|D_1\boldsymbol{p}\|\,\|D_2\boldsymbol{p}\|\,|\sin\theta|$, dove θ è l'angolo formato dai due vettori tangenti $D_1\boldsymbol{p}$ e $D_2\boldsymbol{p}$. Questi due vettori sono rispettivamente tangenti alle curve s =costante e t =costante sulla superficie parametrica $\boldsymbol{p}(s,t) = (p_1 = p_1(s,t), p_2 = p_2(s,t), p_3 = p_3(s,t))$. Le lunghezze di questi due vettori tangenti misurano le velocità di percorrenza delle curve s =costante e t =costante data dalla parametrizzazione scelta. Grazie al fattore $|\sin\theta|$, quindi, la norma dello Jacobiano $\|D_1\boldsymbol{p}(s_0\,,\,t_0) \times D_2\boldsymbol{p}(s_0\,,\,t_0)\|$ è esattamente l'area del parallelogramma generato dai due vettori tangenti $D_1\boldsymbol{p}$ e $D_2\boldsymbol{p}$, e pertanto è il fattore locale (al punto $(s_0\,,\,t_0)$) di ingrandimento delle aree dato dalla parametrizzazione. In altre parole, se $\boldsymbol{p}$ è definita su un rettangolo K, l'area della superficie parametrica $\boldsymbol{p}(K)$ è data dall'integrale

$$\iint_K \|D_1\boldsymbol{p}(s,t) \times D_2\boldsymbol{p}(s,t)\|\,ds\,dt\,.$$

Nel caso particolare della parametrizzazione semplice data dal grafico di una funzione f delle variabili s e t, ossia $\boldsymbol{p}(s,t) = (s,t,f(s,t))$, il fattore di ingrandimento delle aree si calcola facilmente. Infatti, come abbiamo visto, esso è dato dal prodotto delle norme dei vettori velocità di percorrenza delle curve s = costante e t = costante per il seno dell'angolo che questi due vettori formano.

Ma per la scelta della parametrizzazione, ossia $\boldsymbol{p}(s,t) = (s,t,f(s,t))$, i due vettori velocità sono rispettivamente

$$D_1\boldsymbol{p}(s,t) = \left(1, 0, \frac{\partial f}{\partial s}(s,t)\right)$$

e

$$D_2\boldsymbol{p}(s,t) = \left(0, 1, \frac{\partial f}{\partial t}(s,t)\right).$$

Pertanto

$$\|(D_1\boldsymbol{p}(s,t) \times D_2\boldsymbol{p})(s,t)\| = \sqrt{1 + \left(\frac{\partial f}{\partial s}\right)^2 + \left(\frac{\partial f}{\partial t}\right)^2}. \tag{5.2.1}$$

ESEMPIO 5.2.1. *(Area della calotta sferica).* Consideriamo la calotta sferica con disco di base di raggio r_0 nella superficie sferica di raggio R, ossia il grafico della funzione

$$f(s,t) = \sqrt{R^2 - s^2 - t^2} \qquad \text{nel disco} \quad D = \{s^2 + t^2 \leqslant r_0^2\}.$$

La formula precedente ora porta a

$$\|(D_1\boldsymbol{p}(s,t) \times D_2\boldsymbol{p})(s,t)\| = \sqrt{1 + \frac{s^2+t^2}{R^2-s^2-t^2}} = \sqrt{\frac{1}{1 - \left(\frac{s}{R}\right)^2 - \left(\frac{t}{R}\right)^2}}\,.$$

Se si passa in coordinate polari $s = r\cos\theta$, $t = r\sin\theta$, l'ultima espressione diventa

$$\begin{aligned}\sqrt{\frac{1}{1 - \left(\frac{s}{R}\right)^2 - \left(\frac{t}{R}\right)^2}} &= \sqrt{\frac{1}{1 - \left(\frac{r}{R}\right)^2\cos^2\theta - \left(\frac{r}{R}\right)^2\sin^2\theta}} \\ &= \sqrt{\frac{1}{1 - \left(\frac{r}{R}\right)^2}} = \sqrt{\frac{R^2}{R^2 - r^2}}\,.\end{aligned}$$

Pertanto l'area della calotta sferica è

$$\begin{aligned}\int_0^{2\pi}\int_0^{r_0} \sqrt{\frac{R^2}{R^2-r^2}}\, r\, dr\, d\theta &= \pi \int_0^{r_0^2} \sqrt{\frac{R^2}{R^2-u}}\, du \\ &= \pi R^2 \int_0^{r_0^2/R^2} \sqrt{\frac{1}{1-v}}\, dv \\ &= 2\pi R^2 \sqrt{1-v}\Big|_{r_0^2/R^2}^0 = 2\pi R^2 \left(1 - \sqrt{1 - \frac{r_0^2}{R^2}}\right).\end{aligned}$$

□

ESEMPIO 5.2.2. *(Area del cono retto).* Consideriamo il cono retto con altezza h e disco di base di raggio R, ossia il grafico della funzione

$$f(s,t) = h\left(1 - \sqrt{\frac{s^2}{R^2} + \frac{t^2}{R^2}}\right) \qquad \text{nel disco} \quad D = \{s^2 + t^2 \leqslant R^2\}.$$

Ripetendo i calcoli precedenti ora si ottiene il fattore di ingrandimento seguente:

$$\|(D_1\boldsymbol{p} \times D_2\boldsymbol{p})(s,t)|| = \sqrt{1 + \frac{h^2}{R^2}\,\left(\frac{s^2}{s^2+t^2} + \frac{t^2}{s^2+t^2}\right)} = \sqrt{1 + \frac{h^2}{R^2}}\,.$$

Quindi l'area del cono è

$$\sqrt{1+\frac{h^2}{R^2}}\int_0^{2\pi}\int_0^R r\,dr\,d\theta = \sqrt{1+\frac{h^2}{R^2}}\,\text{area}\,(D) = \pi R^2\sqrt{1+\frac{h^2}{R^2}}$$
$$= \pi R\sqrt{R^2+h^2} = \pi RA,$$

dove abbiamo denotato con A la distanza dal vertice del cono ai punti della circonferenza di base (che, in base al teorema di Pitagora, è proprio $\sqrt{R^2+h^2}$). □

ESEMPIO 5.2.3. *(La superficie sferica e la parametrizzazione con gli angoli di latitudine e longitudine.* Così come la circonferenza con centro l'origine e raggio fissato si parametrizza con un angolo, la sfera

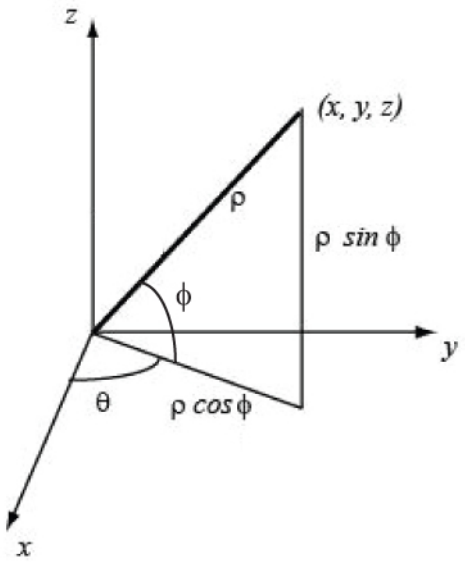


FIGURA 5.2.1. Angoli di latitudine e longitudine, ossia angoli di Eulero modificati in modo che il polo Nord abbia latitudine $+\pi/2$ ed il polo Sud abbia latitudine $-\pi/2$

con centro l'origine e raggio r si può parametrizzare tramite una coppia di angoli, detti angoli di Eulero. Nella versione provvisoria che scegliamo adesso, e che modificheremo a partire dal prossimo Esercizio 5.2.4, la parametrizzazione di Eulero ricalca la descrizione angolare della sfera in termini degli angoli di latitudine e di longitudine determinati dalla scelta di un asse polare: in particolare, la mappa $\boldsymbol{p} = (x(\theta,\phi),\, y(\theta,\phi),\, z(\theta,\phi))$ è data da

$$x = r\cos\theta\,\cos\phi, \qquad y = r\sin\theta\cos\phi, \qquad z = r\sin\phi, \tag{5.2.2}$$

dove r è il raggio della sfera, ϕ varia in $[-\pi/2, \pi/2]$ ed è la latitudine, e θ varia in $[0, 2\pi]$ ed è la longitudine. Si osservi che questa parametrizzazione non è biunivoca al polo nord ed al polo sud, cioè rispettivamente per $\phi = \pi/2$ e $\phi = -\pi/2$. In effetti, ciascuno dei paralleli dati da $\phi = \pm\pi/2$ si condensa in un unico punto, il polo nord o il polo sud, e quindi le curve di livello $\phi = \pi/2$ e $\phi = -\pi/2$ sono costanti. Perciò $D_1\boldsymbol{p}(\theta, -\pi/2) = 0 = D_1\boldsymbol{p}(\theta, \pi/2)$ per ogni θ, e quindi, in base a (5.1.1), $\boldsymbol{n}(\theta, -\pi/2)$ e $\boldsymbol{n}(\theta, \pi/2)$ sono nulli. Ovviamente i versori normali ai poli nord e sud della superficie sferica non sono affatto nulli (sono i due versori verticali), ma in questa parametrizzazione il calcolo dà risultato nullo perché la parametrizzazione non è biunivoca in corrispondenza dei poli,

e quindi non è una parametrizzazione ammissibile.
In tutti gli altri punti, invece, la parametrizzazione è biunivoca. □

ESERCIZIO 5.2.4. Si determini il versore normale alla sfera tramite la parametrizzazione con gli angoli di Eulero e si mostri che coincide con il versore radiale.

Svolgimento. Calcoliamo $D_1\boldsymbol{p}$ e $D_2\boldsymbol{p}$. Indicando con $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ i versori canonici dei tre assi, otteniamo

$$\begin{aligned} D_1\boldsymbol{p}(\theta,\phi) &= (-r\sin\theta\cos\phi, r\cos\theta\cos\phi, 0) \\ &= r(-\sin\theta\cos\phi\boldsymbol{i} + \cos\theta\cos\phi\boldsymbol{j}); \end{aligned}$$

$$\begin{aligned} D_2\boldsymbol{p}(\theta,\phi) &= (-r\cos\theta\sin\phi, -r\sin\theta\sin\phi, r\cos\phi) \\ &= r(-\cos\theta\sin\phi\boldsymbol{i} - \sin\theta\sin\phi\boldsymbol{j}) + r\cos\phi\boldsymbol{k}. \end{aligned}$$

Da qui si ottiene

$$\begin{aligned} \boldsymbol{n} &= D_1\boldsymbol{p}(\theta,\phi) \times D_2\boldsymbol{p}(\theta,\phi) \\ &= r^2(\sin^2\theta\sin\phi\cos\phi\ \boldsymbol{i}\times\boldsymbol{j} - \sin\theta\cos^2\phi\ \boldsymbol{i}\times\boldsymbol{k} \\ &- \cos^2\theta\sin\phi\cos\phi\ \boldsymbol{j}\times\boldsymbol{i} + \cos\theta\cos^2\phi\ \boldsymbol{j}\times\boldsymbol{k}) \\ &= r^2(\sin\phi\cos\phi\ \boldsymbol{k} + \sin\theta\cos^2\phi\ \boldsymbol{j} + \cos\theta\cos^2\phi\ \boldsymbol{i}) \end{aligned}$$

perché $\boldsymbol{i}\times\boldsymbol{j} = \boldsymbol{k} = -\boldsymbol{j}\times\boldsymbol{i},\ \boldsymbol{j} = \boldsymbol{k}\times\boldsymbol{i} = -\boldsymbol{i}\times\boldsymbol{k}$ e $\boldsymbol{i} = \boldsymbol{j}\times\boldsymbol{k}$.
Ora è elementare calcolare la norma di $\boldsymbol{n} = D_1\boldsymbol{p}(\theta,\phi) \times D_2\boldsymbol{p}(\theta,\phi)$. Si trova

$$\|\boldsymbol{n}(\theta,\phi)\| = r^2\cos\phi. \tag{5.2.3}$$

Questa formula quindi fornisce lo jacobiano della trasformazione di coordinate da cartesiane a coordinate di latitudine e longitudine (si osservi che il coseno è non negativo in $[-\pi/2, \pi/2]$). Pertanto

$$\frac{\boldsymbol{n}(\theta,\phi)}{\|\boldsymbol{n}(\theta,\phi)\|} = \cos\theta\cos\phi\,\boldsymbol{i} + \sin\theta\cos\phi\,\boldsymbol{j} + \sin\phi\,\boldsymbol{k} = \frac{\boldsymbol{p}(\theta,\phi)}{r}$$

è il versore diretto radialmente, esattamente come ci si aspettava. □

NOTA 5.2.5 (Angoli di Eulero). *Spesso si usa una parametrizzazione equivalente ma diversa, nella quale l'angolo ϕ è il complementare dell'angolo di latitudine. La scelta degli angoli diventa quindi come in questa figura: Si noti che l'angolo ϕ ora varia da 0 (polo nord) a π (polo sud). Ovviamente, la parametrizzazione corrispondente alla nuova figura, ottenuta scambiando ϕ con il suo complementare, si ricava scambiando fra loro* $\sin\phi$ *e* $\cos\phi$*, e quindi è*

$$x = r\cos\theta\sin\phi, \qquad y = r\sin\theta\sin\phi, \qquad z = r\cos\phi. \tag{5.2.4}$$

Queste coordinate si chiamano le coordinate sferiche, e gli angoli θ e ϕ si chiamano gli angoli di Eulero. *Allo stesso modo, lo jacobiano della trasformazione di coordinate sferiche in* (5.2.3)*, con questa nuova scelta degli angoli, diventa $r^2\sin\phi$.*

5.3. Mappa di tessitura

La mappa di tessitura è un metodo per semplificare il rendering di aree della scena complesse ma di significatività non così rilevante da giustificare un rendering in completo dettaglio. Il procedimento consiste nel mappare su quelle aree immagini prefissate e poi trasformarle in prospettiva. Il metodo fu introdotto in **[6, 7]**. Talvolta l'immagine mappata si chiama (impropriamente) mappa di tessitura ed i suoi pixel vengono chiamati texel. L?immagine è un rettangolo di pixel (o meglio, texel) colorati dotato di un suo sistema di coordinate (u, v). Noi riserviamo il nome mappa di tessitura

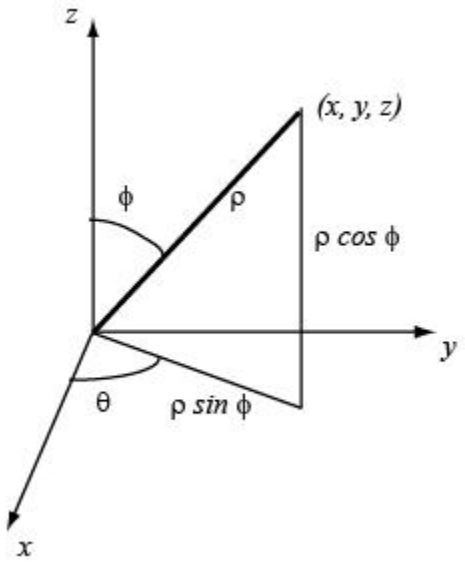


FIGURA 5.2.2. Angoli di Eulero: il polo Nord ha angolo di deviazione polare uguale a zero, il polo Sud ha angolo π

alla trasformazione che fa corrispondere le coordinate (u, v) dell'immagine ad appropriati punti di oggetti della scena.

La mappa viene eseguita nel modo seguente. Consideriamo un pixel del piano di visuale. Le coordinate (x, y) dei suoi vertici vengono mappate su quelle dell'oggetto della scena visibile attraverso quel pixel, ed i punti risultanti vengono trasformati nelle coordinate (u, v) dei punti della immagine di tessitura che gli corrispondono (attraverso la inversa della mappa di tessitura). In tal modo il pixel viene trasformato in un quadrilatero contenuto nel rettangolo dell'immagine della tessitura, il quale generalmente interseca parecchi texel. Il valore di colore del pixel si ottiene come media pesata di quelli di tali texel, ciascuno pesato in proporzione all'area della parte del texel che giace dentro il quadrilatero. Se nell'eseguire l'inversa della mappa di tessitura accade che una parte di un pixel finisca fuori del rettangolo dell'immagine, questa si può supporre prolungata in maniera periodica. Quando, come di solito, le superficie della scena sono poligoni, è d'uso assegnare direttamente ai loro vertici le coordinate (u, v) dell'immagine che vi deve essere mappata, per poi trovare le coordinate di tessitura dei punti interni per interpolazione lineare. In questa fase insorgono quindi gli stessi due problemi già constatati nello studio dell'interpolazione (Sottosezione 3.3). Il primo è il fatto che per poligoni di più di tre lati il risultato dell'interpolazione può dipendere dal modo in cui il poligono è orientato: questo fatto costringe a suddividere tutti i poligoni in triangoli prima di procedere all'interpolazione. Il secondo problema è l'accorciamento dovuto alla distanza nella trasformazione prospettica: l'interpolazione, che avviene riga per riga di scansione, non rimpicciolisce corrispondentemente l'immagine mappata perché due righe consecutive differiscono sempre della stessa altezza, ma si proiettano su curve nel poligono della scena a profondità variabili diverse e non necessariamente equispaziate. L'effetto di distorsione prospettica che ne segue si può ridurre suddividendo i poligoni in poligoni più piccoli; per una soluzione non approssimata occorre effettuare la trasformazione prospettica, punto per punto, nel corso della scansione di interpolazione.

Si badi bene che tappezzare una scena con una tessitura non completa il procedimento di rendering realistico: non basta poi proiettare il risultato sul viewport, ad esempio come nel procedimento di $z-$buffer. Se facessimo questo, l'illuminazione sarebbe sbagliata. Per una illuminazione realistica

abbiamo bisogno di un modello di illuminazione, che utilizzi i versori che individuano la direzione di provenienza della luce, quella dell'osservatore e la direzione normale uscente (ad esempio, il modello di Phong o un modello fisico). Quindi, se ad esempio usiamo ray tracing, al punto della scena individuato dal raggio proiettore proveniente dall'osservatore dobbiamo approfittare della mappa di tessitura per ricavare il colore di quel punto, ma dopo dobbiamo calcolare i versori $\boldsymbol{L}$, $\boldsymbol{V}$ e $\boldsymbol{N}$ ed applicare un modello di illuminazione. ma allora in cosa consiste il vantaggio nella riduzione della mole di calcoli? Non consiste nel fatto che dipingiamo in maniera piatta una scena con una tappezzeria, bensí nel fatto che possiamo scegliere una modellazione a maglie molto piu' larghe (e quindi con molti meno poligoni): infatti, non abbiamo più bisogno che all'interno di ogni maglia il colore cambi di poco, in quanto la mappa di tessitura ci fornisce il colore dei punti interni con elevata precisione (data dalla risoluzione dell'immagine usata per la tessitura, e non dal tasso di variabilità cromatica della scena).

Esempio 5.3.1. *(Tessiture per la superficie terrestre e lunare).* La mappa inversa della trasformazione in coordinate sferiche si chiama *proiezione di Mercatore*. Se applicata al disegno dei continenti terrestri sul mappamondo, questa proiezione ci restituisce una immagine in cui le aree vicino ai poli risultano dilatate e stirate orizzontalmente, come in Figura 5.3.1. Se si vuole invece disegnare nel mappamondo l'aspetto notturno della Terra, con le luci delle città, si può usare una tessitura come in Figura 5.3.2. Dopo aver mappato una di queste tessiture su una sfera, si può sovrapporre al risultato una seconda tessitura, come si farebbe se dovessimo dipingere sopra una parete già affrescata. Ad esempio, in Figura 5.3.3 una tessitura per ammassi nuvolosi che rispetta la legge di Coriolis, per la quale il moto ciclonico delle nuvol eavviene in senso antiorario nell'emisfero Nord ed orario nell'emisfero Sud. In una animazione, potremmo applicare una tessitura animata, che cambia ad ogni nuovo fotogramma, per far muovere le nuvole in questo modo. Se vogliamo disegnare una scena del sistema Terra-Luna, ci occorre anche una proiezione di Mercatore della superficie della Luna. Della Luna vediamo solo la faccia frontale alla Terra, perché il suo periodo di rivoluzione intorno alla Terra coincide con il periodo di rotazione. Ma l'altra faccia è stata fotografata dalle sonde della NASA ed è disponibile su Internet. Riportiamo la tessitura dell'intero globo lunare in Figura 5.3.4. Possiamo creare uno sfondo per il sistema Terra-Luna, tramite una tessitura di sfondo data da un'immagine della Galassia opportunamente colorata e ritagliata, a partire ad esempio dall'immagine in Figura 5.3.5.

Le tessiture della Terra e della Luna si possono fare slittare orizzontalmente, fotogramma dopo fotogramma, per generare una animazione nella quale Terra e Luna sembrano ruotare intorno al proprio asse polare. Ecco il codice Java del metodo di slittamento:

```
public void rotateTexture(double theta){
Point p;
for (int i=0; i<pts.size(); i++) {
p = pts.get(i);
p.u += theta;
//p.u = p.u % 1.0; //if(p.u >= 1) p.u -= 0;
}
}
```

Si noti che, quando la tessitura viene mappata sui punti della scena, le coordinate u e v della tessitura sono state associate direttamente a ciascun punto p della scena: ossia, nella classe *Point* ogni punto vine munito di attributi $p.u$ e $p.v$. Allora la rotazione della Terra viene visualmente resa tramite una traslazione della tessitura lungo l'asse orizzontale di un incremento fissato, lo stesso per ogni fotogramma: $p.u{+}{=}theta$. Si noti, in effetti, che la tessitura è periodica lungo l'asse orizzontale. Perché le coordinate di tessitura, e quindi di colore, sono associate direttamente ai punti della scena?Perché, in generale nel mappamondo, o meglio rispetto al piano dell'eclittica, l'asse polare

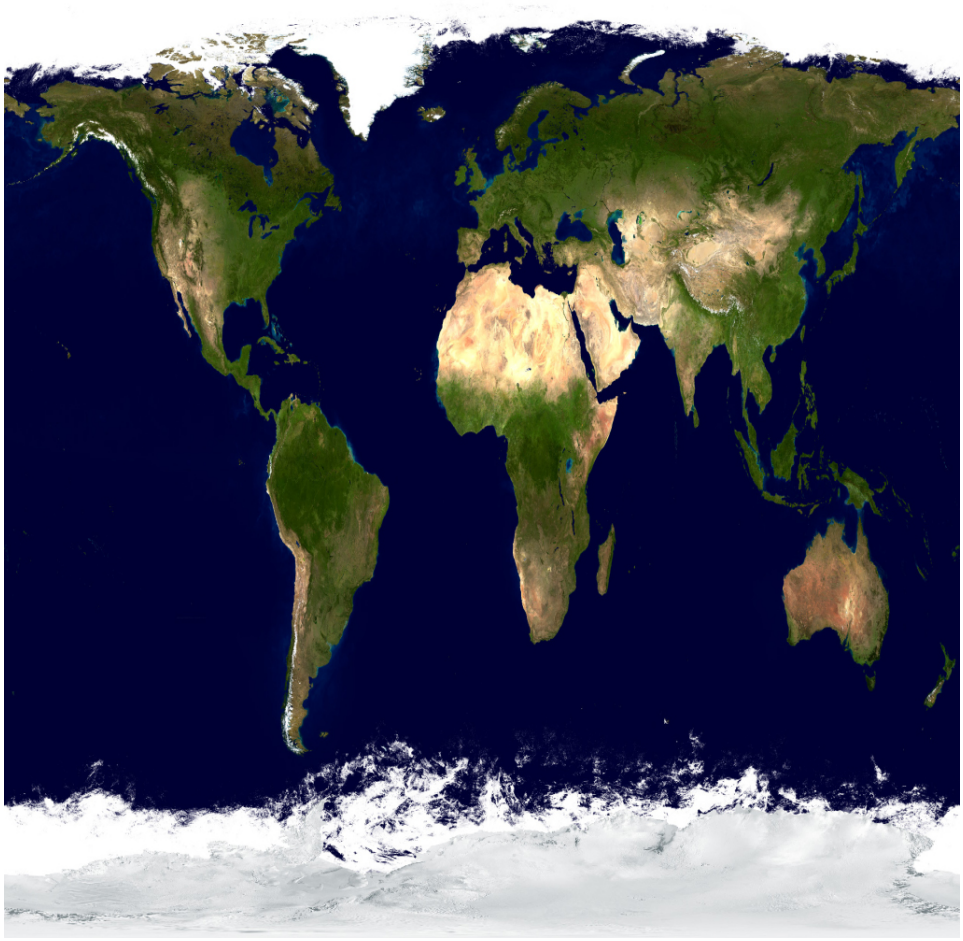


FIGURA 5.3.1. Mappa della Terra in proiezione di Mercatore: applicando a questa immagine la trasformazione in coordinate sferiche, si ottiene una sfera con sopra disegnata la Terra, come nel mappamondo

della Terra non è disposto verticalmente, bensì ad un angolo di circa 23 gradi (più precisamente, 23 26' 16"): questo è l'angolo di inclinazione rispetto al piano dell'eclittica. Per rendere la Terra in questa forma inclinata, dopo aver mappato la tessitura occorre ruotarla di questo angolo nel piano $\{x, y\}$ (ossia intorno all'asse z della profondità. Ma non si può traslare la tessitura di un tale angolo, perché la tessitura è periodica solo orizzontalmente e quindi occorre traslarla orizzontalmente. Per questo la si associa direttamente ai punti della scena e poi si ruotano questi ultimi, cosicché nel ruotare si portano con sé la loro tessitura. Ecco il codice Java dell'appropriato metodo di rotazione:

```
public void rotateObjectY(double theta, Point center){
Point p;
double yr,xr,nxr,nyr;
double cos = Math.cos(theta), sin = Math.sin(theta);
for (int i=0; i<pts.size(); i++) {
p=pts.get(i);
p.shift(center.per(-1));
xr = p.x*cos + p.y*sin;
yr = -p.x*sin + p.y*cos;
```

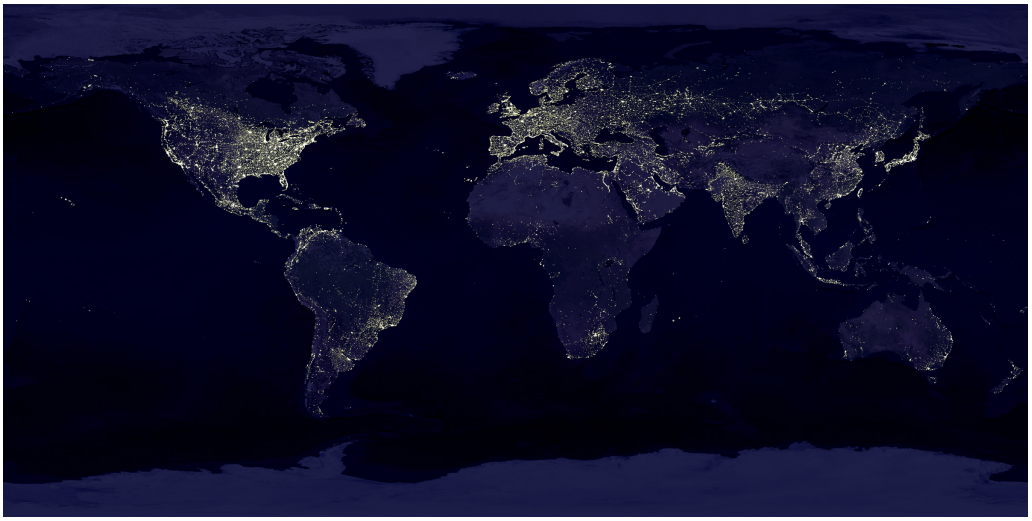


FIGURA 5.3.2. Mappa notturna della Terra in proiezione di Mercatore: applicando a questa immagine la trasformazione in coordinate sferiche, si ottiene un mappamondo con la Terra vista di notte

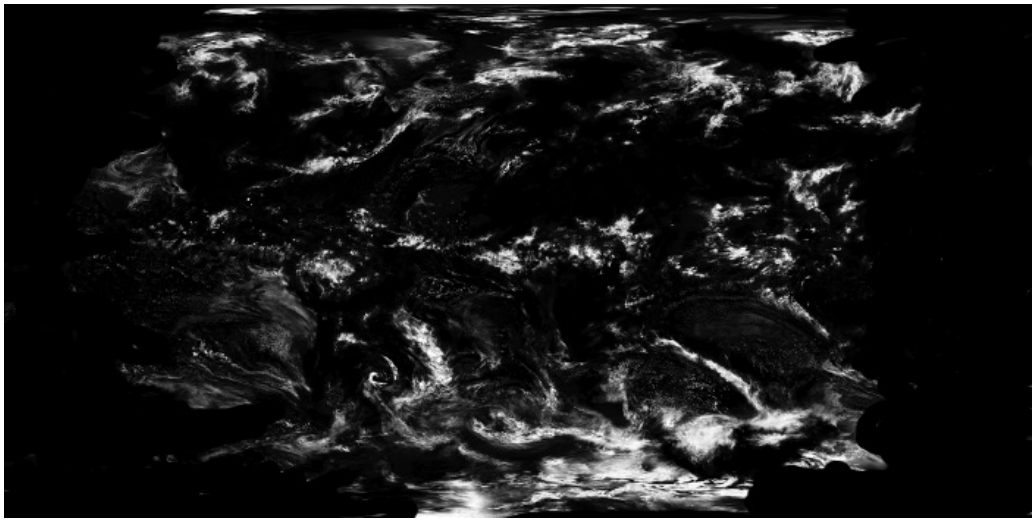


FIGURA 5.3.3. Una tessitura per ammassi nuvolari in proiezione di Mercatore

```
nxr = p.normal.x*cos + p.normal.y*sin;
nyr = -p.normal.x*sin + p.normal.y*sin;
p.set(xr, yr, p.z);
p.normal.set(nxr, nyr, p.normal.z);
p.shift(center);
}
}
```

Se a questo punto aggiungiamo una luce esterna (il Sole) ed una luce ambientale (per evitare che le zone in ombra risultino completamente nere), si ottengono, ad ogni fotogramma, immagini del tipo illustrato in Figura 5.3.6. □

FIGURA 5.3.4. Mappa della Luna in proiezione di Mercatore

FIGURA 5.3.5. Una immagine colorata di un particolare della Via Lattea

5.4. Esercizi sulle mappe di tessitura e di rilievo

ESERCIZIO 5.4.1. Il piano di coordinate u, v viene munito di una tessitura data da un gradiente che varia linearmente dal nero al punto (0,0) al bianco al punto (1,1): per i punti di tale piano con proiezione negativa sull'asse di questo gradiente (ossia la bisettrice del primo e terzo quadrante) il colore è nero, per quelli per cui la proiezione è maggiore di $\|(1,1)\| = \sqrt{2}$, il colore è bianco.
Consideriamo la sfera di raggio 1 e centro l'origine, parametrizzata con gli angoli di Eulero ϕ di latitudine (con $\phi = 0$ al polo Nord e $\phi = \pi$ al polo Sud) e θ di longitudine (con $\theta = 0$ o π sul piano $\{y = 0\}$).

1. Applichiamo su questa sfera la tessitura, mappando il quadrato $0 \leqslant u \leqslant 1$, $0 \leqslant v < 1$ sulla sfera tramite la mappa di tessitura data da $\phi = \pi\, u$, $\theta = 2\pi\, v$. Quali punti della sfera risultano di colore grigio con intensità 50%?
2. Ruotiamo la tessitura di 45 gradi in senso orario, ossia ruotiamo l'asse del gradiente in modo che coincida con l'asse u, e dopo riapplichiamo la mappa di tessitura dal quadrato $[0,1] \times [0,1]$ alla sfera, definita come prima. Ora quali sono i punti della sfera grigi al 50%?

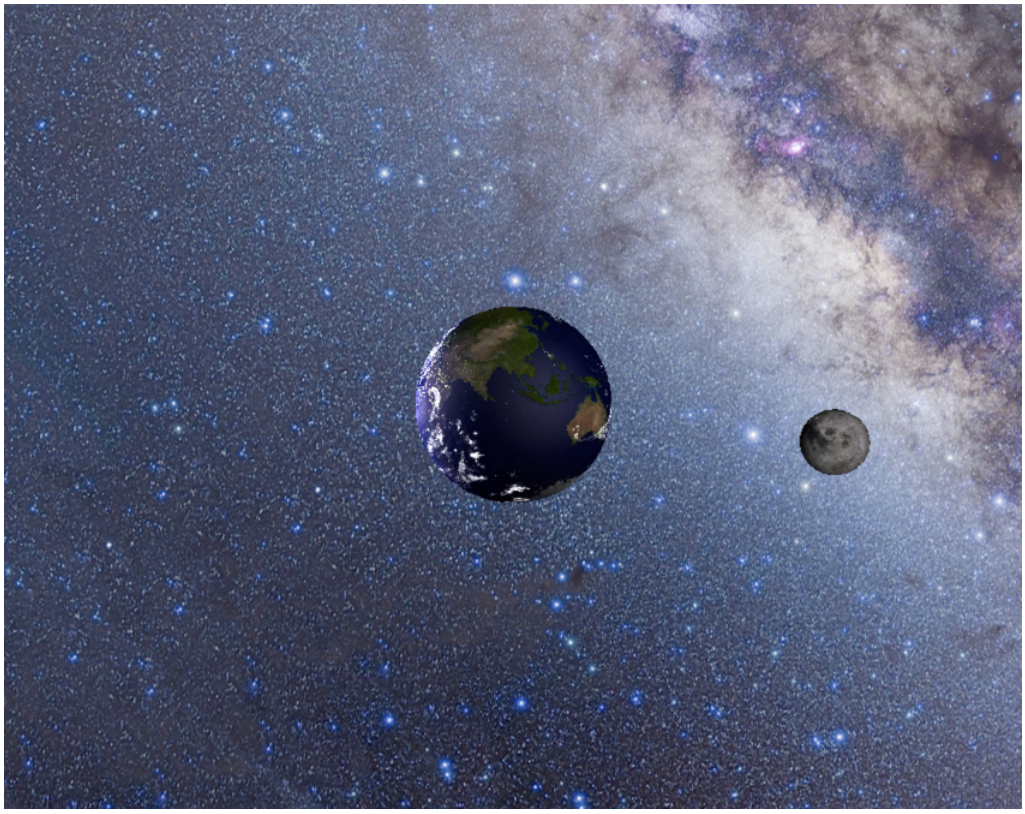


FIGURA 5.3.6. Un fotogramma dell'animazione del sistema Terra-Luna

Svolgimento. Il versore gradiente della tessitura è orientato lungo la bisettrice del primo quadrante, ossia proporzionale a $(1, 1)$. Pertanto la tessitura assume valore costante sulle rette $u+v =$ costante. Il valore 0 è assunto all'origine, il valore 1 al vertice opposto del quadrato, ed il valore $1/2$ sull'antidiagonale a metà strada, ossia la retta $u + v = 1/2$. La risposta alla parte [(1) è l'equazione di questa retta mappata sulla sfera: nelle coordinate ϕ, θ l'equazione della retta diventa

$$\phi + \frac{\theta}{2} = \frac{\pi}{2} \, .$$

Nota: le curve $u =$ costante sono mappate sui paralleli, le curve $v =$ costante sui meridiani; i lati del quadrato $u = 0$ e $u = 1$ sono mappati rispettivamente al polo Nord ed al polo Sud; i lati $v = 0$ e $v = 1$ corrispondono entrambi al meridiano di Greenwich. La curva appena trovata va dal polo Nord al polo Sud intersecando tutti i meridiani e tutti i paralleli.

La risposta alla parte (2) è il parallelo di latitudine 0, ossia l'equatore. □

ESERCIZIO 5.4.2. Per la sfera del problema precedente, si scrivano le coordinate del versore normale esterno al punto generico (ϕ, θ) e si applichi la mappa di rilievo $b(\theta, \phi) = \sin^2 \phi/10$. Come cambia il versore normale esterno? Scrivere le sue nuove coordinate, con la consueta approssimazione di Taylor data dalla mappa di rilievo.

Svolgimento. Basta usare la formula di Blinn (5.1.4). La mappa $\boldsymbol{p} = (x(\theta, \phi), y(\theta, \phi), z(\theta, \phi))$ è data da $x = r\cos\theta\cos\phi$, $y = r sin\theta\cos\phi$, $z = r\sin\phi$. . Poiché la superficie è una sfera, la sua normale $\boldsymbol{N}$ al punto $\boldsymbol{p}$ è il versore radiale $\boldsymbol{N} = \boldsymbol{p}/\|\boldsymbol{p}\|$. Il prodotto vettore $\boldsymbol{n} = D_\theta \boldsymbol{p} \times D_\phi \boldsymbol{p}$ è diretto radialmente, e quindi $\boldsymbol{N} = \boldsymbol{n}/|\,\boldsymbol{n}\|$. Sappiamo che, per la mappa $\boldsymbol{p}$ (una versione della parametrizzazione di Eulero per la sfera) si ha $|\,\boldsymbol{n}\| = \cos\phi$.

In base alla formula di Blinn, dopo aver spostato $\boldsymbol{p}$ in $\boldsymbol{p} + b\boldsymbol{N}$, la nuova normale diventa, al primo ordine di sviluppo di Taylor,
Osservando che

$$\begin{aligned} D_\theta b &= 0, \\ D_\phi b &= \sin\phi\cos\phi/5, \\ D_\theta \boldsymbol{p} &= (-\sin\theta\cos\phi,\ \cos\theta cos\phi,\ 0), \\ D_\phi \boldsymbol{p} &= (-\cos\theta\sin\phi, -\sin\theta\sin\phi, \cos\phi), \end{aligned}$$

si arriva facilmente alla soluzione, che è lasciata al lettore. □

5.5. Mappa di riflessione

Quando un oggetto della scena ne riflette altri si parla di riflessione fra oggetti, o interriflessione. L'effetto può variare fra riflessioni speculari il cui risultato cambia sensibilmente con la posizione dell'osservatore a riflessioni diffuse che non ne dipendono. Il primo caso viene reso in maniera realistica dal ray tracing ricorsivo a numerose generazioni di raggi, il secondo dalla radiosità. Entrambi questi procedimenti sono però estremamente dispendiosi in termini di mole di calcoli. Per quanto concerne la interriflessione speculare, una scorciatoia che rappresenta un buon compromesso fra velocità di esecuzione e precisione del rendering è rappresentata dalla mappa di riflessione, introdotta in **[6]**. Il procedimento è il seguente. Si immagina di inscrivere l'intero ambiente in una grande superficie sferica, con centro prefissato nell'ambiente in vicinanza al centro della scena, se identificabile. Questa sfera contiene tutti gli oggetti al suo interno. Dal centro si esegue la proiezione radiale di ciascun oggetto sulla superficie della sfera, che quindi diventa una tessitura bidimensionale: ciascun texel viene colorato con il colore (e l'ombreggiatura) del punto dell'oggetto più vicino al centro della sfera che la proiezione mappa su quel texel (più vicino al centro della sfera, ossia più lontano dalla sfera stessa). Il colore di tale punto si potrebbe ottenere, ad esempio, mediante il metodo di illuminazione di Phong e l'ombreggiatura di Phong oppure di Gouraud: questa procedura, però, dà origine ad una aberrazione dell'illuminazione (ossia dei valori di luminosità in ogni canale, e quindi del colore), perché il metodo di illuminazione di Phong utilizza l'illuminazione come vista dall'osservatore, mentre in questo caso dobbiamo calcolare invece l'illuminazione come vista dal centro della sfera. Pertanto il meccanismo corretto consiste nel proiettare radialmente sulla sfera il valore dell'illuminazione del punto più vicino al centro della sfera lungo quel raggio, calcolato considerando la posizione delle sorgenti di luce ma immaginando che l'osservatore sia spostato al centro della sfera. Evitare questo spostamento è una semplificazione che talvolta fa risparmiare tempo, perché l'illuminazione vista dalla posizione dell'osservatore a molti punti della scena viene comunque calcolata durante il ray tracing ricorsivo, e potrebbe essere memorizzata per riutilizzo successivo: ma come si è visto introduce una aberrazione cromatica e di luminosità.

Osserviamo che la costruzione della tessitura su cui si basa la mappa di riflessione è un esempio di $z-$buffer radiale (ossia sferico): la proiezione radiale su una sfera. Ma si può riutilizzare il procedimento di $z-$buffer piano precedentemente introdotto, in Sezione **??** proiettando, invece che su una sfera, sulle sei facce di un cubo che include tutta la scena. In tal modi l'intera mappa di riflessione si basa su sei $z-$buffer piani. Notiamo che quindi anche questo metodo consiste di una forma di accelerazine del Ray Tracing tramite $z-$buffer.

Questa tessitura (mappa di riflessione) viene usata come la mappa di tessitura: quando si esegue il rendering di un punto su un oggetto, si calcola il versore riflesso, cioè la riflessione del versore di visuale (direzione dell'osservatore, considerato nella sua posizione vera, non quella spostata al punto osservato) rispetto al versore normale del punto osservato, e lo si usa come puntatore, ossia si recupera il colore da quello del punto della sfera puntato da quel versore (pensato come vettore uscente dal centro della sfera). Pensato come vettore applicato al punto osservato (quindi in un contesto di spazi

affini), questo versore indica in che direzione dovremmo guardare, dal punto osservato, per vedere cosa colpisce la direzione speculare della direzione incidente data dalla posizione dell'osservatore. La mappa di riflessione contiene proprio l'illuminazione vista guardando dal punto osservato verso tale direzione speculare. Pertanto, i coseni direttori del versore riflesso vengono usati come indici per la mappa di riflessione (a questo scopo la cosa più pratica è di scrivere le coordinate del versore riflesso in termini degli angoli di Eulero, cioè degli angoli di latitudine e longitudine sulla sfera: questi angoli indicizzano in maniera biunivoca i punti della sfera eccetto che ai poli nord e sud). Questo procedimento equivale a visualizzare la scena a partire da un suo punto di proiezione centrale (diverso da quello dell'osservatore, che non è centrale ma frontale) come se fosse proiettata su una carta geografica. Una variante di questa parametrizzazione con gli angoli di Eulero, introdotta in [**21**], consiste nel sostituire l'angolo di latitudine ' con sin': in tal modo, poiché la misura equidistribuita sulla sfera in termini degli angoli di Eulero si esprime come r sin' d' d' , la proiezione preserva le aree. Una variante ancora più utile e comunemente usata consiste nel proiettare non su una sfera ma su un cubo. In tal modo, la determinazione del colore dei texel delle sei facce del cubo si ottiene tramite altrettanti $z-$buffer, facilmente implementabili via hardware e veloci. Si deve solo fare attenzione ad orientare correttamente gli $z-$buffer, in maniera che i lati combacianti delle sei facce corrispondano.

La mappa di riflessione introduce distorsioni di parallasse. Infatti, nel determinare il colore di un punto, non si segue il raggio riflesso a partire da quel punto (questo coinciderebbe col metodo di ray tracing), ma si utilizzano i coseni direttori di tale versore come puntatore alla mappa di riflessione, in altre parole si segue la direzione del versore riflesso applicato al centro di proiezione (centro della sfera o del cubo). Quanto maggiore è la distanza fra il punto osservato ed il centro di proiezione, tanto maggiore è l'errore di parallasse. Solo se la sfera ha raggio infinito (cioè se l'intera mappatura avviene nel senso della geometria proiettiva) l'errore di parallasse si annulla. Da questa osservazione segue la necessità di scegliere il raggio della sfera (o cubo) molto grande rispetto alle dimensioni della scena, ed il suo centro molto prossimo al baricentro della scena. In ogni caso, le distorsioni aumentano quando si rende la riflessione di altri oggetti su una superficie della scena lontana dal centro. Per questo motivo spesso si usa più di una mappa di riflessione: si scelgono varie sfere, con centri diversi in prossimità delle superficie riflettenti, e per ciascuna superficie si utilizza la mappa di riflessione sulla sfera con il centro più vicino. Si deve comunque osservare che la riflessione su una superficie curva è comunque soggetta a distorsione geometrica: le forme vengono distorte dalla curvatura. Quindi l'effetto visivo della ulteriore distorsione causata dall'errore di parallasse è assai ridotto, tranne nel caso in cui la superficie sia uno specchio piano. D'altra parte, per specchi piani di piccole dimensioni, l'area da rendere è piccola e quindi una distorsione ha poca influenza sul realismo visivo. Invece, per specchi piani grandi, la mappa di riflessione è poco utile e di solito non impiegata, perché la direzione del versore riflesso varia molto lentamente e quindi esso punta più o meno ovunque quasi allo stesso punto della mappa: in tal caso si avrebbe una colorazione quasi costante, mentre nella realtà (e nel ray tracing ricorsivo, che non compie errori di parallasse) la riflessione su uno specchio crea una scena virtuale con colorazione simmetrica a quella della scena originale.

5.5.1. Esercizi sulla mappa di riflessione. Nei prossimi esercizi ignoriamo il fatto che i valori di illuminazione nella mappa di riflessione si debbano ottenere proiettando i valori di illuminazione della scena reale calcolati immaginando l'osservatore spostato al centro della sfera di proiezione. In effetti, questi esercizi mirano solo all'esame della parallasse geometrica, e quindi, per semplicità, dipingiamo artificialmente la mappa di riflessione, ossia dipingiamo la sfera (con meridiani e paralleli, numerati).

Esercizio 5.5.1. Una scena consiste di una sfera S con centro l'origine e raggio 1, dalle pareti perfettamente speculari, ed una sfera B con centro l'origine e raggio 10, dipinta con i meridiani ed i paralleli, come un mappamondo. L'osservatore si trova al punto (2,0,0).

1. Quale parte di B (in termini di meridiani e paralleli) l'osservatore vede riflessa in S? Si usi la parametrizzazione della sfera tramite gli angoli di Eulero (5.2.2).
2. Se l'osservatore guarda la sfera S con una deviazione angolare di angolo ψ rispetto alla direzione del centro, quali punti della sfera B vi vede riflessi? Ossia, si determini l'andamento dell'angolo φ di apertura angolare del cono che descrive i punti della sfera B visti quando l'osservatore guarda S con una deviazione angolare ψ. In particolare, si determini l'andamento di φ quando ψ è piccolo (ossia si sviluppi secondo Taylor $\varphi(\psi)$ intorno a $\psi = 0$).
3. **(La distorsione geometrica nel riflesso su una sfera.)** Si valuti l'andamento di φ rispetto a ψ in prossimità dell'angolo di tangenza: l'ordine di divergenza di tale andamento misura la distorsione geometrica massima della mappa di riflessione, ossia il fattore di ampliamento fra area guardata e corrispondente area riflessa.

□

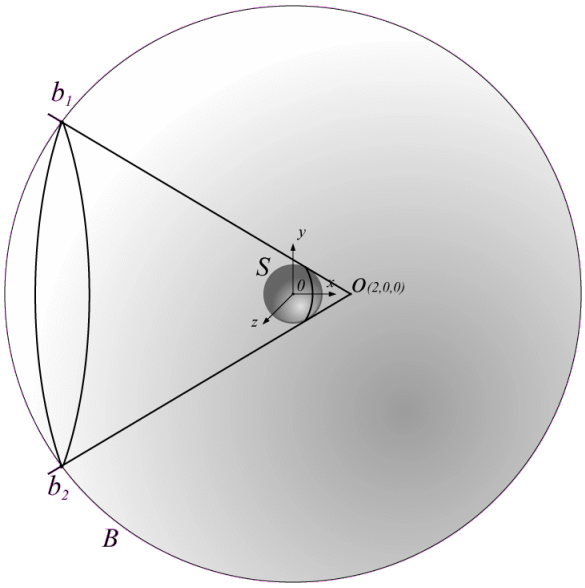


FIGURA 5.5.1. La scena dell'Esercizio 5.5.1

Trascrivere qui la soluzione dell'esercizio precedente e e del prossimo

ESERCIZIO 5.5.2. Una scena consiste di un cubo con centro l'origine e lato 2, dalle pareti perfettamente speculari parallele agli assi coordinati, ed una sfera S con centro l'origine e raggio 10, dipinta con i meridiani e i paralleli, come un mappamondo. L'osservatore si trova nel punto $\boldsymbol{o} = (2, 2, 2)$.

1. Sia F la faccia del cubo più vicina all'osservatore e parallela al piano $\{x, y\}$. Quale valore di meridiano e parallelo (cioè quale latitudine e longitudine) l'osservatore vede riflessi al centro della faccia F? Si utilizzi la parametrizzazione della sfera data dagli angoli di Eulero.
2. Come cambia la risposta precedente se il cubo viene traslato in modo che il centro della faccia F sia l'origine?
3. Si consideri la faccia più vicina all'osservatore e parallela al piano $\{x, z\}$. Si consideri il segmento I dato dall'intersezione di questa faccia con il piano $\{y, z\}$. Si trovino i punti della sfera che l'osservatore vede riflessi in I.

□

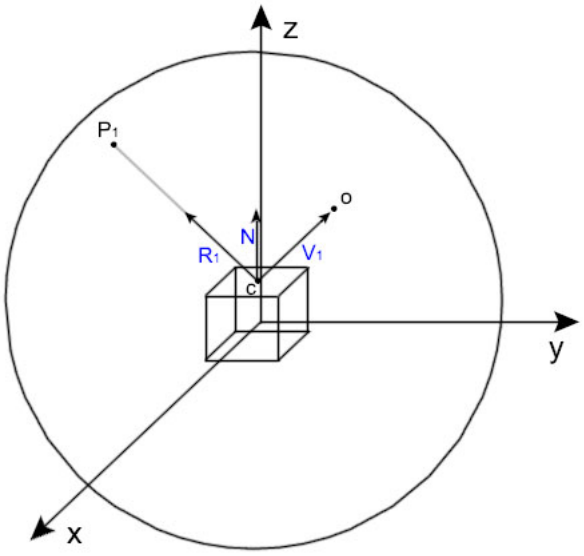


FIGURA 5.5.2. La scena dell'Esercizio 5.5.2

ESERCIZIO 5.5.3. Una scena si compone di una stanza cubica di lato 3 metri con i muri opachi (coefficiente di riflettività 0 e di Lambert 1, coefficiente di Phong 1), pavimento rosso puro, pareti blu puro, soffitto verde puro, tutti e tre i colori alla massima intensità, e di uno specchio ideale (coefficiente di riflettività 1 e di trasmissione e di Lambert 0, coefficiente di Phong infinito) di forma quadrata, di diametro 1 metro, sospeso ad altezza 1 metro, disposto parallelamente al pavimento con i lati paralleli ai muri e con il centro posto verticalmente sopra il centro del pavimento. Supponiamo che l'origine coincida con uno dei vertici del pavimento, e che l'osservatore stia sul soffitto al punto $\boldsymbol{e} = (2, 2, 3)$. La luce è al punto $\boldsymbol{l} = (1.5, 1.5, 2.5)$. Che colore vede l'osservatore se guarda verso il centro dello specchio ma la riflessione è calcolata con la mappa di riflessione avente per centro il centro del pavimento? □

ESERCIZIO 5.5.4. Una scena si compone di un triangolo T rosso puro alla massima intensità, opaco, con vertici in $(2, 0, 0)$, $(2, 2, 0)$ e $(0, 0, 2)$, e della calotta sferica data dalla parte S della superficie sferica Σ di raggio 5 con latitudine compresa fra $\pi/4$ e $\pi/2$ (equatore), verde pura alla massima intensità, opaca.
Si calcoli la proiezione centrale della scena sulla sfera Θ con centro l'origine e raggio 10 e centro di proiezione l'origine, come segue.

(*i*) Si determinino le proiezioni su Σ dei tre vertici di T. *(Suggerimento: basta rinormalizzare i tre vertici in maniera che abbiano norma uguale al raggio di Σ).*
(*ii*) Si determinino i tre lati della proiezione di T. *(Suggerimento: basta parametrizzare i lati, ad esempio come segmenti di rette in forma parametrica, e rinormalizzare come prima).*
(*iii*) Si determini la proiezione di S. *(Suggerimento: si tratta della calotta sferica in Θ dall'equatore al parallelo a nord a $\frac{\pi}{4}$ radianti).*
(*iv*) Ora si calcoli la mappa di riflessione richiesta. Si può assumere che il colore di sfondo (ossia quello della sfera Θ nelle aree su cui non si proietta nessun poligono della scena) sia

nero. *(Suggerimento: il triangolo è più vicino all'origine della calotta sferica, quindi la copre laddove le proiezioni si sovrappongono).*

□

ESERCIZIO 5.5.5. Si aggiunga alla scena una superficie puramente riflettente (specchio ideale) R data dall'emisfero con centro l'origine e raggio 1 nel semispazio $\{z \geqslant 0\}$, un osservatore nel punto $(0, 0, 4)$.

1. Si parametrizzi R con gli angoli di Eulero e per ciascun suo punto ϕ, θ si calcoli la direzione del versore $\mathbf{r}$ ottenuto riflettendo rispetto alla normale il versore $\mathbf{v}$ che individua la posizione dell'osservatore. *(Suggerimento: il versore normale al punto di angoli di Eulero ϕ, θ è il versore radiale* $(\sin\phi, \cos\phi \sin\theta, \cos\phi\cos\theta)$*).*
2. Per ciascun tale punto si determini tramite mappa di riflessione cosa vede l'osservatore quando guarda quel punto, supponendo che ci sia solo una luce ambiente bianca di intensità 1, e che tutti i coefficienti di riflessione della luce ambiente valgano 1. *(Suggerimento: i poligoni colorati dalla sola luce ambientale hanno colore costante; si usi la direzione del versore riflesso* $\mathbf{r}$ *come puntatore alla mappa di riflessione precedentemente calcolata, che va espressa in termini degli angoli di Eulero).*

□

ESERCIZIO 5.5.6. Ora supponiamo che T abbia coefficienti di Lambert e di riflessione speculare entrambi uguali a $\frac{1}{2}$ e che quelli di S siano entrambi uguali a $\frac{1}{3}$. Entrambi gli esponenti di Phong valgono 1. Lo specchio ideale Q ovviamente ha coefficiente di Lambert 0, di riflessione speculare 1 e di Phong infinito, e colore bianco. Tutti i coefficienti di riflessione ambientale sono 0. Lo sfondo è nero. C'è una sorgente di luce puntiforme che coincide con la posizione dell'osservatore. Lo sfondo è la sfera Σ dipinta di nero.
Si calcoli il colore che vede l'osservatore quando guarda in direzione $\left(\frac{1}{20}, \frac{1}{20}, -1\right)$, utilizzando il Ray Tracing ricorsivo con una sola generazione di raggi. *(Suggerimento: si determini il piano dove giace il triangolo T ed il punto* $\mathbf{p}$ *dove il raggio proiettore lo interseca. Si provi che questo punto è interno a T (ad esempio proiettando su uno dei piani coordinati per ridurre il problema a dimensione due, oppure esprimendo* $\mathbf{p}$ *come combinazione convessa dei vertici di T). Si calcoli il contributo di* $\mathbf{p}$ *all'illuminazione grazie all'equazione di Phong (ma attenzione all'angolo fra la direzione di provenienza del raggio proiettore e quella del versore riflesso della luce: se è maggiore di 90 gradi il termine di Phong è nullo, e rimane solo quello di Lambert). Si trovi poi il versore riflesso* $\mathbf{r}$ *del versore da* $\mathbf{p}$ *alla posizione dell'osservatore* $\mathbf{v}$*, e si tracci il raggio riflesso, calcolando le intersezioni con il piano di base e con il piano orizzontale su cui giace il bordo superiore della calotta sferica. Si noti che l'intersezione con il piano di base è esterna a Σ mentre quella col piano superiore è interna - quindi il raggio proiettore deve intersecare la calotta S. Si calcoli il punto di intersezione ed il suo contributo al Ray Tracing ricorsivo).* □

ESERCIZIO 5.5.7.

1. Una scena consiste di una stanza cilindrica avente per base un disco intorno all'origine di raggio 1 ed altezza 2; il pavimento è un cono retto che intrude nella stanza (la quale quindi non è convessa), di altezza pari al raggio. Il cono è uno specchio ideale, mentre tutti gli altri muri sono opachi. Consideriamo dentro la stanza punti $p = (x_p, y_p, z_p)$ con x_p e $y_p \neq 0$. L'asse z è quello ortogonale al soffitto. Un osservatore è situato al centro del soffitto (rammentiamo che il soffitto è ad altezza 2). Quali sono le coordinate dei punti p interni alla stanza che l'osservatore può vedere riflessi nel cono? *(Suggerimento: trasformare il problema in un problema equivalente bidimensionale. Per risolverlo, cominciare a determinare quali punti della stanza l'osservatore vede riflessi quando guarda:*

- *quasi verso il vertice del cono, ossia il più in alto possibile (vicino al vertice ma non proprio al vertice perché lì la normale cambia e si ha un punto con direzione riflessa eccezionale, ma un solo punto è irrilevante per il rendering)*
- *il più in basso possibile, ossia punti del cono vicino alla sua base (di nuovo, vicino alla base ma non proprio alla base, perché sui punti della base la normale cambia, ma si tratta di una curva eccezionale e non una superficie, quindi irrilevante per il rendering)*

Poi ci si dovrebbe chiedere se l'angolo di riflessione varia in maniera monotòna quando l'osservatore fa variare il punto osservato fra questi due estremi, e dedurne quali punti vede.

2. Per ciascun tale punto p, verso quale punto del cono l'osservatore deve guardare per vederne il riflesso? *(Suggerimento: parametrizzare il cono e scrivere per ciascun suo punto q il vettore normale ed il raggio riflesso del raggio che unisce l'osservatore a q; tracciare questo raggio riflesso e vedere per quale scelta di q esso passa per p. È molto conveniente trasformare il problema in un problema equivalente bidimensionale.)*

□

5.6. Mappe d'ombra

Le mappe d'ombra consistono in scorciatoie nelle quali si calcola separatamente la distribuzione di ombre di ogni sorgente di luce, e dopo il Ray Tracing la si usa per dipingere le aree in ombra con appropriati toni di grigi. Se le ombre si dipingono con toni di grigio parziali anziché di nero, ossia se si usano per attenuare la luminosità, questi metodi sono viziati da errore, perché eventuali riflessi di luce, calcolati dal Ray Tracing in base alla posizione di una sorgente di luce, vengono attenuati nelle aree dove quella sorgente è in ombra ma non eliminati del tutto. Vedremo però un metodo che, nella sua forma più onerosa, non è affetto da questo errore: il metodo del doppio $z-$buffer di Williams.

5.6.1. Metodo del doppio $z-$buffer di Williams. Il metodo della mappa d'ombra (shadow map, o $z-$buffer delle ombre) è un procedimento a precisione di immagine per il rendering delle ombre, introdotto in **[53]**. Il procedimento si basa sul confronto fra due $z-$buffer, quello consueto centrato sull'osservatore ed un altro centrato sulla sorgente di luce (se ci sono più sorgenti si usano ulteriori $z-$buffer). Anzitutto si calcola lo $z-$buffer dal punto di vista della sorgente di luce, che serve a determinare quali punti della scena sono direttamente visibili dalla sorgente, e quindi in luce. Poi si calcola lo $z-$buffer dal punto di vista dell'osservatore, ma nella variante seguente. Per ogni pixel, dopo aver determinato quale poligono della scena è visibile dall'osservatore, si trovano le coordinate tridimensionali del punto osservato (nel sistema di riferimento dell'osservatore: chiamiamole (x_o, y_o, z_o)). Queste coordinate vengono trasformate grazie alla trasformazione prospettica che sposta l'osservatore nella sorgente. Scriviamo le coordinate così trasformate (x'_o, y'_o, z'_o): esse sono ora coordinate nel sistema di riferimento centrato sulla sorgente di luce. Pertanto le prime due componenti identificano il pixel dello $z-$buffer centrato sulla luce attraverso cui passa il raggio proiettore che passa più vicino al punto della scena osservato: quindi permettono di recuperare il valore dello $z-$buffer della sorgente a quel pixel (chiamiamolo z_l). La terza coordinata, z'_o, misura la profondità (e quindi la distanza) del punto osservato rispetto alla posizione della sorgente. Invece z_l misura la distanza dalla sorgente del punto della scena ad essa più vicino lungo il raggio proiettore che attraversa quel pixel. PerciĂČ se $z_l > z'_o$ (ossia il punto illuminato è più prominente verso la sorgente rispetto al punto guardato dall'osservatore) qualche altro poligono si frappone fra la sorgente ed il punto osservato dall'osservatore: quindi tale punto è in ombra, ed allora nello $z-$buffer dell'osservatore si memorizza non il colore pieno del poligono illuminato, ma quello più scuro del poligono in ombra, che si calcola, ovviamente, a partire dall'equazione dell'illuminazione (se c'è una sola sorgente il valore memorizzato a questo pixel è quello che deriva dalla sola illuminazione ambientale). Qui stiamo assumendo che tutti i poligoni siano opachi: il caso di poligono trasparenti

può essere trattato qui con uno $z-$buffer multiplo, analogamente alla procedura di Mammen per il rendering della trasparenza non rifrattiva, che verrà presentata nella Sottosezione 5.7.3.
In altre parole, lo $z-$buffer centrato sulla sorgente è uno $z-$buffer delle ombre. Si osservi che il punto osservato non giace di solito esattamente sul raggio proiettore che passa per il centro del pixel della luce, quindi il valore z_l si ricava tramite interpolazione a partire dai valori circostanti dello $z-$buffer della luce: se ne fa una media pesata in cui i pesi sono le combinazioni convesse che esprimono (x'_o, y'_o) in termini delle coordinate di questo buffer. In tal modo si attua una approssimazione, che è accurata purché non ci siano troppi punti con coordinate vicine o uguali a (x'_o, y'_o). Infatti, tutti questi punti risulterebbero con lo stesso valore di z_l , anche quando le profondità effettive z'_o sono molto diverse: in tal caso un oggetto interposto potrebbe coprire alcuni di questi punti ma non altri, e per assicurare una risoluzione adeguata sarebbe necessario un sovracampionamento.

Questo procedimento, come tutti quelli basati su $z-$buffer, è a precisione di immagine. Esso risente pesantemente degli inerenti arrotondamenti numerici ad esso inerenti: infatti si basa in maniera cruciale sul confronto fra i due valori z_l e z'_o, entrambi soggetti ad errori di arrotondamento, ed il secondo in misura ancora maggiore perché ottenuto attraverso una trasformazione matriciale che richiede calcoli aritmetici. Se valori di z_l e z'_o sono vicini ed in conseguenza di tali arrotondamenti si scavalcano, allora un punto che avrebbe dovuto apparire illuminato appare in ombra o viceversa, generando una aberrazione evidente. Per ridurre questo effetto di aliasing si deve, come sempre, effettuare filtraggio, ed in questo caso si puo' usare anche dithering. Invece non è possibile in questo metodo applicare un sovrarcampionamento a passo variabile. Per vederlo, riesaminiamo con maggiore attenzione l'osservazione che abbiamo fatto poco fa circa la necessità di un sovracampionamento dove la risoluzione è insufficiente. Il metodo si basa su una trasformazione tridimensionale di coordinate che sposta il centro di prospettiva dalla posizione dell'osservatore a quella della sorgente di luce. I punti dello spazio da trasformare sono i punti della scena visti dall'osservatore tracciando raggi proiettori attraverso il centro dei pixel del viewport. Si tratta di un fascio di raggi uscenti dalla posizione dell'osservatore ed angolarmente equispaziati. Però i punti della scena che essi intersecano potrebbero non essere equidistribuiti nello spazio, ed ancora meno probabile è che lo siano dopo la proiezione sul rettangolo di pixel disposto di fronte alla posizione della luce che dobbiamo usare per costruire lo $z-$buffer della luce (questo ad esempio accade se poligoni diversi della scena che sottendono la stessa area quando proiettati sul viewport dell'osservatore coprono due aree molto diverse fra loro quando proiettati sul viewport della luce). In tali casi alcuni punti dopo la trasformazione risultano fittamente distribuiti sul viewport della luce ed altri risultano molto sparsi. Molti punti diversi possono cadere nello stesso pixel del viewport della luce, ed in altri pixel possono non caderne nessuno. Il campionamento fisso dato dai pixel del viewport della luce è allora sufficiente per calcolare uno $z-$buffer preciso nelle aree dove i punti da trasformare sono distribuiti in modo sparso, ma puĂČ essere molto insufficiente per una adeguata precisione nelle aree con punti da trasformare sono fittamente distribuiti. Una versione a passo variabile del campionamento richiede di infittire i raggi proiettori nelle aree in cui le variazioni di intensità di colore sono elevate, ma qui i punti da trasformare sono determinati dalle intersezioni con la scena dei proiettori attraverso i centri dei pixel di un altro viewport, quello dell'osservatore: quindi l'unico modo di infittire i proiettori in modo variabile è di tracciare raggi più fitti attraverso i pixel di questo viewport, nella fase iniziale prima della trasformazione: ma questo deve avvenire laddove i punti di intersezione sono più fitti quando visti dalla posizione della luce, ovvero dopo la trasformazione. Quali siano le zone della scena in cui dovrebbero arrivare più raggi proiettori si puĂČ solo sapere dopo la trasformazione, e si puĂČ solo determinare tracciando effettivamente raggi nello spazio per determinare le posizioni tridimensionali dei punti di intersezione, poi trasformare questi punti ed infine ritracciare i raggi a ritroso, un procedimento assai oneroso dal punto di vista numerico.

Come sempre con lo $z-$buffer, ogni poligono deve essere processato dal metodo, anche se dopo viene ricoperto da un altro. Però in questo caso questa ridondanza produce un dispendio di tempo

di calcolo molto maggiore che con il solo $z-$buffer, perché per ogni pixel, oltre che la scansione necessaria per lo $z-$buffer, ora si deve anche effettuare una trasformazione di coordinate. Per ridurre la mole di calcolo Williams ha proposto la variante seguente: calcolare prima lo $z-$buffer consueto dal punto di vista dell'osservatore, poi per ciascun pixel trasformare le coordinate del punto della scena che lo $z-$buffer individua come visibile all'osservatore e confrontare i valori di z come prima. Se $z_l < z'_o$ allora al colore del pixel determinato dallo $z-$buffer dell'osservatore viene aggiunto un contributo di colore scuro di ombra (viene 'dipinto d'ombra'). Questa variante del procedimento è molto più veloce, perché per ogni pixel dello $z-$buffer dell'osservatore viene trasformato solo il punto della scena visibile attraverso il pixel, e non tutti i punti intersecati dal raggio proiettore (uno per ciascun poligono man mano che si svolge la scansione); però è meno precisa, perché (come preaccennato all'inizio di questa Sezione) il primo passaggio di $z-$buffer determina l'ombreggiatura senza le ombre, e quindi determina anche gli incrementi di intensità di illuminazione dovuti ai contributi della riflessione speculare: laddove queste zone riflettenti chiare vengono dipinte d'ombra, la loro brillantezza viene attenuata, ma non del tutto annullata, come invece dovrebbe accadere dal momento che, essendo in ombra, non riflettono la luce. Però, per scene che consistono di superficie prevalentemente opache e poco riflettenti, questa variante è vantaggiosa.

5.6.2. Esercizi sul metodo del doppio $z-$buffer di Williams.

Esercizio 5.6.1. Una scena si compone di un triangolo equilatero E con vertici in (1,0,0), (0,1,0) e (0,0,1) ed un triangolo T con vertici in (1,0,0), (0,1,0) e (1,0, 1/2). L'osservatore è disposto lungo l'asse z positivo, all'infinito. C'è una sola sorgente di luce, all'infinito lungo l'asse x positivo: quindi una luce direzionale che emette un fascio di raggi paralleli all'asse x. Lo $z-$buffer dell'osservatore si basa su un viewport dato da una griglia di pixel quadrati di lato $1/4$ sul piano $z = 1$, con il pixel centrale che ha centro sull'asse z. Lo $z-$buffer della luce si basa su un viewport dato da una griglia di quadratini di lato $1/4$ disposti sul piano $x = 1$, con quello centrale che ha centro sull'asse x.

1. Si calcoli lo $z-$buffer dell'osservatore nel suo pixel centrale (quale triangolo ivi proiettato sta più avanti verso l'osservatore e che profondità spaziale z_o esso ha su quel pixel). Quali sono le tre coordinate spaziali (x_o, y_o, z_o) del punto $\boldsymbol{p}$ osservato?
2. Si trasformino le coordinate in maniera che la posizione dell'osservatore diventi la posizione $\boldsymbol{l}$ della sorgente, l'asse z' diventi quello che prima era l'asse x ma con verso opposto, l'asse y' diventi quello che prima era l'asse z ma con verso opposto, l'asse x resti inalterato e la nuova origine diventi il punto $(x = 0, y = 0, z = 0)$. Mettiamo il viewplane della luce in modo che coincida con il trasformato sotto questa trasformazione di coordinate di quello che era il viewplane dell'osservatore: il nuovo pixel centrale giace quindi sul piano $z' = 0$ ed ha centro in $(x', y', z') = (0, 0, 0)$. Il centro del vecchio pixel centrale pertanto ha coordinate $(x', y', z') = (0, 1, 1)$. Si scrivano le coordinate dei vertici dei due triangoli in queste nuove coordinate.
3. Si calcolino le nuove coordinate (x'_o, y'_o, z'_o) del punto $\boldsymbol{p}$ dopo la trasformazione.
4. Si determini quale è il pixel (ossia quali sono le sue coordinate centrali (x'_o, y'_o)) che corrisponde al punto $\boldsymbol{p}$ sullo $z-$buffer della luce.
5. Si trovi il valore della profondità z dello $z-$buffer della luce al pixel di coordinate centrali (x'_o, y'_o).
6. Da questi dati, si decide se il punto $\boldsymbol{p}$ è in luce o in ombra.

Si verifichi che le risposte non cambiano se l'osservatore è situato a distanza finita, ad esempio in $\boldsymbol{o} = (0, 0, 2)$.

Svolgimento.

1. L'osservatore guarda perpendicolarmente al piano $\{z = 1\}$, quindi direttamente verso il basso. La proiezione sul suo viewport del triangolo T, che è disposto verticalmente, è il

segmento da (1,0) a (0,1), ottenuto semplicemente scartando la coordinata z (proiezione ortogonale sull'asse z, dal momento che l'osservatore sta all'infinito); questa proiezione non passa per il pixel centrale in $(0, 0)$. Invece la proiezione del triangolo obliquo E, ottenuta nello stesso modo, è il triangolo di vertici (x, y)= (1, 0), (0, 1) e (0, 0) sul piano z=1: questa proiezione copre il pixel centrale ubicato alle coordinate (x, y) = (0, 0). Il corrispondente punto tridimensionale $\boldsymbol{p}$ è il vertice (0, 0, 1) di E, e quindi il valore di profondità spaziale z_o vale 1. Si osservi che, se l'osservatore stesse a distanza finita, ad esempio in O=(0, 0, 2), allora si dovrebbe eseguire una proiezione centrale su questo punto invece che una proiezione ortogonale, e quindi la proiezione del triangolo T diventerebbe un triangolo invece che un segmento, ma chiaramente non coprirebbe il punto (0,0): quindi le risposte non cambierebbero.

2. La luce è disposta lungo l'asse x: quindi nel suo sistema di coordinate dobbiamo portare z in x, ad esempio scambiando z e x, ma per avere una terna destrorsa occorre anche riflettere y in $-y$. Questo porta alla trasformazione

$$x \mapsto z, \qquad z' \mapsto x, \qquad y' \mapsto -y.$$

Nota: se la luce fosse ad un punto $\boldsymbol{l}$ a distanza finita sarebbe inoltre elegante (ma non indispensabile) effettuare una traslazione per spostare l'origine al punto $\boldsymbol{l}$; ad esempio, se fosse $\boldsymbol{l} = (2, 0, 0)$, che giace sull'asse x originale, dopo la trasformazione $\boldsymbol{l}$ si troverebbe sul nuovo asse z in posizione 2. Quindi la trasformazione diventerebbe:

$$x' = z, \qquad y' = -y, \qquad z' = x - 2.$$

Ma in questo esercizio, invece, $\boldsymbol{l}$ si trova a distanza infinita e quindi non eseguiamo la traslazione.

I vertici di E ora diventano $(x', y', z') = (0, 0, 1)$, $(0, -1, 0)$ e $(1, 0, 0)$; quelli di T sono $(x', y', z') = (0, 0, 1)$, $(0, -1, 0)$ e $(1/2, 0, 1)$.

3. Poiché le coordinate originali di $\boldsymbol{p}$ sono $(x_o, y_o, z_o) = (0, 0, 1)$, le sue coordinate trasformate sono $(x'_o, y'_o, z'_o) = (1, 0, 0)$. Quindi:
4. Questo punto si proietta sul pixel dello $z-$buffer della luce di coordinate centrali $(x'_o, y'_o) = (1, 0)$.
5. Ovviamente, per il precedente punto (4), sul pixel di coordinate centrali $(x'_o, y'_o) = (1, 0)$ la proiezione di E proviene dal punto P, ossia $(x'_o, y'_o, z'_o) = (1, 0, 0)$: questo fornisce un valore di profondità $z' = 0$. Nelle nuove coordinate, invece, il triangolo verticale T, che per il punto (2) ha vertici $(x', y', z') = (0, 0, 1)$, $(0, -1, 0)$ e $('$, $0, 1)$, ha proiezione sul viewport della luce (x = 1, ovvero z' = 1) data dal triangolo proiettato i cui vertici si ottengono, come prima, scartando la terza coordinata z', e quindi sono $(x', y') = (0, 0)$, $(0, -1)$ e $(1/2, 0)$. Si vede facilmente che il punto $(x'_o, y'_o) = (1, 0)$ è esterno a tale triangolo (infatti il massimo valore di x' per i tre vertici vale $1/2$, e quindi per convessità questo è anche il massimo valore di x' nell'intero triangolo proiettato T, che pertanto non raggiunge mai il valore $x' = 1/2$). Pertanto nello $z-$buffer della luce, in corrispondenza del pixel $(x', y') = (0, 0)$ individuato dalla posizione di $\boldsymbol{p}$, il valore z' è proprio il valore della profondità di $\boldsymbol{p}$, ossia $z' = z'_o = 0$.
6. Il valore z' nello $z-$buffer della luce al pixel (x', y') corrispondente al punto osservato $\boldsymbol{p}$ è proprio il valore di profondità z'_o di $\boldsymbol{p}$. Quindi niente si frappone fra la sorgente di luce ed il punto $\boldsymbol{p}$, e pertanto $\boldsymbol{p}$ è illuminato.

□

Esercizio 5.6.2. Nella stessa scena del problema precedente, adesso la sorgente di luce si trova sull'asse x a distanza finita, diciamo in $\boldsymbol{l} = (2, 0, 0)$. Come cambiano le risposte?

Svolgimento. Ora è più elegante effettuare la trasformazione di coordinate comprensiva della traslazione spiegata nella risposta al punto (2) dell'esercizio precedente (ma non è indispensabile): usiamo queste nuove coordinate. La proiezione sul viewport dello $z-$buffer della luce ora diventa una proiezione centrale, con centro in $(x = 2, y = 0, z = 0)$. Si vede subito che la proiezione centrale del punto $\boldsymbol{p} = (1, 0, 0)$ su questo viewport è il punto $\boldsymbol{q} = (1, 0, 1/2)$ ottenuto per similitudine dei triangoli, dal momento che il piano del viewport è proprio a metà fra $\boldsymbol{p}$ e $\boldsymbol{l}$. La sua profondità nelle coordinate trasformate è $z' = -2$. Quindi il pixel del viewport della luce che occorre considerare è quello che contiene $\boldsymbol{q}$. Ma il punto $\boldsymbol{q}$, che giace sul viewport della luce, è anche il vertice alto del triangolo T : esso ha coordinate trasformate $(x', y', z') = (1/2, 0, -1)$, e quindi profondità $z' = -1$, come ovvio perché questa è la profondità del piano del viewport stesso. Quindi il valore di $z-$buffer è $z' = -1$ invece che $z' = -2$ di $\boldsymbol{p}$: pertanto $\boldsymbol{p}$ è in ombra, ma appena appena: sarebbe bastato spostare $\boldsymbol{l}$ indietro di una quantità arbitrariamente piccola e $\boldsymbol{p}$ sarebbe stato in luce. □

Esercizio 5.6.3. Nella stessa scena del problema precedente, adesso la sorgente di luce si trova a distanza 4 dall'origine lungo la bisettrice dell'ottante positivo. Poniamo il viewport della luce a distanza 2 dall'origine, con asse x parallelo al piano di base. Si risponda alle stesse domande.

Suggerimento: Nel calcolo della trasformazione, ora occorre portare l'asse z nella direzione della suddetta bisettrice: quindi la trasformazione manda $(0, 0, 1)$ in $\frac{1}{\sqrt{3}}\ (1, 1, 1)$. Gli assi x e y vanno in direzioni trasversali a tale bisettrice: l'asse x è parallelo al piano di base, e quindi il punto $(1, 0, 0)$ va nel versore di terza coordinata 0 e prime due coordinate (ossia proiezione sul piano di base) perpendicolari alla proiezione della bisettrice, ossia al vettore $(1, 1)$: scegliamo di mandare $(1, 0, 0)$ nel versore $\frac{1}{\sqrt{2}}\ (1, -1, 0)$. Il terzo vettore della base canonica iniziale va nel prodotto vettore di questi due: è ovvio che la sua proiezione sul piano di base deve essere perpendicolare a quella del versore precedente, e quindi diretta come quella della bisettrice, ossia multipla di $(1, 1)$; invece la terza coordinata deve formare un angolo di $\pi/2$ con la suddetta bisettrice, quindi il versore è diretto lungo la bisettrice dell'ottante con x e y negativi e z positivo. Pertanto esso è multiplo di $(-1, -1, 1)$, ossia è il versore $\pm\frac{1}{\sqrt{2}}\ (1, 1, 2)$ (*cautela:* il verso dovrebbe essere scelto in modo che la terna sia destrorsa!). Alternativamente, tutte queste considerazioni geometriche si possono sviluppare in termini trigonometrici mediante gli angoli di Eulero, ossia esprimendo le coordinate come

$$x = r\cos\theta\cos\phi, \qquad y = r\sin\theta\cos\phi, \qquad z = r\sin\phi.$$

A questo punto non è difficile scrivere la trasformazione di coordinate (si rammenti che la posizione della luce è sì lungo la bisettrice, ma a distanza 4 dall'origine, ossia nel punto $\frac{4}{\sqrt{3}}\ (1, 1, 1)$: non si dimentichi, nella trasformazione di coordinate, la necessaria traslazione che sposta l'origine su questo punto!).
Il resto dell'esercizio si sviluppa come nell'esercizio precedente. □

Esercizio 5.6.4. Nella stessa scena dei problemi precedenti, adesso la sorgente di luce si trova a distanza infinita dall'origine lungo l'asse x positivo, ma l'osservatore è ubicato lungo la bisettrice del primo ottante, all'infinito. Lo $z-$buffer dell'osservatore si basa su un viewport dato da una griglia di pixel quadrati di lato $1/4$ sul piano perpendicolare alla bisettrice a distanza 1 dall'origine, con il pixel centrale centrato sulla bisettrice. Come cambiano le risposte circa il fatto che il punto che l'osservatore vede al centro del suo pixel centrale sia in luce od in ombra' □

Esercizio 5.6.5. La scena consiste di un cubo con centro l'origine e lato 2, dalle pareti perfettamente speculari, ed una sfera S con centro l'origine e raggio 10, dipinta con i meridiani ed i paralleli, come un mappamondo. L'osservatore si trova al punto (2,2,2), la sorgente di luce in (8,0,0). Si determini se l'osservatore vede il centro della faccia $x = 1$ in luce o in ombra, mediante il meccanismo di trasformazione e confronto dell'algoritmo di Williams, ma (per evitare di calcolare a mano i

due $z-$buffer per ogni pixel) qui modificato usando il tracciamento di raggi invece che lo $z-$buffer (ovviamente seguito dalla appropriata trasformazione di coordinate e dal confronto). □

Esercizio 5.6.6. Una scena consiste di un pavimento infinito sul piano $\{z = 0\}$ con una sfera di raggio 1 appoggiata all'origine, ed un disco D sul piano $\{z = 3\}$ di raggio 1 e centro in (1,1,3). L'osservatore si trova al punto (0,0,6), la sorgente di luce al punto (5, 5, 5). Come procede il metodo di Williams a determinare se il punto che l'osservatore vede guardando in basso (ossia nella direzione del versore dell'asse z diretto verso la parte negativa di tale asse) è in luce o in ombra? Si svolgano i calcoli.

Svolgimento. L'osservatore si trova sull'asse z, al punto di altezza 6. Consideriamo lo $z-$buffer dell'osservatore: possiamo supporre che esso sia costruito su un quadrato ad esempio nel piano $\{z = 5\}$, centrato in (0,0,5). L'osservatore guarda verso il basso, quindi nella direzione z negativa, quindi verso l'origine, e pertanto guarda verso il pixel centrale di questo $z-$buffer. Calcoliamo il contenuto del buffer. Per similitudine, si vede che la proiezione centrale sul piano $\{z = 5\}$ del disco D, con punto di fuga nella posizione dell'osservatore (0,0,6), è il disco di centro (1/3, 1/3, 1) e raggio 1/3. Questo disco non contiene l'origine, perché ha raggio 1/3 ma il suo centro dista dall'origine $\sqrt{2}/3 > 1/3$. Invece la proiezione centrale della sfera (il cui disco equatoriale orizzontale è ad altezza $z = 1$) sul piano del buffer $\{z = 5\}$ è contenuta nella proiezione del suo disco equatoriale, ossia del disco di centro (0,0) e raggio 1, che è il disco orizzontale ad altezza $z = 5$ di centro (0,0). Questo disco ovviamente contiene l'origine, e quindi l'osservatore, che sta guardando verso l'origine, nella direzione di osservazione vede la sfera, la quale in quel punto (il suo polo nord) ha altezza (ovvero profondità z) uguale a 2. Il punto osservato, quindi, è (0,0,2).
Ora cambiamo coordinate per centrarle sulla sorgente di luce, al punto (5,5,5). Basiamo lo $z-$buffer della luce su un quadrato perpendicolare al segmento da (0,0,2) a (5,5,5): allora il punto osservato si proietta sul centro dello $z-$buffer della luce, ossia (4,4,4). Si tratta di verificare se tale segmento interseca il disco D. Ma D giace sul piano $\{z = 3\}$. Il segmento sta sulla retta

$$\boldsymbol{r}(t) = (0, 0, 2) + t(5, 5, 3) = (5t, 5t, 2 + 3t).$$

Questa retta interseca il piano $\{z = 3\}$ per $t = 2/3$, e quindi nel punto $\boldsymbol{q} = (x, y, z) = (10/3, 10/3, 3)$. Il disco D ha centro nel punto $(x = 1, y = 1)$ e raggio 1. Quindi la condizione che $\boldsymbol{q}$ sia interno a D è che si abbia

$$1 \geqslant \left(\frac{10}{3} - 1\right)^2 + \left(\frac{10}{3} - 1\right)^2 = 2\left(\frac{7}{3}\right)^2.$$

Ovviamente ciĂČ è falso, e quindi il disco non si frappone fra la posizione della luce e quella del punto osservato: perciĂČ tale punto è in luce.
Nota: abbiamo usato uno $z-$buffer perpendicolare al segmento dal punto osservato alla sorgente di luce. Questo rende facile la soluzione ma non è molto corretto: qui stiamo solo interessandoci all'unico punto osservato, ma in un caso di rendering vero i punti osservati sono tanti e lo $z-$buffer della luce uno solo, non lo possiamo spostare quando osserviamo un altro punto, e quindi la maggior parte dei punti osservati cade fuori dal centro dello $z-$buffer della luce.In effetti, abbiamo risolto il problema con un Ray Tracing a partire dalla posizione della luce invece che con uno $z-$buffer.
Recependo l'osservazione in questa nota, ripetiamo il calcolo basandolo solo su $z-$buffer. Costruiamo lo $z-$buffer della luce su un quadrato perpendicolare al segmento, diciamo, dall'origine a (5,5,5), ossia alla bisettrice del primo ottante: ad esempio il piano ortogonale a tale bisettrice che la interseca in (4,4,4) (si tratta del piano π di equazione $x + y + z = 4$). Eseguiamo la proiezione centrale di disco e sfera sul piano dello $z-$buffer della luce, con punto di fuga nella posizione $\boldsymbol{l} = (5, 5, 5)$ della luce. A questo scopo, osserviamo che la proiezione ortogonale di un punto $\boldsymbol{r} = (x_0\,, y_0\,, z_0)$ sul piano $x + y + z = 4$ si ottiene componendo $\boldsymbol{r} = \boldsymbol{r}_d + \boldsymbol{r}_n$, dove $\boldsymbol{r}_d$ è la componente lungo la diagonale e $\boldsymbol{r}_n$ è quella trasversale. Calcoliamo $\boldsymbol{r}_d$. Sia $d_0 = x_0 + y_0 + z_0$. Il punto $\boldsymbol{r} = (x_0\,, y_0\,, z_0)$ giace sul piano

parallelo a π di equazione $x+y+z = d_0$, quindi la sua proiezione sulla bisettrice è $\boldsymbol{r}_d = (d_0\,, d_0\,, d_0)/3$. Pertanto $\boldsymbol{r}_n = (x_0 - d_0/3, y_0 - d_0/3, z_0 - d_0/3)$, un punto del piano π_0 parallelo a π che passa per l'origine, e la proiezione centrale di $\boldsymbol{r} = \boldsymbol{r}_d + \boldsymbol{r}_n$ sul piano π è $\boldsymbol{r} = 4/5\boldsymbol{l} + \boldsymbol{r}_n/(5 - d_0)$ (di nuovo per similitudine, perché il piano di proiezione si trova a distanza $4\sqrt{3}$ dall'origine, mentre il punto $\boldsymbol{r}_d$ è a distanza $\sqrt{3}d_0$ ed il punto di fuga $\boldsymbol{l}$ è a distanza $4\sqrt{3}$: quindi la compressione di scala è di un fattore$(5 - 4)/(5 - d0_) = 1/(5 - d0)$).

Quindi la componente trasversale $\boldsymbol{r}_n$ si proietta sul piano π nel punto $4/5\boldsymbol{l}+\boldsymbol{r}_n/(5-d_0$: per vedere il punto $\boldsymbol{r} = (x_0\,, y_0\,, z_0)$ dalla ubicazione della sorgente di luce bisogna guardare nella direzione data dal pixel in posizione $4/5\boldsymbol{l} + \boldsymbol{r}_n/(5 - d0)$ nello $z-$buffer della luce. Ora concentriamo la nostra attenzione al punto $\boldsymbol{r} = (0, 0, 2)$ che l'osservatore sta osservando: allora $d_0 = 2$, $\boldsymbol{r}_n = (-2/3, -2/3, 4/3)$ e $4/5\boldsymbol{l} + \boldsymbol{r}_n/(5 - d0)$ dista dal centro dello $z-$buffer della luce (che è il punto $4/5\boldsymbol{l} = (4, 4, 4)$) una distanza pari a

$$\frac{\}\boldsymbol{r}_n\|}{5 - d_0} = \frac{5}{9}\,.$$

Invece, il disco D ha per bordo la circonferenza $(x - 1)^2 + (y - 1)^2 = 1$, $z = 3$. Scriviamo i punti del bordo di D come $\boldsymbol{p} = (x, y, z)$. Procedendo come prima, vediamo che $\boldsymbol{p} = \boldsymbol{p}_d + \boldsymbol{p}_n$ con $\boldsymbol{p}_d = (x + y + z)(1/3, 1/3, 1/3)$ e $\boldsymbol{p}_n = (2x - y - z, 2y - x - z, 2z - x - y)/3$, e che i punti del disco hanno proiezione centrale sul piano π nei punti $4/5\boldsymbol{l} + \boldsymbol{p}_n/(5-x-y-z) = (4, 4, 4) + (2x-y-z, 2y-x-z, 2z-x-y)/(3/(5-x-y-z))$. La deviazione laterale rispetto al centro (4,4,4) dello $z-$buffer è il vettore (2x-y-z, 2y-x-z, 2z-x-y)/(3/(5-x-y-z)). Si tratta di verificare se la norma di questo vettore è inferiore a $5/9$ allorché $z = 3$ e $(x-1)2+(y-1)2 = 1$. Per $z = 3$ il vettore di deviazione dal centro dello $z-$buffer diventa $(2x - y - 3, 2y - x - 3, 6 - x - y)/(3/(5 - x - y - 3))$. La disuguaglianza da verificare è quindi una disequazione quadratica in x e y. Per ogni x dobbiamo verificare che non ci sono soluzioni reali in y. Lasciamo al lettore i tediosi calcoli. □

5.6.3. Cenni su mappa di occlusione. Il metodo moderno di dipingere le ombre tramite mappe d'ombra è la mappa di occlusione, che qui, per completezza ci limitiamo a citarla senza descriverla. Per i dettagli rinviamo il lettore alla successiva Sottosezione 6.13.3.

5.7. Accelerazione del rendering della trasparenza evitando il tracciamento di raggi rifratti

Il rendering della trasparenza con un procedimento che non tiene conto delle leggi della rifrazione ha senso solo quando le superficie trasparenti sono piane e sottili, e non si è interessati a rendere la separazione dei colori dovuta al fatto che la rifrazione provoca deviazioni angolari differenti a colori monocromatici di diverse lunghezze d'onda. Se le superficie sono costituite da lastre trasparenti piane, ogni raggi che le attraversa ne esce su una retta parallela alla direzione di ingresso (incidente), quindi con una deflessione ma senza una deviazione angolare. Se la lastra è sottile la deflessione è piccola, e spesso trascurabile nella resa di una scena. Quindi ha senso trascurarla, e pertanto trascurare l'effetto della rifrazione. Invece bisogna rendere l'attenuazione dovuta alla perdita di energia nell'attraversamento della lastra, e la colorazione che si ottiene se la lastra è colorata. La modellazione empirica di questi fenomeni, senza ricorso alle leggi dell'ottica relative alla rifrazione ed alla riflessione, si chiama la modellazione della trasparenza non rifrattiva.

Ci sono due diverse modellazioni della trasparenza non rifrattiva: interpolata oppure filtrata.

5.7.1. Trasparenza interpolata. La trasparenza interpolata modella la situazione in cui una superficie frontale (assumeremo per semplificare la terminologia che si tratti di un poligono) semitrasparente, che denotiamo come poligono 1, ne copre (in tutto od in parte) una posteriore, che denotiamo con 2. Il modello assume che i colori dei due poligoni si mescolino tramite una appropriata combinazione convessa (cioè un'interpolazione) dei loro valori: l'equazione risultante per

l'illuminazione è, per ogni lunghezza d'onda λ,

$$I_\lambda = (1 - k_{t1})I_{\lambda 1} + k_{t1}I_{\lambda 2}\,,$$

dove il coefficiente di trasmissione k_{t1} è la percentuale di luce che il poligono 1 lascia passare per trasparenza, e varia fra 0 e 1: quando vale 0 il poligono è completamente opaco, quando vale 1 esso è completamente trasparente e non contribuisce in niente al colore risultante. Questa modellazione è accurata nel caso si stia rendendo un poligono posteriore al quale si antepone una griglia sottile colorata, in modo che il colore risultante, pixel per pixel, sia una media del colore della griglia e di quello del poligono retrostante, con coefficienti di interpolazione che variano a secondo di quanto è fitta la griglia.

5.7.2. Trasparenza filtrata. La trasparenza filtrata è il modello per la situazione in cui la superficie (poligono) anteriore è un filtro semitrasparente che ha il suo proprio colore, ed inoltra colora (con un colore magari diverso) una superficie (poligono) posteriore. Al variare della lunghezza d'onda λ, l'equazione dell'illuminazione diventa

$$I_\lambda = I_{\lambda 1} + k_{t1}O_{t\lambda 1}I_{\lambda 2}\,,$$

dove il coefficiente di trasmissione k_{t1} è definito come per la trasparenza interpolata, e $O_{t\lambda 1}$ è il colore di trasparenza del poligono frontale, cioè il colore che esso dà a ciĂČ che sta dietro. Questo modello è di solito più realistico della trasparenza interpolata.

Per entrambi i modelli, se più superficie semitrasparenti insistono sullo stesso pixel, esse vanno rese dal dietro in avanti ricorsivamente (o iterativamente) perché le formule si compongano nel modo giusto.

In prossimità dei bordi di una superficie trasparente curva la modellazione finora esposta è meno soddisfacente, non per colpa degli algoritmi, ma a causa del fatto che la modellazione dei filtri come sottili lastre piane di vetro non è sufficientemente precisa, perché in quelle zone il raggio proiettore attraversa uno spessore di vetro maggiore, a causa della curvatura, e quindi la luce si attenua di più. Una formula empirica non lineare per modellare questo effetto è proposta in **[28]**. In questa formula, il coefficiente di trasmissione k_t viene espresso come un opportuno valore intermedio fra i coefficienti massimo e minimo della superficie, che che chiamiamo $k_{t\ \min}$ e $k_{t\ \max}$ e che si ottengono nei punti di spessore minimo o massimo, rispettivamente. Il valore intermedio è determinato da una espressione non lineare che involve la componente z_n nella direzione della profondità del versore normale alla superficie (dopo la trasformazione prospettica). La formula è la seguente:

$$k_t = k_{t\ \max} + (k_{t\ \min} - k_{t\ \max})[1 - (1 - z_n)^m].$$

Qui m è un esponente empirico che dipende dal materiale di cui è composta la superficie, e di solito ha valore fra 2 e 3. Più la superficie è sottile, più m è grande.

5.7.3. Accelerazione del rendering della trasparenza non rifrattiva tramite $z-$buffer: il metodo dello $z-$buffer multiplo. Si tratta di un procedimento a precisione di immagine per il rendering della trasparenza filtrata, introdotto da A. Mammen **[31]**. Abbiamo visto che, per il rendering della trasparenza filtrata, i poligoni nella pila che insiste sullo stesso pixel deve essere trattati dal dietro in avanti. Il metodo di Mammen perviene all'ordinamento corretto utilizzando vari $z-$buffer. Dapprima si effettua lo $z-$buffer consueto per trattare tutti i poligoni opachi (e solo quelli). Poi vengono scanditi i poligoni trasparenti in vari $z-$buffer, a piu' passi, nel modo seguente. Lo $z-$buffer viene modificato per aggiungervi, oltre al colore del poligono visibile a ciascun pixel ed alla sua profondità, anche il valore di trasparenza ed un bit logico (una flag), inizialmente posto uguale a falso. A differenza dallo $z-$buffer solito, qui la variabile di profondità viene inizializzata al valore più alto possibile, cioè quello più vicino all'osservatore (la profondità del viewport, il piano di visuale). La memorizzazione della profondità in questo $z-$buffer dei poligoni trasparenti avviene nel modo seguente: durante la scansione di un poligono trasparente, se ne memorizza la profondità

ad un pixel solo se essa è maggiore (poligono trasparente più vicino all'osservatore) di quella del poligono opaco più vicino (che viene letta nello $z-$buffer dei poligoni opachi), ma minore poligono trasparente più lontano) di tutti gli altri poligoni trasparenti a quel pixel. Oltre alla profondità si memorizzano il colore ed il valore di trasparenza, e si pone true la flag. Al termine della scansione di tutti i poligoni trasparenti, il valore di z in questo $z-$buffer misura, pixel per pixel, la profondità del poligono trasparente più lontano dall'osservatore. Si ripete il procedimento per determinare i poligoni trasparenti successivamente più vicini, cioè, per ogni dato pixel, la pila dei poligoni trasparenti via via più vicini, che vengono memorizzati in analoghi $z-$buffer successivi: ma solo pedr quei pixel la cui flag era diventata true al primo passo (in corrispondenza degli altri evidentemente non c' è alcun poligono trasparente). In ciascuno di questi $z-$buffer successivi non viene memorizzato il valore di colore e trasparenza del poligono che viene messo nel frame buffer, bensì il valore che risulta dall'applicazione della equazione di illuminazione applicata a quel poligono ed al valore risultante allo stesso pixel nello $z-$buffer precedente. In tal modo, nel corso della scansione, ogni $z-$buffer acquisisce il valore di colore risultante dai poligoni trasparenti e opachi sottostanti nella pila, cioè quello dato dalla sovrapposizione dei successivi filtri colorati (semi)trasparenti. Al termine del procedimento l'ultimo $z-$buffer contiene i valori finali di colore.

Esempio 5.7.1. Una scena consiste dei seguenti poligoni, visti frontalmente nel modo in cui si ricoprono. I quattro rettangoli orizzontali o verticali si indicano con 0, 1, 2, 3 rispettivamente, in senso antiorario a partire da quello verticale a sinistra. Le profonditaà dei vertici sono le seguenti:

0: vertici alti a $z = 0$, vertici bassi a $z = 12$;
1: vertici di sinistra a $z = 0$, vertici di destra a $z = 12$;
2: vertici bassi a $z = 6$, vertici alti a $z = 12$;
3: vertici di destra a $z = 4$, vertici di sinistra a $z = 8$.

Si disegnino i grafici dei valori di profondità (al variare dei pixel in ascissa) dello $z-$buffer nelle due righe di scansione tratteggiate in Figura 5.7.1, dopo la scansione del primo poligono, del secondo, e così via fino all'ultimo, supponendo di scandirli nell'ordine seguente: prima il rettangolo orizzontale alto, poi quello basso, poi gli altri tre nell'ordine da sinistra a destra. □

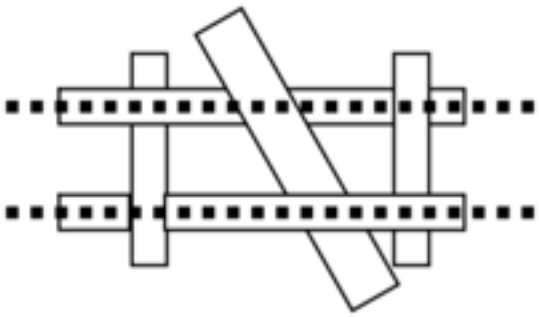


Figura 5.7.1. La scena dell'Esempio 5.7.1

5.8. Appendice: codice in Java di un'animazione del sistema Terra-Luna basata su $z-$buffer per rimpiazzare il Ray Tracing, e su tessiture animate ed inclinate

Nella Sezione 1.9 abbiamo presentato un algoritmo per realizzare una variante velocizzata del Ray Tracing (con metodo di illuminazione di Phong) basata sullo $z-$buffer. In questo modo si ottiene una notevole accelerazione del Ray Tracing senza ricorrenza. Rammentiamo come funziona questo algoritmo. Al termine della procedura di $z-$buffer, il buffer dei poligoni ci dice quale è il poligono (triangolo) visibile attraverso il centro di ciascun pixel, e quindi il versore normale $\boldsymbol{N}$ al punto osservato. Si noti che le coordinate x', y' del centro del pixel individuano un punto sul viewplane, la cui profondità è, convenzionalmente, $z' = 1$ se l'ossservatore è posto a $z' = 0$. Da quersto e dal dal valore di profondità z del punto osservato (immagazzinato nello $z-$buffer), ricaviamo tramite una omotetia (ossia una proporzione, quella della proiezione centrale) le coordinate tridimensionali di tale punto, che sono (x, y, z) dove $x/z = x'$ e $y/z = y'$. Ora, dalla posizione nello spazio del punto osservato, ricaviamo il versore $\boldsymbol{L}$ della direzione della sorgente di luce visto da tale punto, e quindi otteniamo da (7.7.1) il versore riflesso. Questo ci ha permesso di illuminare il punto osservato mediante l'equazione di illuminazione di Phong. Ma possiamo inoltre tracciare il raggio riflesso ed implementare il Ray Tracing alla prima generazione di ricorrenza. Analogamente si possono poi eseguire le fasi successive di tracciamento dei raggi riflessi per ottenere le successive generazioni di ricorrenza: questo porta ad una corrispondente velocizzazione del Ray Tracing ricorsivo introdotto nella Sezione 4.2.

Ora possiamo scrivere un codice che sfrutta questa accelerazione del Ray Tracing data dallo $z-$buffer e le tessiture animate ed inclinate sviluppate nel precedente Esempio 5.3.1 per rendere una animazione del sistema Terra-Luna.
Però per questo rendering occorre anche tracciare nello spazio, da ogni punto osservato su ciascuna delle due sfere che rappresentano i corpi celesti, un raggio d'ombra verso la sorgente di luce (introdotto nella Sezione 4.2) ed intersecarlo con i triangoli dell'altra sfera. Questo è indidpensabile al fine di rendere le eclissi, nelle quali uno dei due corpi celesti oscura la luce su una parte dell'altro. Ma è anche possibile tracciare dallo stesso punto anche un raggio riflesso, per simulare il Ray Tracing ricorsivo alla prima generazione. L'operazione di tracciamento ed intersezione è svolta nella classe Ray.class del codice Java qui sotto.
Il movimento dei due corpi celesti è determinato dalla legge di gravitazione di Newton, integrata mediante il metodo numerico di Eulero. Sono state trascurate le forse di marea, perché in realtà né la Terra né la Luna ruotano sul proprio asse: sono solo le tessiture dipinte sopra di loro a ruotare!

Chi esegue questo codice sta in realtà utilizzando un linguaggio di programmazione semi-interpretato e quindi assai lento. Nonostante questo, l'accelerazione data dallo $z-$buffer è adeguata per produrre una animazione abbastanza veloce del movimento di Terra e Luna, nonostante la mole di calcolo aggiuntiva richiesta dalle tessiture animate. I test di verifica dei tempi m=necessari al procedimento nelle sue varie parti, a seconda dei metodi impiegati, sono calcolati dalla classe benchmark.class.

Anche il codice Java che segue è stato elaborato e comunicato da Giacomo Nazzaro [**32**].

```
import java.awt.*;
import java.util.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.awt.image.BufferedImage;

public class Main extends JFrame{
   public MyJPanel canvas;
```

```
public Scene scn;
public static Main sv;
public RenderingEngine rnd;


public Main(String s, int width, int height){
// Main si occupa solo di far eseguire il programma e gestire l'interfaccia
    grafica.
// Non e' interessante dal punto di vista teorico. Non e' pienamente commentato.

   super(s);//costruttore della classe genitore

   /////
   setTitle("Prova");

   setSize(width,height);
   setLayout(new BorderLayout());

     // Il parametro numerico di scn e' la luce ambientale
   scn = new Scene(0.04);

     Benchmark.start();
   scn.addSphere(300, new Point(0,0, 1100), 100, "world.jpg", 0.1, 1.0);
   scn.obj.get(0).setColor(new Point(0,0,1));
   scn.obj.get(0).rotateObjectY(0.41, new Point(0,0,1100));
   scn.obj.get(0).setStill(true);

   scn.addSphere(100, new Point(700, 0, 1100), 40, "moon_NASA0.jpg", 0.2, 0.8);
   scn.obj.get(1).rotateObjectY(-0.116588, new Point(0,0,400));
   scn.obj.get(1).rotateTexture(-0.0);
   scn.obj.get(1).setVelocity(new Point(0,0,40));

     /* il primo parametro e' il raggio della sfera;
      l'argomento Point e' il centro: la variabile y
      e' l'altezza e cresce verso il basso,
      z e' la profondita' spaziale e cresce all'allontanarsi
      dal viewplane, che e' a z=0;
      il parametro successivo e' il numero di ripartizioni
      della latitudine e della longitudine nel creare
      la modellazione a maglie triangolari della sfera;
      i tre parametri successivi sono le coordinate R, G, B del colore, fra 0 e
          1;
      infine abbiamo il coefficiente di riflessione e quello di diffusione.
      */

     Benchmark.show("make scene");

     // Creo e aggiungo le sorgenti di luce
   Light l1 = new Light(-366, -166, 800, 0);
```

```
Light l2 = new Light(-3000, 0, 0, 1);
//l2.setDirectional(new Point(1,0,-1));
l2.l=0.0;
l2.c=1.0;
l1.l=0.5;
l2.q=l1.q=0;
scn.lgt.add(l1);
scn.lgt.add(l2);

rnd = new RenderingEngine(width, height, 700, new Point(0,0,-200), new
    Point(0,0,1), scn);

canvas = new MyJPanel(width, height);
this.add(canvas);

this.addKeyListener(new KeyListener() {
      public double theta, vx, vy, vz, intensity;
      Point v = new Point();
   String texture_name = "world.jpg";
   @Override public void keyPressed(KeyEvent e) {
        if(e.getKeyCode() == KeyEvent.VK_UP) vy = -10;
        if(e.getKeyCode() == KeyEvent.VK_DOWN) vy = 10;
        if(e.getKeyCode() == KeyEvent.VK_LEFT) vx = -10;
        if(e.getKeyCode() == KeyEvent.VK_RIGHT) vx = 10;

        if(e.getKeyCode() == KeyEvent.VK_BACK_SPACE) vz = -10;
        if(e.getKeyCode() == KeyEvent.VK_SPACE) vz = 10;

      rnd.shiftCamera(new Point(vx,vy,vz));
      vx=0; vy=0; vz=0;
       repaint();
    }

   @Override public void keyTyped(KeyEvent e) {
      theta = 0;
        if(e.getKeyChar() == 'g') rnd.gouraud = !rnd.gouraud;
        if(e.getKeyChar() == 't') rnd.texture_mapping = !rnd.texture_mapping;
        /*if(e.getKeyChar() == 'n') {
         if(texture_name.equals("world.jpg"))
            texture_name = "world_night.jpg";
         else
            texture_name = "world.jpg";

         scn.obj.get(0).setTexture(texture_name);
        }*/
        if(e.getKeyChar() == 'z') rnd.zShading = !rnd.zShading;
        if(e.getKeyChar() == 'b')
            {scn.obj.get(0).setBlinn(true);scn.obj.get(1).setBlinn(true);}
```

```
            if(e.getKeyChar() == 'p')
                {scn.obj.get(0).setBlinn(false);scn.obj.get(1).setBlinn(false);}
            if(e.getKeyChar() == 'm') {Benchmark.enabled = !Benchmark.enabled;}
            if(e.getKeyChar() == 'l') {
             intensity = 200 - intensity;
             scn.lgt.get(0).setIntensity(intensity);
            }
            if(e.getKeyChar() == 'n') {Object.c = !Object.c;}
           repaint();
        }

        @Override public void keyReleased(KeyEvent e) {
          theta = 0;
          vx=vy=vz=0;
          v.set(0,0,0);
        }
      });

      this.addMouseListener( new MouseListener() {


    public void mousePressed(MouseEvent e) {
       //System.out.println("Mouse pressed; # of clicks: ");
    }

    public void mouseReleased(MouseEvent e) {
       //System.out.println("Mouse released; # of clicks: ");
    }

    public void mouseEntered(MouseEvent e) {
       //System.out.println("Mouse entered");
    }

    public void mouseExited(MouseEvent e) {
       //System.out.println("Mouse exited");
    }

    public void mouseClicked(MouseEvent e) {
      Point p = rnd.camera.sum(e.getX()-rnd.w_2, e.getY()-rnd.h_2, 0);
       p.z = 800;
       scn.lgt.get(0).set(p);
    }
   });
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.pack();
      this.setVisible(true);
      sv = this;
}
```

```
public static void main(String args[]){

   Benchmark.disable();
   System.out.println(
      "\nCOMANDI\n\nFreccette direzionali per muovere la camera nello spazio.
      \nSpacebar per avvicinare la camera.\nBackspace per allontanare la camera.
      \n\nG: attivare/disattivare il Gouraud shading\nT: attivare/disattivare il
         texture mapping\nN: attivare/disattivare le nuvole
      \nB/P: scegliere fra il modello di illuminazione (Blinn o Phong)\nM:
         attivare/disattivare il benchmarking
      \nZ: attivare/disattivare lo 'z-shading' (l'ombreggiatura si basa sul
         valore di profondit\'{a} del pixel nello $z-$buffer)
      \nL: attivare una seconda luce\nClick sinistro: posizionare seconda
         luce\n");
   int width = 1000;
   int height = 800;

   Main window = new Main("Renderkid", width, height);

   while(true) window.repaint();

}
```

La prossima classe MyJPanel crea la finestra di visuale. L'ultima riga incrementa la variabile temporale dell'animazione.

```
class MyJPanel extends JPanel{

   public Point v = new Point();
   public float r, g, b;
   public int f,x,y,width,height;
   public int t;
   public BufferedImage image;
   public double sin, cos;
   public Object earth, moon;
   public double distance_sq;
   public Point acceleration, velocity;

   public MyJPanel(int w, int h){
      super();
      width = w;
      height = h;
      this.setPreferredSize(new Dimension(width, height));
      this.setVisible(true);

      earth = scn.obj.get(0);
      moon = scn.obj.get(1);
```

```
    }

     @Override public void paintComponent(Graphics gr){
       super.paintComponent(gr);
       //gr.setPaint ( new Color ( r, g, b ) );
       //gr.fillRect ( 0, 0, b_img.getWidth(), b_img.getHeight() );

       Benchmark.start();
       image = rnd.render();
         Benchmark.show("render");

         Benchmark.start();
       gr.drawImage(image, 0, 0, null);
       Benchmark.show("draw pixels");

       Object.attract(moon, earth);
       scn.lgt.get(1).rotateY(0.00595);
       scn.simulate();
       earth.rotateTexture(-0.01);
       moon.rotateTexture(-0.0107);
       t+=1; //incremento la variabile temporale ad ogni frame
    }

  }

}
```

La prossima classe esegue test di velocità opportuni per identificare colli di bottiglia nei vari metodi che si possono utilizzare.

```
import java.util.*;

public class Benchmark{

   public static double t;
   public static boolean enabled=true;

   public static void main(String[] args){
      double N;
      if(args.length != 0 && args[0] != null){
         N = Double.parseDouble(args[0]);
      }
      else {
         N = 1000000000;
      }
      int k;
      double x;
      // varibaili test
```

```
   /*ArrayList<Point> list = new ArrayList<Point>();
   for (int i=0; i<N; i++) {
      //list.add((new
         Point(Math.random(),Math.random(),Math.random()).per(1000)));
   }
   Point p1 = new Point(0.34, 0.23, 1.3);
   Point p2 = new Point(983.34, 55.06, 123.3);
   Point p3 = new Point(870.63, 2346.2, 678.3);
   Point q = new Point(324.453, 543.23, 841.32);
   Point p;
   System.out.println("N = "+N+"\n");
   */
   int i=0;


   //------- TEST 1 ----------
   start();
   for(;i<N;i++){
   // operazioni da testare qui sotto
      x =  Math.sqrt(N*825726243.5564);


   // operazioni da testare qui sopra
   }
   show("Sqrt");
   //-----------------------//


   i=0;

   //------- TEST 2 ----------
   start();
   for(;i<N;i++){
   // operazioni da testare qui sotto

      x = 3245545.4353*N*(825726243.5564);

   // operazioni da testare qui sopra
   }
   show("New");
   //-----------------------//
}

public static void enable(){
   enabled = true;
}

public static void disable(){
   enabled = false;
```

```
}

public static void doSwitch(){
   enabled = !enabled;
}

public static void start(){
   if(!enabled) return;
   t = System.nanoTime();
}

public static void start(String s){
   if(!enabled) return;
   System.out.println(s);
   t = System.nanoTime();
}

public static void stop(){
   if(!enabled) return;
   t = System.nanoTime() - t;
   t /= 1000000;
}

public static void show(String s){
   if(!enabled) return;
   stop();
   print(s);
}

public static void show(){
   if(!enabled) return;
   stop();
   print("No name");
}

public static void show(double d){
   if(!enabled) return;
   stop();
   print(d);
}

public static void print(String s){
   if(!enabled) return;
   System.out.println(s+": "+t+" ms");
}

public static void print(double d){
   if(!enabled) return;
   System.out.println(d+": "+t);
```

```
   }

}
```

```
import java.util.*;
import java.awt.*;

public class Point{

  // cooridnate del punto nello spazio
  public double x;
  public double y;
  public double z;


  // colore
  public double diffuse;
  public double specular;
  public double u, v;
  // versore normale al punto
  public Point normal;

  public int id;

  public Object ob;

  public Point(){
    //costruttore
      this.set(0, 0, 0);
  }

  public Point(double a){
    //costruttore
      this.set(a, a, a);
  }

  public Point(double a, double b, double c){
    //costruttore da coordinate
    this.set(a, b, c);
  }

  public Point(Point P){
    // costruttore che copia un altro punto P
    this.set(P.x, P.y, P.z);
    normal = P.normal;
```

```
    diffuse = P.diffuse;
    specular = P.specular;
    u = P.u;
    v = P.v;
  }

  public Point copy(Point P){
    // costruttore che copia un altro punto P
    Point cp = new Point(P.x, P.y, P.x);
    cp.normal = P.normal;
    cp.diffuse = P.diffuse;
    cp.specular = P.specular;
    cp.u = P.u;
    cp.v = P.v;

    return cp;
  }


  public void set(double a, double b, double c){
    // set delle coordinate e colore a grigio 70% circa
    x = a;
    y = b;
    z = c;
  }

  public void set(Point P){
    // set delle coordinate
    x = P.x;
    y = P.y;
    z = P.z;
  }

  public Point sum(Point P){
    // somma vettoriale
    return new Point(x + P.x, y + P.y, z + P.z);
  }

  public Point sum(double a, double b, double c){
    // somma vettoriale
    return new Point(x + a, y + b, z + c);
  }

  public void add(Point P){
    x += P.x;
    y += P.y;
    z += P.z;
  }
```

```
public void add(double s){
  x += s;
  y += s;
  z += s;
}

public void shift(Point v){
  // traslazione di vettore v
  x += v.x;
  y += v.y;
  z += v.z;
}

public void shift(double a, double b, double c){
  // traslazione da coordinate
  x += a;
  y += b;
  z += c;
}

public Point min(Point P){
  // differenza vettoriale
  return new Point(x - P.x, y - P.y, z - P.z);
}

public Point per(double l){
  // moltiplicazione per uno scalare l
  return new Point(x * l, y * l, z * l);
}

public void scale(double l){
  x *= l;
  y *= l;
  z *= l;
}

public Point star(Point P){
  return new Point(x*P.x, y*P.y, z*P.z);
}

public double dot(Point P){
  // prodotto scalare
  return x*P.x + y*P.y + z*P.z;
}

public double dot(double x_, double y_, double z_){
  // prodotto scalare
  return x*x_ + y*y_ + z*z_;
}
```

```
public Point vect(Point P){
  // prodotto vettoriale
 return new Point(this.y*P.z - this.z*P.y, this.z*P.x - this.x*P.z, this.x*P.y -
     this.y*P.x );
}

public Point to(Point P){
  // resituisce il vettore applicato che va da "this" al punto P
  return P.min(this);
}

public double norm(){
  // norma euclidea
  return Math.sqrt(this.dot(this));
}

public double squareNorm(){
  // norma euclidea al quadrato
  return this.dot(this);
}

public Point normalized(){
  // noramlizzazione
  double n = this.norm();
  if(n == 0) return this;
  return this.per( 1/n );
}

public double normalize(){
  // noramlizzazione
  double n = this.norm();
  if(n == 0) return 0;
  x /= n; y /= n; z /=n;
  return n;
}

public Point normalizeL1(){
  // normalizzazione secondo la norma l1
  double n = Math.abs(x) + Math.abs(y) + Math.abs(z);
  if(n == 0) return this;
  return this.per(1/n);
}

public Point getNormal(){
  // getNormal() restituisce la normale, se esiste
  if(normal == null){
    System.out.println("Normal is null!:");
    return new Point(0,0,0);
```

```
  }
  return this.normal;
}

public void setNormal(Point v){
  // imposta normale al punto
  this.normal = v;
  normal.normalize();
}

public void clamp(double min, double max){
 // clamp() costringe i valori del vettore fra min e max
  x = clamp(x, min, max);
  y = clamp(y, min, max);
  z = clamp(z, min, max);
}

public static double clamp(double value, double min, double max){
  // clamp() costringe i valori di un double fra min e max
  if(value<min) value=min;
  if(value>max) value=max;
  return value;
}

public void print(){
 // stampa a schermo delle coordinate del punto
  System.out.println("p("+x+" "+y+" "+z+")");
}

public Point pow(double p){
  return new Point( Math.pow(x, p), Math.pow(y, p), Math.pow(x, p));
}

public double getDistFrom(Point P){
  // distanza fra due punti
  Point V = this.to(P);
  return Math.sqrt(V.dot(V));
}

public double getSquareDistFrom(Point P){
  // distanza fra due punti
  Point V = this.to(P);
  return V.dot(V);
}


public Point project(double f){
  /* project() calcola la proiezione centrale di un punto sul piano z=0 rispetto
      al centro (0,0,-f)
```

```
    f rappresenta la distanza del viewplane dal punto di vista, in termini
       fotografici
    e' la lunghezza focale */
   double k = f/(f + z);

   // risultato della matrice di proiezione, conservo la coordinata z dello spazio
   return new Point(k*x, k*y, z);
  }

  public static Point comb(double a, Point A, double b, Point B){
   // comb calcola la combinazione lineare di due vettori
   // usando i coefficienti a e b.
   return (A.per(a)).sum(B.per(b));
  }



  public static Point getInterpolation(double a, Point A, double b, Point B){
   //Point p = comb(a,A,b,B);
   Point p = new Point(a*A.x+b*B.x, a*A.y+b*B.y, a*A.z+b*B.z);
   p.u = a*A.u + b*B.u;
   p.v = a*A.v + b*B.v;
   p.diffuse = a*A.diffuse + b*B.diffuse;
   p.specular = a*A.specular + b*B.specular;
   return p;
  }

  public static Point comb(double a, Point A, double b, Point B, double c, Point
     C){
   // comb calcola la combinazione lineare di tre vettori
   // usando i coefficienti a,b e c.
   Point p = new Point(a*A.x + b*B.x + c*C.x, a*A.y + b*B.y + c*C.y , a*A.z +
      b*B.z + c*C.z);
   return p;
  }

  public static Point average(Point P, Point Q){
   // punto medio fra due punti
   return (P.sum(Q)).per(0.5);
  }

  public static Point average(Point P, Point Q, Point U){
     // baricentro di tre punti
   return (P.sum(Q).sum(U)).per(1/3.0);
  }


  public int toRGB(){
   // normalizzo a valori compresi fra 0 e 255
```

```
  Point shade = this.per(255);

  // contengo l'illuminazione fra il valore massimo (255) e minimo (0)
  shade.clamp(0,255);
  int r = round(shade.x);
  int g = round(shade.y);
  int b = round(shade.z);
  int clr = (r << 16) | (g << 8) | b;
  return clr; // restituisco l'illuminazione finale
}

public Point getConvexCoeff(Point v1, Point v2, Point v3){
  Point l1 = v1.to(this);
  Point l2 = v2.to(this);
  Point l3 = v3.to(this);
  double A3 = (l1.vect(l2)).norm();
  double A2 = (l3.vect(l1)).norm();
  double A1 = (l3.vect(l2)).norm();
  double A = A1+A2+A3;

  return new Point(A1/A, A2/A, A3/A);
}

public Point getConvexCoeffNew(Point v0, Point v1, Point v2){
  Point w1 = v1.min(v0);
  Point w2 = v2.min(v0);
  Point q = this.min(v0);
  double alpha = q.dot(w1) / w1.dot(w1);
  double beta = q.dot(w2) / w2.dot(w2);
  return new Point(1-alpha-beta, alpha, beta);
}


public Point getConvexCoeff(Triangle tr){
  Point v1=tr.a, v2=tr.b, v3=tr.c;
  return this.getConvexCoeff(v1,v2,v3);
}

public Point getNormalInterpolation(Point p1, Point p2, Point p3){
  Point coeff = this.getConvexCoeff(p1,p2,p3);
  double alpha = coeff.x, beta = coeff.y, gamma = coeff.z;
  Point N = Point.comb(alpha, p1.normal, beta, p2.normal, gamma, p3.normal);
  return N;//.normalize();
}

public Point getNormalInterpolation(Triangle tr){
  return this.getNormalInterpolation(tr.a, tr.b, tr.c);
}
```

```
  public Point project(Point camera, double f){
    Point pc = this.min(camera);
    double depth = pc.dot(camera.normal);
    pc.x *= f / depth;
    pc.y *= f / depth;
    pc.z *= -1;
    //pc.shift(camera);
    Point p = new Point(pc.x, pc.y, depth);
    p.diffuse = diffuse;
    p.specular = specular;
    p.u = u;
    p.v = v;
    return p;


  }

  public void rotateZ(double theta){
    double yr,xr;
    double cos = Math.cos(theta), sin = Math.sin(theta);
    xr = x*cos + y*sin;
    yr = -x*sin + y*cos;
    x = xr;
    y = yr;
  }

  public void rotateY(double theta){
    double zr,xr;
    double cos = Math.cos(theta), sin = Math.sin(theta);
    xr = x*cos + z*sin;
    zr = -x*sin + z*cos;
    x = xr;
    z = zr;
  }

  public static int round(double x){
    // round() arrotonda valori double all'intero piu' vicino
    // e' utilizzata per passare dalle corrdinate dello spazio (double)
    // alle coordinate della pixelMatrix (int)
    return (int) (Math.round(x));
  }


}
```

```
import java.awt.*;

public class Light extends Point{
   // sorgente luminosa puntiforme

  public double i; // intensita' della sorgente luminosa
  public double q, l, c; // coefficienti di decadimento {quadratico, lineare e
      costante} */
  public Point direction;
  public char type;
  //public Point color;

  public Light(double x, double y, double z, double intensity){
   super(x, y, z);
   i = intensity;
   c=0;
   q = 1;
   l = 0;
  }

  public void setDirectional(Point d){
    type = 'd';
    direction = d.normalized();
    this.set(direction.per(1000));
  }

  public void setIntensity(double intensity){
    i = intensity;
  }

  public Point getDirection(Point p){
    if(type == 'd')
      return direction;
    else return p.to(this);
  }

  public double getAttenuation(double d){
      // getAtt() restituisce l'attenuazione dovuta dalla distanza, usando i
          coefficienti
      // caratteristici della sorgente luminosa

    double att = (c + l*d + q*d*d);
    if(att<1)
        return 1;
    return 1/att;
  }
}
```

```
import java.awt.*;

public class Triangle{
   public Point a, b, c;
   public int IDa, IDb, IDc;


   public Point center;



   public Triangle(Point v1, Point v2, Point v3){
      a=v1;
      b=v2;
      c=v3;

      if(a != null && b != null && c != null)
         center = Point.average(a, b, c);

      if(a.normal != null && b.normal != null && c.normal != null)
         center.normal = Point.average(a.normal, b.normal, c.normal);
      else
         this.makeNormal();

      /*if(a.clr != null && b.clr != null && c.clr != null)
         center.normal.clr = Point.average(a.clr, b.clr, c.clr);
      */
   }

   public Point getNormal(){
      if(center.normal != null)
         return center.normal;
      if(this.updateNormal() != null)
         return center.normal;

      return this.makeNormal();
   }

   public Point updateNormal(){
      if(a.normal != null && b.normal != null && c.normal != null){
         center.normal = Point.average(a.normal, b.normal, c.normal);
         return center.normal;
      }
      else return null;
   }
```

```
   public Point makeNormal(){
      Point v = a.min(b);
      Point w = c.min(b);
      center.normal = w.vect(v).normalized();
      return center.normal;
   }

   public void print(){
      System.out.println("tr:");
      a.print();
      b.print();
      c.print();
   }

   public Point getCenter(){
      Point cnt = Point.average(a,b,c);
      center = cnt;
      return center;
   }

   public Triangle sum(Point p){
      return new Triangle(a.sum(p), b.sum(p), c.sum(p));
   }

   public void shift(Point v){
       // traslazione di vettore v
       a.shift(v);
       b.shift(v);
       c.shift(v);
       center = this.getCenter();
   }


}
```

```
import java.util.*;

public class Scene{

   public ArrayList<Light> lgt; // Lista delle luci nella scena
   public double amb; // Colore della luce ambientale
   public ArrayList<Object> obj;
```

```
public Scene(double ambientLight){
   amb = ambientLight;
   lgt = new ArrayList<Light>();
   obj = new ArrayList<Object>();
}


public Object get(int i){
   return obj.get(i);
}

public void add(Object o){
   obj.add(o);
}

public void addLight(Light l){
   lgt.add(l);
}


public int size(){
   return obj.size();
}

public Object addSphere(double r, Point center, int n, double R, double G,
    double B, double ref, double dif){
   Object sphere = new Object();
   sphere.makeSphere(r,center,n,R,G,B,ref,dif);
   this.add(sphere);
   return this.get(this.size()-1);

}

public Object addSphere(double r, Point center, int n, String bitmap, double
    ref, double dif){
   Object sphere = new Object();
   sphere.makeSphere(r,center,n,bitmap,ref,dif);
   this.add(sphere);
   return this.get(this.size()-1);
}

public void simulate(){
   for (int i=0; i<obj.size(); i++) {
      obj.get(i).simulate();
   }
}


/*void shift(Point v, int start, int end){
```

```
      // shift() opera una traslazione di vettore v a tutti i punti
      // della scena compresi fra gli indici della lista 'start' e 'end'
      for (int i=start; i<end && i<this.pts.size(); i++) {
         this.pts.get(i).shift(v); //traslazione del punto
      }
   }*/

}
```

```
import java.awt.*;
import java.util.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.awt.image.BufferedImage;
import java.awt.Color;


public class RenderingEngine{
   public BufferedImage image;
   public Triangle[][] itemBuffer;
   public double[][] zBuffer;
   public int f, w, h, w_2, h_2;
   public Point p1, p2, p3, p4,left,right;
   public Point p[] = new Point[4];
   public Point camera, ux, uy, u0;
   public Scene scn;
   public double kDif, kRef;
   public ArrayList<Point> fragments;
   public Object object;
   public Point color;
   public boolean texture_mapping, gouraud, zShading = false;
   public int reflection_exponent;

   public RenderingEngine(int width, int height, int focal, Point camera_, Point
       normal, Scene scn_){
      //Costruttore del renderer
      w = width;
      h = height;
      f = focal;
      w_2 = w/2;
      h_2 = h/2;
      scn = scn_;
      camera = camera_;
      camera.normal = normal;
```

```
      camera.normal.normalize();
      u0 = camera.sum(camera.normal.per(f));
      uy = new Point(0, camera.normal.z, -camera.normal.y);
      uy.normalize();
      ux = uy.vect(camera.normal);
      image = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
      zBuffer = new double[w][h];
      itemBuffer = new Triangle[w][h];
      fragments = new ArrayList<Point>();
      texture_mapping = true;
      gouraud = true;
   }

   public BufferedImage render(){
      // render() esegue tutte le funzioni necessarie a fere il rendering della
          scena
      //Benchmark.start();
      // inizializzo la matrice dei pixel con il colore dello sfondo
      // e quella dello $z-$buffer con la profondit\‘{a} massima
      for(int i=0; i<w; i++)
         for(int j=0; j<h; j++){
            image.setRGB(i, j, 0);
            zBuffer[i][j] = Double.POSITIVE_INFINITY;
         }


      // per ogni triangolo della scena, calcolo il suo rendering
      Point vertex, frag;
      for(int i=0; i<scn.obj.size(); i++){
         object = scn.obj.get(i);
         for(int j=0; j<object.pts.size(); j++){
            vertex = object.pts.get(j);
            frag = vertex.project(camera, f);
            // salvo i valori di illuminazione di ogni vertice nel suo proiettato
                (fragment)
            setIllumination(vertex, frag);
            fragments.add( frag );
         }
         render(object);
         fragments.clear();
      }
      return image;
   }

   public void render(Object obj){
      kDif = object.kDif;
      kRef = object.kRef;

      if(!texture_mapping || obj.texture == null)
```

```
      color = obj.color;

   for (int i=0; i<obj.trn.size(); i++)
      render(obj.trn.get(i));
}

public void render(Triangle tr){
   // Questa funzione implementa il metodo di z-Buffer trovando
   // la proiezione del triangolo tr sul viewplane, ed
   // aggiornando i valori delle matrici image e zBuffer

   Point n = tr.getNormal(); // normale del triangolo tr

   // backface culling
   if( n.dot(tr.a.min(camera)) > 0 ) return;

   // carico in p[i] il vertici del triangolo, gia' proiettati sul viewport
      (fragments)
   p[1] = fragments.get(tr.a.id);
   p[2] = fragments.get(tr.b.id);
   p[3] = fragments.get(tr.c.id);


   // trasformo i punti nelle coordinate relativa alla camera
   toCameraSpace(p[1]);
   toCameraSpace(p[2]);
   toCameraSpace(p[3]);


   // frustum culling
   if(
      (Math.abs(p[1].x)>w_2 || Math.abs(p[1].y)>h_2) &&
      (Math.abs(p[2].x)>w_2 || Math.abs(p[2].y)>h_2) &&
      (Math.abs(p[3].x)>w_2 || Math.abs(p[3].y)>h_2)
   ) return;


   // eseguo scambi per far s\'{i} che p[1] sia il vertice piu' in alto sul
      viewplane (coordinata y pi\'{u} bassa),
   // p[2] al centro e p[3] sia il pi\'{u} basso
   if(p[1].y>p[2].y) swap(1,2);
   if(p[1].y>p[3].y) swap(1,3);
   if(p[2].y>p[3].y) swap(2,3);


   double zSlopeX = -n.x / n.z; // \'{e} l'incremento di z per ogni spostamento
      di 1 pixel lungo l'asse x
   double zSlopeY = -n.y / n.z; // \'{e} l'incremento di z per ogni spostamento
      di 1 pixel lungo l'asse y
```

```
    /*
       Adesso occorre fare un clipping del triangolo proiettato. Infatti la
           funzione scan() che disegna
       il triangolo su image funziona solo se uno dei lati \‘{e} parallelo
           all’asse x del view plane.
       Si opera dunque un taglio orizzontale all’altezza del vertice p[2]
           ottentedo due triangoli (uno
       superiore e uno inferiore), entrambi con un lato orizzontale (in comune)
    */

    // Caso banale: se due vertici hanno la stessa y, allora un lato \‘{e}
        orizzontale. Posso dunque eseguire
    // direttamente la funzione scan() e disegnare l’intero triangolo

    if(p[1].y == p[2].y){
       if(p[1].x>p[2].x) swap(1,2);
       scan(p[3], p[1], p[2], -1, zSlopeX, zSlopeY, tr);
       return;
    }

    // Nel caso generale devo trovare l’intersezione del triangolo proiettato sul
        viewplane con
    // la retta orizzontale passante per p[2]. Un punto d’interzezione \‘{e}
        ovviamente p[2], l’altro
    // si trova sul lato opposto e si ottiene tramite interpolazione lineare

    // coefficiente di interpolazione lineare
    double cc = (p[2].y - p[3].y) / (p[1].y - p[3].y);
    p[0] = Point.getInterpolation(cc, p[1], 1-cc, p[3]);

    // Ora bisogna disegnare i due triangoli proiettati sul viewplane (ora
        entrambi
      // con un lato orizzontale).
    // Opero degli scambi affinche’ p[1],p[2],p[0] sia il triangolo superiore e
    // p[3],p[2],p[0] quello inferiore
    if(p[2].x>p[0].x) swap(2,0);

    // si possono ora disegnare le due porzioni di triangolo con la funzione
        scan()
    scan(p[1], p[2], p[0], 1, zSlopeX, zSlopeY, tr);
    scan(p[3], p[2], p[0], -1, zSlopeX, zSlopeY, tr);
}


public void scan(Point p1, Point p2, Point p3, int verse, double zSlopeX, double
    zSlopeY, Triangle tr){
    /*
```

```
   scan() trova, con un procedimento incrementale, tutti i pixel interni del
      triangolo p1,p2,p3 e per ognuno
   di essi richiama draw(), la funzione che aggiorna di fatto le matrici
      image e zBuffer.
   Il valore 'v' indica se si sta operando lo scan di un triangolo superiore
      (+1) o inferiore (-1) del clipping
*/

//  Se si considera la retta che passa lungo il lato sinistro del triangolo,
//    ls rappresente l'incremento di x per ogni spostamento di 1 pixel lungo
    l'asse y del view plane
// (dunque l'inverso del coefficiente angolare)
double ls = (p1.x - p2.x) / (p1.y - p2.y);

// rs rappresenta la stessa quantita' per il lato destro del triangolo
double rs = (p1.x - p3.x) / (p1.y - p3.y);

double x, y, z, xl, xr, zx, c1, c2, c3, u, v, width;
int i, j;

double diffuse, specular;
int clr;
double p1u, p2u, p3u,p1v, p2v, p3v;
// Nel caso in cui v=+1, si sta facendo lo scan del triangolo superiore,
    dall'alto verso il basso
// (dunque dal vertice in alto verso il lato orizzontale)

double inv_height = 1.0/Math.abs(p2.y - p1.y);
if(verse==1){
   c1 = 1.0;
   z = p1.z;
   xl = xr = p1.x;
   for(y=p1.y; y<=p3.y; y++){
      // questo for() viene eseguito per ogni riga orizzonatale del triangolo
      zx = z;
      c2 = 1 - c1;
      width = xr - xl;
      for(x=xl; x<=xr; x++){
         //questo for() viene eseguito per ogni pixel della stessa riga
         // con draw(), scrivo sulla matrice di pixel il valore del colore c
             nel punto x,y del viweplane
         // e aggiorno in tali coordinate anche il valore dello zBuffer
         c3 = 1 - c1 - c2;
         u = c1*p1.u + c2*p2.u + c3*p3.u;
         v = c1*p1.v + c2*p2.v + c3*p3.v;
         if(gouraud){
            diffuse = c1*p1.diffuse + c2*p2.diffuse + (1-c1-c2)*p3.diffuse;
            specular = c1*p1.specular + c2*p2.specular + (1-c1-c2)*p3.specular;
         }
```

```
            else{
               diffuse = (p1.diffuse + p2.diffuse + p3.diffuse)/3.0;
               specular = (p1.specular + p2.specular + p3.specular)/3.0;
            }
            clr = getShading(u, v, diffuse, specular).toRGB();
            if(zShading) clr = new Point(-0.3+1000.0/zx).toRGB();

            draw(x, y, zx, clr,tr);

            // ottengo il nuovo valore di profondita' per la prossima iterazione
                (pixel di destra)
            zx += zSlopeX;
            c2 -= (1-c1)/width;
         }
         z += ls*zSlopeX + zSlopeY; // ottengo il nuovo valore di profondita'
             per la prossima iterazione (riga sottostante)
         xl += ls; // ottengo l'estremo sinistro dei valori di x per la prossima
             riga
         xr += rs; // ottengo l'estremo destro dei valori di x per la prossima
             riga
         // xl e xr sono gli estremi della riga per il ciclo interno, sono usati
             nella condizione
         // di permanenza del ciclo for() interno
         c1 -= inv_height;
      }
   }

   // Nel caso in cui v=-1, si sta facendo lo scan del triangolo inferiore,
       dall'alto verso il basso
   // (dunque dal lato orizzontale verso il vertice in basso)
   // le operazioni sono analoghe
   if(verse==-1){
      c1 = 0.0;
      z = p3.z;
      xl = p2.x;
      xr = p3.x;
      for(y=p3.y; y<=p1.y; y++){
         zx = z;
         c2 = 1 - c1;
         width = xr - xl;
         for(x=xl; x<=xr; x++){

            c3 = 1 - c1 - c2;
            u = c1*p1.u + c2*p2.u + c3*p3.u;
            v = c1*p1.v + c2*p2.v + c3*p3.v;
            if(gouraud){
               diffuse = c1*p1.diffuse + c2*p2.diffuse + (1-c1-c2)*p3.diffuse;
               specular = c1*p1.specular + c2*p2.specular + (1-c1-c2)*p3.specular;
            }
```

```
            else{
               diffuse = (p1.diffuse + p2.diffuse + p3.diffuse)/3.0;
               specular = (p1.specular + p2.specular + p3.specular)/3.0;
            }
            clr = getShading(u, v, diffuse, specular).toRGB();
            if(zShading) clr = new Point(-0.3+1000.0/zx).toRGB();

            draw(x, y, zx, clr,tr);
            zx += zSlopeX;
            c2 -= (1-c1)/width;
         }
         z += ls*zSlopeX + zSlopeY;
         xl += ls;
         xr += rs;
         c1 += inv_height;
      }
   }
}

public void draw(double xs, double ys, double z, int c, Triangle tr){
   // Se necessario, la funzione draw() aggiorna image con il colore c nel punto
       (xs,ys)
   // e inoltre aggiorna lo zBuffer nello stesso punto con il valore z

   int x, y;
   // passo dalle coordinate dello spazio a quelle dello schermo (in pixel)
   x = round(xs + w_2);
   y = round(ys + h_2);

   // Se le coordinate (x,y) del pixel non sono contenute nel viewplane, esco da
       draw()
   if( !(x<w && x>=0 && y<h && y>=0) )
      return;


   /* Se la coordinata z di profondita' e' minore di quella gia' presente nello
      zBuffer, allora aggiorno i valori nelle due matrici con il nuovo colore e
          la nuova profondita'*/
   double zbuffer = zBuffer[x][y];
   if( (z<zbuffer && z>0) ){
      zBuffer[x][y] = z;
      //itemBuffer[x][y] = tr;
      image.setRGB(x, y, c);
   }
}


public void swap(int i, int j){
```

```
    // swap() e' una funzione che scambia i nomi (puntatori) di due oggetti di
       tipo Point
    Point t;
    t = new Point(p[i]);
    p[i] = new Point(p[j]);
    p[j] = new Point(t);
  }


  public void setIllumination(Point p, Point fragment){
    // getShade ottiene l'illuminazione del punto p nella scena, con normale N,
    // calcolando il contributo ambientale, diffusivo (Lambert) e riflessivo
       (Phong)

    // inizializzazione di variabili
    double intensity;
    double lambert=0, phong=0, phong_base, diffuse=0, specular=0, d;
    Light light=null;
    double amb = scn.amb; // amb e' il colore della luce ambientale
    Point shade , N = p.normal;
    Point L, V;
    Point o = camera; // centro di proiezione
    Ray shadow_ray;
    boolean visible;
    for(int i=0; i<scn.lgt.size(); i++){
      // per ogni luce della scena, calcolo il suo contributo d'illuminazione
      visible = true;
      light = scn.lgt.get(i);

      // V e' il vettore che congiunge il centro del triangolo al centro di
         proiezione o
      V = p.to(o);

      // L e' il vettore che congiunge il centro del triangolo alla posizione
         della luce
      // (chiamato anche 'raggio d'ombra')
      L = light.getDirection(p);



      if(L.dot(N)<0){
        // Se la superficie del triangolo non 'vede' la luce,
        // non calcolo affatto il contributo di questa sorgente e
        // salto direttamente alla prossima luce
        continue;
      }

      shadow_ray = new Ray(light, p);
      for(int j=0; j<scn.obj.size() && visible; j++){
```

```
   if(p.ob != scn.obj.get(j))
      visible = shadow_ray.getVisibility(scn, j);
}

if(!visible) continue;
/*
RenderingEngine shadowCheck = new RenderingEngine(50,50,1, p, L,scn,
    generation+1);
print(shadowCheck.render()[25][25].getRed());
//itemBuffer[w_2][h_2].a.print();
if(shadowCheck.itemBuffer[25][25] != null) { continue;}
*/
        // d e' la distanza del triangolo dalla luce
intensity = light.i;
if(light.type != 'd'){
   d = L.norm();
   intensity = light.i * light.getAttenuation(d); // intensity e'
       l'intensita' luminosa che raggiunge il punto
   L.normalize(); // normalizzo il raggio d'ombra
}

V.normalize(); // normalizzo il raggio di vista
// L e V sono ora versori

/* diff e' il contributo d'illuminazione, diffusivo calcolato con la
    formula
   di Lambert: diff = <L,N>. Notare che a questo punto della funzione diff
       non
   puo' essere negativo per il controllo effettuato prima */
lambert = L.dot(N);
diffuse += lambert*intensity; // incremento il contributo diffusivo

/* ref e' il contributo di luce riflettente calcolato con la formula
   di Phong: ref = 2 <N,L> <N,V> - <L,V> */
//phong = 2*N.dot(L)*N.dot(V) - L.dot(V);

// H = L+V/||L+V|| blinn = <H, N>

if(object.blinn){
   phong_base = (L.sum(V).normalized()).dot(N);
   phong_base*=phong_base; phong_base*=phong_base; //Blinn
}
else
   phong_base = 2 * N.dot(L) * N.dot(V) - L.dot(V);

// se il contributo e' positivo
if(phong_base>0){
   phong = phong_base;
   for(int j=1; j<object.reflection_exponent; j++)
```

```
                phong *= phong_base;

              specular += phong*intensity;
           }
        }

        diffuse += amb;
        fragment.diffuse = diffuse;
        fragment.specular = specular;
}

   public Point getShading(double u, double v, double diffuse, double specular){
      Point clr;
      if(object.texture != null && texture_mapping){
         clr = object.getColor(u, v);
      }
      else
         clr = color;
      Point shading = clr.per(diffuse*kDif);
      //specular = 0.1 * clr.z/(clr.x+clr.y+clr.z);
      //specular *= Point.clamp( 1.2*(2*clr.z-(clr.x+clr.y))
          ,0,1);//(clr.x+clr.y+clr.z);
      specular *= Point.clamp( 2*(clr.z-clr.x-clr.y) ,0,1);//(clr.x+clr.y+clr.z);
      shading.add(specular);
      shading.clamp(0,1);
      return shading;
   }

   public int round(double x){
      // round() arrotonda valori double all'intero piu' vicino
      // e' utilizzata per passare dalle corrdinate dello spazio (double)
      // alle coordinate della image (int)
      return (int) (Math.round(x));
   }

   public static void print(double x){
      System.out.println(x);
   }

   public void toCameraSpace(Point p){
      Point pu = p.min(camera.normal.per(f));
      p.set(p.dot(ux), p.dot(uy), p.z);
   }

   public void shiftCamera(Point v){
      camera.shift(v);
      u0.shift(v);
   }
```

```
   public void rotateCameraSTEP(double theta){
      if(theta!=0) camera.normal = new Point(1,0,0);
   }
   public void rotateCamera(double theta){
      Point N = camera.normal;
      double cos = Math.cos(theta), sin = Math.sin(theta);
      double xr = N.x*cos + N.z*sin;
      double zr = -N.x*sin + N.z*cos;
      camera.normal.set(xr, N.y, zr);

      camera.normal.normalize();

   /* u0 = camera.sum( camera.normal.per(f) );

      ux = uy.vect(camera.normal);
      ux.normalize();*/
      u0 = camera.sum(camera.normal.per(f));
      uy = new Point(0, camera.normal.z, -camera.normal.y);
      uy.normalize();
      ux = uy.vect(camera.normal);
      ux.normalize();
   }

}
```

```
import java.util.*;
import java.awt.*;

public class Ray{

  // direzione raggio
  Point rd;

  // origine del raggio
  Point ro;

  public Ray(Point origin, Point direction){
    //costruttore
      rd = direction.min(origin);

      ro = origin;
  }

  public boolean getVisibility(Scene scn, int i){
    double t0, t1;
    t0 = this.intersect(scn.obj.get(i));
```

```
    if( (t0<0.001 || t0>0.9) ) return true;
    else return false;
  }

  public double intersect(Object ob){
    Point o_c = ro.min(ob.center);
    double b = rd.dot(o_c);
    double a = rd.dot(rd);
    double c = o_c.dot(o_c) - ob.size*ob.size;

    double delta = b*b - a*c;
    if(delta<0)
      return -10;
    delta = Math.sqrt(delta);
    double t1 = (-b + delta) / a;
    double t2 = (-b - delta) / a;
    if(t1<0) t1 = t2;
    if(t2<0) t2 = t1;
    double t = (t1<t2)? t1 : t2;
    if(t<0) return -20;
    return t;
  }


  public double intersect(Triangle tr){

    Point N = tr.getNormal();
    double rdn = N.dot(this.rd);
    //if( rdn<0.01 && rdn>-0.01) return 0.0;
    double d = N.dot(tr.a);
    double t = (d - this.ro.dot(N)) / rdn;
    if(t >= 0.99 || t<0.01 ){
     return 0;
    }
    return t;
  }
}
```

```
import java.awt.image.BufferedImage;
import java.awt.Color;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.event.*;
import java.util.*;
```

```
public class Object{
   public ArrayList<Point> pts;
   public ArrayList<Triangle> trn;
   public static boolean c=true;
   public double size;
   public Point center;

   public double mass;
   public Point v;
   public boolean still = false;

   public Point color;
   public double kDif, kRef;
   public int reflection_exponent = 2;
   public boolean blinn = false;
   public BufferedImage texture;
   public static BufferedImage clouds, night;

   public Object(){
      pts = new ArrayList<Point>();
      trn = new ArrayList<Triangle>();
      texture = null;
      v = new Point(0);
      if(clouds==null)
         try {
            clouds = ImageIO.read(new File("clouds_new_rs.jpg"));
         } catch (IOException e) {}
       if(night==null)
         try {
            night = ImageIO.read(new File("world_night.jpg"));
         } catch (IOException e) {}

   }

   public Object(ArrayList<Point> points, ArrayList<Triangle> triangles, double
       k_dif, double k_ref){
      pts = points;
      trn = triangles;
      kDif = k_dif;
      kRef = k_ref;
      texture = null;
   }

   public void setTexture(String bitmap_name){
       if(bitmap_name == null){
         texture = null;
       }

       try {
```

```
        texture = ImageIO.read(new File(bitmap_name));
      } catch (IOException e) {}
      if(texture != null) return;

        try {
          texture = ImageIO.read(new File("texture_not_found.png"));
      } catch (IOException e) {}
  }

  double length(double a, double b, double c, double d){
     double x = a-c, y = b-d;
     return Math.sqrt( x*x + y*y );
  }


  public Point getColor(double u, double v){
     u = (u % 1.0);
     v = (v % 1.0);
     if(u<0) u+=1.0;

     double d = 0;
     int xpixel = (int) ( u*texture.getWidth() );
     int ypixel = (int) ( v*texture.getHeight() );

     Color RGB = new Color(texture.getRGB(xpixel, ypixel));
     Point clr = new Point(RGB.getRed(), RGB.getGreen(), RGB.getBlue());
     clr = clr.per(1.0/255);

     // SHADER
     //u+=Main.sv.canvas.t/50.0;
     boolean bool = false;
     if(this==Main.sv.scn.obj.get(0)) bool = true;

     if( bool ){

        xpixel = (int) ( u*night.getWidth() );
        ypixel = (int) ( v*night.getHeight() );
        RGB = new Color(night.getRGB(xpixel, ypixel));
        Point clrn = new Point(RGB.getRed(), RGB.getGreen(), RGB.getBlue());
        clrn = clrn.per(1.0/255);
        double x = (u+Main.sv.canvas.t*(0.01+0.000947)) + 0.5;
        x = x%1.0;
        if(x<0) x+=1.0;
        x-=0.5;
        double a = 2*Math.abs(x);
        double b = Math.cos(3*x);
        b *= b; b*=b; b*=b; b*=b;b*=10;
        clr = Point.comb(a,clr,b,clrn);
```

```
      if(c){

         double uc=u, vc=v, verse=1.0/50;
         Point p = new Point(u-0.5,-v+0.5,0);
      //for (int i = 0; i < 2; i ++){
         p.add( FlowField(p).per( 0.03 ));
         p.add( FlowField(p).per( 0.03 ));

         uc = p.x-0.3;
         vc = p.y+0.5;
         uc = (uc % 1.0);
         vc = (vc % 1.0);
         if(uc<0) uc+=1.0;
         if(vc<0) vc+=1.0;

   /*while(uc>=1) uc -= 1;
   while(vc>=1) vc -= 1;
   while(uc<0) uc += 1;
   while(vc<0) vc += 1;
   if(uc>=1) uc=0;
   if(vc>=1) vc=0;*/
   //RenderingEngine.print(uc);
   //RenderingEngine.print(vc);

         Color cloud = new Color(clouds.getRGB( (int) ( uc*clouds.getWidth() ),
             (int) ( vc*clouds.getHeight() )));
         d = cloud.getRed()/100.0 - 0.2;
         d*=d;
         //if(d>0.5) d+= (Math.sin(100000*vc*u-v+uc)%1.0)/1.0;
         //if(d<0) d=0;
         d = d>1.8? 1.8:d;
         clr.add(d);
      }
   }

   /*{  double d1 = Math.sin( 100.0 * length(u,v,.3,.2) );
      double d2 = Math.sin( 140.0 * length( u,v,.1,.5) );
      double d3 = Math.sin( 130.0 * length( u,v,.9,.7) );
      d = d2+d1+d3;
      d += 2*Math.sin(10*(u*u+v*v)+Main.sv.canvas.t); //+u_time);
      d /= 5;
   }*/

   //RenderingEngine.print(d);
   //d*=d;
//d*=d;
   //d = Point.clamp(d, 0, 1);
```

```
    return clr;
  }

  public void setColor(Point c){
    //color = new Point(c.getRed()/255.0, c.getGreen()/255.0, c.getBlue()/255.0 );
    color = c;
   }

  public void setkDif(double x){
    kDif=x;
   }

   public void setkRef(double x){
    kRef=x;
   }

   public void setBlinn(boolean b){
    blinn = b;
   }

  public void makeSphere(double r, Point centro, int n, double R, double G, double
      B, double ref, double dif){
    this.makeSphere(r, centro, n);
    color = new Point(R, G, B);
    kDif = dif;
    kRef = ref;
    texture = null;
  }

  public void makeSphere(double r, Point centro, int n, String bitmap, double ref,
      double dif){
    //this();
    this.makeSphere(r,centro,n,1,1,1,ref,dif);
    this.setTexture(bitmap);
    //this.paintPoints();
  }

  public void makeSphere(double r, Point centro, int n){
    /* makeSphere() genra una mesh sferica di triangonli
       la sfera ha raggio r, centro c ed e' divisa in n paralleli ed
       n meridiani. In tutto e' composta da n^2 triangoli.
       La sfera ha colore (R,G,B) e coefficiente riflessivo e diffusivo
          rispettivamente pari a ref e dif. */
      //Object sphere = new Object(this.trn.size(), this.trn.size()+n*n);

    center = centro;
    size = r;
    mass = r*r*r;
    double x, y, z;
```

```
   double pi = Math.PI;
   int i,j;
   Triangle t;
   Point p;
   int id = 0;
   double theta = 0, phi = 0, dphi = pi/n, dtheta = dphi*2;
   double cosphi, costheta, sinphi, sintheta;

   for(i=0; i<n; i++){
      sinphi = Math.sin(phi); cosphi = Math.cos(phi);
      theta = 0;
      for(j=0; j<n; j++){
         sintheta = Math.sin(theta); costheta = Math.cos(theta);
         x = r * sinphi * costheta + center.x;
         z = r * sinphi * sintheta + center.z;
         y = -r * cosphi + center.y;
         p = new Point(x,y,z);
         p.setNormal(center.to(p));
         p.u = theta / (2*pi);
         p.v = phi / pi;
         p.diffuse = -1.0;
         p.id = id;
         p.ob = this;
         pts.add(p);
         id++;
         theta += dtheta;
      }
      phi += dphi;
   }
   int a, b, c;
   for (i=0; i<n-1; i++) {
      for (j=0; j<n; j++) {
         a =(i*n+j)%(n*n);
         b = ((i+1)*n+j)%(n*n);
         c = (i*n+j+1)%(n*n);
         t = new Triangle( pts.get(a), pts.get(b), pts.get(c));
         trn.add(t);

         a = ((i+1)*n+j)%(n*n);
         b = ((i+1)*n+j+1)%(n*n);
         c = (i*n+j+1)%(n*n);
         t = new Triangle( pts.get(a), pts.get(b), pts.get(c));
         trn.add(t);
      }
   }
}

public void rotateObjectZ(double theta, Point center){
   Point p;
```

```
      double zr,xr,nxr,nzr;
      double cos = Math.cos(theta), sin = Math.sin(theta);
      for (int i=0; i<pts.size(); i++) {
         p=pts.get(i);
         p.shift(center.per(-1));
         xr = p.x*cos + p.z*sin;
         zr = -p.x*sin + p.z*cos;
         nxr = p.normal.x*cos + p.normal.z*sin;
         nzr = -p.normal.x*sin + p.normal.z*sin;
         p.set(xr, p.y, zr);
         p.normal.set(nxr, p.normal.y, nzr);
         p.shift(center);
      }
   }

   public void rotateObjectY(double theta, Point center){
      Point p;
      double yr,xr,nxr,nyr;
      double cos = Math.cos(theta), sin = Math.sin(theta);
      for (int i=0; i<pts.size(); i++) {
         p=pts.get(i);
         p.shift(center.per(-1));
         xr = p.x*cos + p.y*sin;
         yr = -p.x*sin + p.y*cos;
         nxr = p.normal.x*cos + p.normal.y*sin;
         nyr = -p.normal.x*sin + p.normal.y*sin;
         p.set(xr, yr, p.z);
         p.normal.set(nxr, nyr, p.normal.z);
         p.shift(center);
      }
   }

   public void rotateTexture(double theta){
      Point p;
      for (int i=0; i<pts.size(); i++) {
         p = pts.get(i);
         p.u += theta;
         //p.u = p.u % 1.0; //if(p.u >= 1) p.u -= 0;
      }
   }


   void shift(Point v){
      // shift() opera una traslazione di vettore v a tutti i punti
      // della scena compresi fra gli indici della lista 'start' e 'end'
      for (int i=0; i<pts.size(); i++) {
         pts.get(i).shift(v); //traslazione del punto
      }
      center.shift(v);
```

```
}

void scale(double s){
   Point p;
   for (int i=0; i<pts.size(); i++) {
      p = pts.get(i);
      p.shift(center.per(-1));
      p.scale(s);
      p.shift(center);
   }
   size *= s;
   mass *= s*s*s;
}

public static double smoothstep(double edge0, double edge1, double x){
   double t; /* Or genDType t; */
   t = Point.clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
   return t * t * (3.0 - 2.0 * t);
 }

Point VortF (Point q, Point c){
   Point d = q.min(c);
   Point result = new Point(d.y, - d.x, 0);
   result = result.per( 0.25 / d.dot(d) + 0.05);
   return result;
}

Point FlowField (Point q){
  Point vr, c;
  double dir = 1.;
  c = new Point ((Main.sv.canvas.t%75 - 100)/60.0 , 0.6 * dir, 0);
  vr = new Point (0,0,0);
  for (int k = 0; k < 5; k++) {
    vr.add( VortF( q.per(4), c).per(dir) );
    c = new Point (c.x + 1., - c.y, 0);
    dir = - dir;
  }
  return vr;
}

public void setVelocity(Point p){
   v = p;
}

public static void attract(Object a, Object b){
   Point a_b = a.center.to(b.center);
   double d = a_b.normalize();
   if(d>5000) return;
   double force = 1.0/(22*d*d);
```

```
      a.v.add(a_b.per(force*b.mass));
      b.v.add(a_b.per(-1*force*a.mass));
   }

   public void simulate(){
      if(!still)
         this.shift(v);
   }

   public void setStill(boolean b){
      still = b;
   }



}
```

La prossima classe traccia un raggio nello spazio e lo interseca con i triangoli delle sfere. Questo è indidpensabile perché occorre tracciare raggi d'ombra verso la sorgente di luce al fine di rendere le eclissi, nelle quali uno dei due corpi celesti oscura la luce su una parte dell'altro. Ma è anche possibile trcciare un raggio riflesso, per simulare il Ray Tracing ricorsivo alla prima generazione.

```
import java.util.*;
import java.awt.*;

public class Ray{

  // direzione raggio
  Point rd;

  // origine del raggio
  Point ro;

  public Ray(Point origin, Point direction){
    //costruttore
      rd = direction.min(origin);

      ro = origin;
  }

  public boolean getVisibility(Scene scn, int i){
    double t0, t1;
    t0 = this.intersect(scn.obj.get(i));
    if( (t0<0.001 || t0>0.9) ) return true;
    else return false;
  }
```

```
  public double intersect(Object ob){
    Point o_c = ro.min(ob.center);
    double b = rd.dot(o_c);
    double a = rd.dot(rd);
    double c = o_c.dot(o_c) - ob.size*ob.size;

    double delta = b*b - a*c;
    if(delta<0)
      return -10;
    delta = Math.sqrt(delta);
    double t1 = (-b + delta) / a;
    double t2 = (-b - delta) / a;
    if(t1<0) t1 = t2;
    if(t2<0) t2 = t1;
    double t = (t1<t2)? t1 : t2;
    if(t<0) return -20;
    return t;
  }


  public double intersect(Triangle tr){

    Point N = tr.getNormal();
    double rdn = N.dot(this.rd);
    //if( rdn<0.01 && rdn>-0.01) return 0.0;
    double d = N.dot(tr.a);
    double t = (d - this.ro.dot(N)) / rdn;
    if(t >= 0.99 || t<0.01 ){
     return 0;
    }
    return t;
  }
}
```

```
import java.awt.*;

public class Light extends Point{
   // sorgente luminosa puntiforme

  public double i; // intensita' della sorgente luminosa
  public double q, l, c; // coefficienti di decadimento {quadratico, lineare e
      costante} */
  public Point direction;
  public char type;
  //public Point color;
```

```
  public Light(double x, double y, double z, double intensity){
   super(x, y, z);
   i = intensity;
   c=0;
   q = 1;
   l = 0;
  }

  public void setDirectional(Point d){
    type = 'd';
    direction = d.normalized();
    this.set(direction.per(1000));
  }

  public void setIntensity(double intensity){
    i = intensity;
  }

  public Point getDirection(Point p){
    if(type == 'd')
      return direction;
    else return p.to(this);
  }

  public double getAttenuation(double d){
      // getAtt() restituisce l'attenuazione dovuta dalla distanza, usando i
          coefficienti
      // caratteristici della sorgente luminosa

    double att = (c + l*d + q*d*d);
    if(att<1)
        return 1;
    return 1/att;
  }
}
```

5.9. Appendice: Ray Tracing velocizzato tramite $z-$buffer: una implementazione dell'animazione Terra-Luna in C++

In questa Appendice viene riscritta nel linguaggio C++ l'animazione del sistema Terra-Luna sviluppata in Java nella precedente Apppendice 5.8. Qui viene aggiuntna luines enza agli oceani, che riflettono la luce solare. Oltre alle tessiture animate già implementate nella versione in Java, a questo fine viene aggiunta una procedure di calcolo di questo effetto di illuminazione basato su raggi riflessi generati con una opportuna distribuzione di probabiità basata sulla BRDF, un metodo che verrà ampiamente studiato nel Capitolo 9 nell'ambito dei modelli di Illuminazione Globale. Il codice è stato sviluppato da Marco Petreri [**37**].

---

main.cpp:

```
/*
  File principale del programma contenente la funzione main().
  Ha il compito di instanziare l'interfaccia del programma e avviare il loop
      dell'animazione
*/

#include <iostream>

#include "gui.hpp"

int main(int argc, char const *argv[]){

  try {
      nanogui::init();
      {
          GUI * app = new GUI(); // creo l'interfaccia
          app->drawAll();
          app->setVisible(true);
          nanogui::mainloop(1); // avvio il loop
      }
      nanogui::shutdown();
  } catch (const std::runtime_error &e) {
      std::string error_msg = std::string("Caught a fatal error: ") +
          std::string(e.what());
      #if defined(_WIN32)
          MessageBoxA(nullptr, error_msg.c_str(), NULL, MB_ICONERROR | MB_OK);
      #else
          std::cerr << error_msg << std::endl;
      #endif
      return -1;
  }
  return 0;
}
```

---

world.cpp: questa è la classe principale che svolge sia la dinamica sia il rendering.

```
/*
  File di definizione della classe World, la classe principalmente coinvolta nel
      rendering della scena e nelle animazioni.
  Ha il compito di gestire e lanciare i processi basilari del calcolo
      dell'immagine, come la creazione della scena,
  la fase di proiezione su schermo, la rasterizzazione dei triangoli, la fase di
      ombreggiatura e il calcolo delle animazioni.
*/

#include "world.hpp"
#include "geometry/triangle.hpp"
#include "geometry/objectGroup.hpp"
#include "cameras/camera.hpp"
#include "utilities/film.hpp"
#include "utilities/background.hpp"
#include "utilities/parser.hpp"
#include "utilities/timer.hpp"
#include "utilities/material.hpp"
#include "lights/directional.hpp"
#include "lights/point.hpp"

using namespace std;

World::World(): parser(nullptr), camera(nullptr), frameBuffer(nullptr),
    zBuffer(nullptr){}

World::~World(){
  for(auto & i : objs)
    delete i;
  for(auto & i : materials)
    delete i.second;
  for(auto & i : lights)
    delete i;
  delete parser;
  delete camera;
  delete frameBuffer;
  delete background;
  delete [] zBuffer;
}

unsigned char * World::buildScene(const Vec3i & size, void (*buildF)(World *)){
  timer = new Timer(); // istanzio il timer
  parser = new Parser(this); // istanzio il parser
  frameBuffer = new Film(size.x, size.y, 3); // istanzio il framebuffer
  zBuffer = new float[frameBuffer->size()]; // istanzio lo z-buffer
  for(int i = 0; i < frameBuffer->size(); ++i) // per tutte le celle dello z-buffer
    zBuffer[i] = INFINITY; // setto a +oo il valore di distanza della cella i-esima
```

```
  buildF(this); // lancio la funzione che crea la scena

  return frameBuffer->map; // ritorno il riferimento al buffer del frame
}

void World::deleteScene(){ // distrugge la scena attuale
  clearBuffers();
  for(auto & i : objs)
    delete i;
  objs.clear();
  triangles.clear();
  for(auto & i : materials)
    delete i.second;
  materials.clear();
  for(auto & i : lights)
    delete i;
  lights.clear();
  delete background;
  delete camera;
}


void World::renderScene(){ // funzione principale del rendering della scena
  timer->start(); // avvio il timer
  for(auto & tr : triangles){ // per tutti i triangoli della scena
    Vec3f v[3], vc[3]; // creo un vettore di vertici nello spazio dello schermo
        (v[]) e un vettore di vertici nello spazio della camera (vc[])
    for(u_int i = 0; i < 3; ++i){ // scorro sui tre vertici del triangolo
      Vec4f vt = camera->worldToCam*Vec4f(tr->operator[](i), 1.f); // recupero il
          vertice i-esimo, lo includo in un vettore di 4 dimensioni e lo passo
          dallo spazio del World a quello della Camera tramite la matrice di
          cambiamento di base
      vc[i] = vt.xyz(); // mi salvo una copia del vertice in 3 dimensioni nello
          spazio della Camera
      vt = camera->perspective*vt; // proietto il vertice sullo schermo tramite
          matrice di proiezione
      float invW = 1.f/vt.w; // calcolo il reciproco della quarta coordinata
      vt.x *= invW; // normalizzo tramite la quarta coordinata
      vt.y *= invW; // normalizzo tramite la quarta coordinata
      vt.w = 1.f; // setto la quarta coordinata a 1
      v[i] = (camera->ndcToRaster*vt).xyz(); // passo il vertice proiettato dallo
          spazio Normalizzato allo spazio dello Schermo e mi salvo una copia
      v[i].x = floor(v[i].x); // arrotondo all'intero inferiore
      v[i].y = floor(v[i].y); // arrotondo all'intero inferiore
    }

    // in questa fase applico del clipping ai vertici, escludendo i triangoli con
        almeno un vertice dietro al piano di visione
```

```
if(v[0].z < 0 || v[1].z < 0 || v[2].z < 0) // se almeno una coordinata z di un
    vertice Ăš < del piano di visione posto a 0
  continue; // salto il triangolo

Vec3f e0 = v[1] - v[0], e2 = v[0] - v[2]; // calcolo i due spigoli del triangolo
float A = edgeFunction(-e2,e0); // calcolo l'area del parallelogrammo descritta
    dai due spigoli ( non c'Ăš bisogno di calcolare l'area del triangolo)

// in questa fase applico il backface culling, escludendo i triangoli con
    superfice opposta alla visuale
if(A < EPS) // se l'area ha segno negativo
  continue; // salto il triangolo

Vec3i min, max; // creo due vettori per il minimo e massimo del bounding-box
    del triangolo sullo schermo
min = max = Vec3i(v[0].x, v[0].y, 0); // setto entrambi al primo vertice
for(int i = 0; i < 3; ++i) // per tutte le coordinate del vertice
  for(int j = 1; j < 3; ++j){ // per i due vertici restanti
    int val = v[j][i]; // mi salvo il valore della coordianata i-esima del
        vertice j-esimo
    min[i] = val < min[i] ? val : min[i]; // se Ăš minore del valore attuale di
        min allora sostituisco
    max[i] = val > max[i] ? val : max[i]; // se Ăš maggiore del valore attuale
        di max allora sostituisco
  }


// in questa fase applico del clipping al bounding box del triangolo per
    evitare che ricada anche in parte all'esterno dell'immagine
min.x = clamp(min.x, 0, frameBuffer->width - 1); // clampo il valore tra 0 e
    width - 1
min.y = clamp(min.y, 0, frameBuffer->height - 1); // clampo il valore tra 0 e
    height - 1
max.x = clamp(max.x, 0, frameBuffer->width - 1); // clampo il valore tra 0 e
    width - 1
max.y = clamp(max.y, 0, frameBuffer->height - 1); // clampo il valore tra 0 e
    height - 1

Vec3f min_f(min.x, min.y, 0.f), uv, e1 = v[2] - v[1]; // mi salvo una copia in
    float del min, il vettore delle coordinate texture e lo spigolo mancante
float invZ0 = 1.f/v[0].z, invZ1 = 1.f/v[1].z, invZ2 = 1.f/v[2].z, invA = 1.f/A;
    // calcolo i reciproci delle coordinate z dei vertici e il reciproco
    dell'area del parallelogramma
float c0_b, c1_b, c2_b; // creo tre float per salvare le coordinate
    baricentriche relative al min

c0_b = edgeFunction(min_f-v[1], e1); // calcolo la cooordinata baricentrica di
    min rispetto a e1
```

```
c1_b = edgeFunction(min_f-v[2], e2); // calcolo la cooordinata baricentrica di
    min rispetto a e2
c2_b = edgeFunction(min_f-v[0], e0); // calcolo la cooordinata baricentrica di
    min rispetto a e0

// in questa fase controllo se ogni punto del bounding box ĂŠ interno al
    triangolo, se lo ĂŠ passo alla fase di ombreggiatura
#pragma omp parallel for collapse(2) private(uv) schedule(dynamic) // direttiva
    del preprocessore che parallelizza i due for sui core del processore
for(int i = min.y; i <= max.y; ++i){ // per tutta l'altezza del bounding-box
  for(int j = min.x; j <= max.x; ++j){ // per tutta la larghezza del
      bounding-box
    float c0 = c0_b + (j - min.x)*e1.y - (i - min.y)*e1.x; // calcolo la
        coordinata baricentrica del punto attuale rispetto a e1, partendo da
        quella rispetto a e1 del min e incrementandola con una combinazione
        lineare delle coordinate di e1 e del gap sulle x e sulle y del punto
        attuale rispetto al min
    float c1 = c1_b + (j - min.x)*e2.y - (i - min.y)*e2.x; // calcolo la
        coordinata baricentrica del punto attuale rispetto a e2, partendo da
        quella rispetto a e2 del min e incrementandola con una combinazione
        lineare delle coordinate di e1 e del gap sulle x e sulle y del punto
        attuale rispetto al min
    float c2 = c2_b + (j - min.x)*e0.y - (i - min.y)*e0.x; // calcolo la
        coordinata baricentrica del punto attuale rispetto a e0, partendo da
        quella rispetto a e0 del min e incrementandola con una combinazione
        lineare delle coordinate di e1 e del gap sulle x e sulle y del punto
        attuale rispetto al min
    if(c0 >= 0 && c1 >= 0 && c2 >= 0){ // se tutte e tre le coordinate
        baricentriche sono > 0 allora il punto ĂŠ interno al triangolo
      c0 *= invA*invZ0; c1 *= invA*invZ1; c2 *= invA*invZ2; // normalizzo le
          coordinate tramite l'area e la rispettiva coordinata z
      float z = 1.f/(c0 + c1 + c2); // calcolo l'effettiva profonditĂ  del
          punto, persa durante la fase di proiezione
      Vec3f pc = (vc[0]*c0 + vc[1]*c1 + vc[2]*c2)*z; // calcolo il punto
          attuale nello spazio della camera come interpolazione dei vertici
          nello spazio della camera
      Vec3f n = tr->lerpNormal(c0, c1, c2)*z; // calcolo la normale al punto
          come interpolazione delle normali a i vertici
      Vec3f wo = pc.hat(); // calcolo la direzione di visuale tra l'occhio
          della videocamera (posta in (0,0,0) nello spazio della camera) e il
          punto osservato
      Vec3f wi = wo.reflect(n); // calcolo la direzione del raggio riflesso
      Vec3f h = (wo + wi).hat(); // calcolo il vettore di halfway
      if(tr->mat->albedo_txt || tr->mat->specular_txt || tr->mat->emission_txt
          || tr->mat->reflection_txt) // se il materiale usa le texture
        uv = tr->lerpTexture(c0, c1, c2)*z; // calcolo le texture del punto
            come interpolazione delle texture ai vertici
```

```
          Vec3f R = tr->mat->emissive ? tr->mat->getEmission(uv) : Vec3f(); // se
              il materiale ĂŠ emissivo allora campiono la texture emissiva
              altrimenti lo setto a 0
          for(auto & l : lights){ // per tutte le luci della scena
            float NoL = n*l->getDirection(pc); // calcolo il prodotto scalare tra
                normale e direzione dela luce
            if(NoL < EPS) // se il punto non guarda la luce
              continue; // salto l'ombreggiatura del punto
            R += NoL*tr->mat->shade(n, h, uv)%l->getAll(pc); // altriment calcolo
                l'ombreggiatura tramite i valori della luce, la brdf e NoL
          }
          R += tr->mat->reflective ?
              tr->mat->getReflection(uv)%background->getColor(wi) : Vec3f();
          draw(i*frameBuffer->width + j, z, clamp(R, 0, 1));
        }
      }
    }
  }
  timer->stop(); // fermiamo il timer
  if(BENCH){ // se il benchmarking ĂŠ attivo
    timer->print(); // stampiamo le tempistiche del rendering del frame
    cout << "\n";
  }

}

float World::edgeFunction(const Vec3f & p, const Vec3f & e) const{ // calcola il
    prodotto vettoriale tra il vettore che unisce il punto e un vertice e lo
    spigolo che ha quel vertice in comune, ci dice se il punto ĂŠ interno e la
    coordinata baricentrica di quel punto rispetto a quello spigolo
  return p.x*e.y - e.x*p.y; // e = spigolo , p = punto da testare, se p ĂŠ
      sinistra dello spigolo (quindi interno) ho un risultato positivo. Mi muovo
      tra i vertici in senso antiorario
}

void World::draw(int i, float depth, const Vec3f & color) const{ // salva il
    valore di radianza e profonditĂ
  if(depth < zBuffer[i]){ // se la profonditĂ  attuale ĂŠ < di quella precedente
    zBuffer[i] = depth; // sostituisco la profonditĂ
    frameBuffer->impress(i, color); // imprimo la radianza sulla pellicola
  }
}

void World::animate(double t){ // gestisce le animazioni del programma

  if(ANIM || t == -1){ // se l'animazione ĂŠ attiva
    if(ATTRACTION)
      attract(objs[0], objs[1]); // calcolo la dinamica di attrazione
    for(auto & i : objs){ // per tutti gli oggetti
```

```
      if(i->anim) // se sono animabili
        i->animate(); // calcolo l'animazione
    }
    for(auto & l : lights){ // per tutte le luci
      if(l->anim) // se sono animabili
        l->animate(); // calcolo l'animazione
    }
  }

  if((m_pos.x != m_l_pos.x || m_pos.y != m_l_pos.y) && MOUSE) // se ho spostato il
      mouse e il mouse Ăš attivo
    moveCamera(Vec3f(), Vec3f(m_pos.x-m_l_pos.x, m_pos.y-m_l_pos.y, 0.f)*100,
        camera->t_dir); // sposto la visuale


  // se mi sto muovendo in una data direzione, sposto la camera in quella direzione
  if(RIGHT)
    moveCamera(camera->l_dir*VEL, Vec3f(), camera->t_dir);
  if(LEFT)
    moveCamera(-camera->l_dir*VEL, Vec3f(), camera->t_dir);
  if(BACK)
    moveCamera(camera->f_dir*VEL, Vec3f(), camera->t_dir);
  if(FRONT)
    moveCamera(-camera->f_dir*VEL, Vec3f(), camera->t_dir);
  if(TOP)
    moveCamera(camera->t_dir*VEL, Vec3f(), camera->t_dir);
  if(BOTTOM)
    moveCamera(-camera->t_dir*VEL, Vec3f(), camera->t_dir);


}



void World::moveCamera(const Vec3f & e, const Vec3f & lA, const Vec3f & up){
  camera->eye += e;
  camera->lookAt += e + lA;
  camera->up = up;
  camera->buildWorldToCam();
}

void World::zoomCamera(float dir){
  camera->fov += dir*5;
  camera->fov = clamp(camera->fov, 0.5, 179.);
  camera->buildPerspective();
}

void World::attract(ObjectGroup * og1, ObjectGroup * og2){ // calcola l'attrazione
    gravitazionale tra due corpi
```

```
  float k = 17.1; // costante necessaria a mantenere un'orbita stabile e circolare
  Vec3f d = og1->bbox->c - og2->bbox->c; // calcolo il vettore che congiunge i due
      corpi
  float dist = d.length(); // calcolo la norma del vettore
  if(dist < 1){ // se la distanza Ăš abbastanza piccola
    og1->vel = Vec3f(); // fermo i due corpi
    og2->vel = Vec3f();
    return;
  }
  d.normalize(); // normalizzo il vettore
  float F = 1./(dist*dist*k); // calcolo la forza di attrazione proporzionale
      all'inverso del quadrato della distanza
  og2->vel += d*F*og1->m; // sommo alla velocitĂ  del corpo la velocitĂ  di
      attrazione
  *og2->anim = translation(og2->vel); // cambio la trasformazione di animazione
      del corpo con una traslazione della velocitĂ  data
}

void World::initVelocity(ObjectGroup * og1, ObjectGroup * og2){ // inizializza la
    velocitĂ  dei due corpi
  float k = .171; // costante necessaria a mantenere un'orbita stabile e circolare
  Vec3f d = (og2->bbox->c - og1->bbox->c); // calcolo il vettore che congiunge i
      due corpi
  float dL = d.length(); // calcolo la sua norma
  d.normalize(); // normalizzo il vettore
  og1->setVelocity(d^Vec3f(0,1,0)*sqrt(2*og2->m/dL)*k); // calcolo la velocitĂ 
      tangenziale e la setto al corpo
}

void World::importPrims(ObjectGroup * model){ // importa le primitive nel buffer
    dei triangoli
  if(model->nPrims > 0)
    for(auto & obj : model->objs)
      triangles.push_back(obj);
  else if(model->nChilds > 0){
    for(auto & child : model->childs)
      importPrims(child);
  }
  cout << model->toString() << "\n";
}

void World::importAllPrims(){ // importa tutte le primitive
  for(auto & i : objs)
    importPrims(i);
}

void World::clearBuffers(){ // metodo che svuota i buffers
  background->fill(frameBuffer); // svuota il framebuffer
  for(int i = 0; i < frameBuffer->size(); ++i) // svuota lo z-buffer
```

```
    zBuffer[i] = INFINITY;
}

void World::resetGlobalVar(){
  BENCH = false;
  MOUSE = false;
  ANIM = false;
  RENDER = true;
  LEFT = false;
  RIGHT = false;
  FRONT = false;
  BACK = false;
  TOP = false;
  BOTTOM = false;
  VEL = .1f, L_SPEED = 1.f, H_SPEED = 3.f;
  m_pos = m_l_pos = Vec3f();
}
```

world.hpp:

```
#ifndef _WORLD
#define _WORLD

#include <vector>
#include <unordered_map>
#include "geometry/vec3f.hpp"
#include "geometry/vec3d.hpp"
#include "geometry/vec3i.hpp"

class Timer;
class Parser;
class Camera;
class Film;
class PixelBuffer;
class Background;
class Triangle;
class ObjectGroup;
class Material;
class Light;

class World{
public:
  World();
  ~World();

  unsigned char * buildScene(const Vec3i &, void (*)(World *));
  void deleteScene();
  void renderScene();
  float edgeFunction(const Vec3f &, const Vec3f &) const;
```

```
  void draw(int, float, const Vec3f &) const;
  void animate(double);
  void test();
  void attract(ObjectGroup *, ObjectGroup *);
  void initVelocity(ObjectGroup *, ObjectGroup *);
  void moveCamera(const Vec3f &, const Vec3f &, const Vec3f &);
  void zoomCamera(float);
  void importPrims(ObjectGroup *);
  void importAllPrims();
  void clearBuffers();
  void resetGlobalVar();

  // booleani per controllare gli eventi da tastiera
  bool BENCH = false;
  bool MOUSE = false;
  bool ANIM = false;
  bool RENDER = true;
  bool LEFT = false;
  bool RIGHT = false;
  bool FRONT = false;
  bool BACK = false;
  bool TOP = false;
  bool BOTTOM = false;
  bool ATTRACTION = true;
  float VEL = .1f, L_SPEED = 1.f, H_SPEED = 3.f;
  Vec3f m_pos, m_l_pos;

  Timer * timer; // puntatore al timer
  Parser * parser; // puntatore al parser
  Camera * camera; // puntatoree alla videocamera
  Film * frameBuffer; // puntatore al frameBuffer
  float * zBuffer; // array dello z-buffer
  std::vector<Triangle *> triangles; // lista di triangoli della scena
  std::vector<Light *> lights; // lista di luci della scena
  std::vector<ObjectGroup *> objs; // lista di modelli 3D della scena
  std::unordered_map<std::string,Material *> materials; // lista di associazione
     tra materiali e il proprio nome
  Background * background; // background del frame
};



#endif
```

gui.hpp:

```
/*
  File di definizione della classe GUI, interfaccia del programma
   che ha il compito di gestire la creazione della finestra e dei suoi componenti.
```

```
*/

#ifndef _GUI
#define _GUI

#include <nanogui/screen.h>
#include <nanogui/window.h>
#include <nanogui/layout.h>
#include <nanogui/label.h>
#include <nanogui/button.h>
#include <nanogui/messagedialog.h>
#include <nanogui/textbox.h>
#include <nanogui/combobox.h>
#if defined(_WIN32)
#include <windows.h>
#endif
#include <nanogui/glutil.h>

#include "world.hpp"
#include "utilities/timer.hpp"
#include "canvas.hpp"


using namespace std;

static World * w;
static const float WIDTH = 1000; // larghezza della finestra
static const float HEIGHT = 1000; // altezza della finestra

class GUI : public nanogui::Screen{
public:
  GUI(): nanogui::Screen({1,1}, "Sistema Terra/Luna - Render"){
    using namespace nanogui;

    w = new World(); // creo World

    // da qui definisco gli oggetti dell'interfaccia
    Button *b;

    Window * container = new Window(this, "");
    container->setLayout(new BoxLayout(Orientation::Horizontal, Alignment::Minimum,
        0, 0));


    Widget * panel = new Widget(container);
    panel->setLayout(new GroupLayout());

    new Label(panel, "Seleziona Scena :", "sans-bold", 18);
```

```
ComboBox * cbox = new ComboBox(panel, { "Sistema Terra/Luna", "Particelle",
    "R2D2", "Mappa di Riflessione"});
cbox->setCallback([this](int i){
  if(scene == i)
    return;
  w->ATTRACTION = false;
  w->RENDER = w->ANIM = false;
  w->deleteScene();
  switch(i){
    case 0:
    w->ATTRACTION = true;
    terra_luna(w);
    break;
    case 1:
    particles(w);
    break;
    case 2:
    r2d2(w);
    break;
    case 3:
    reflection(w);
    break;
  }
  w->resetGlobalVar();
  resetStats();
  scene = i;
});

new Label(panel, "Performance :", "sans-bold", 18);
fps_box = new TextBox(panel);
fps_box->setEditable(false);
fps_box->setUnits("fps");
fps_box->setDefaultValue("0");
fps_box->setValue("0");
ms_box = new TextBox(panel);
ms_box->setEditable(false);
ms_box->setUnits("ms");
ms_box->setDefaultValue("0");
ms_box->setValue("0");

new Label(panel, "Informazioni :", "sans-bold", 18);
b = new Button(panel, "Scorciatoie Tastiera");
b->setCallback([&]{
  auto dlg = new MessageDialog(this, MessageDialog::Type::Information,
    "Scorciatoie da Tastiera",
```

```
      "Muovere la videocamera:\n W -> Avanti\n S -> Indietro\n A -> Sinistra\n D
         -> Destra\n Left_Alt -> Sopra\n Left_Ctrl-> Sotto\n Left_Shift ->
         Accelera\n Mouse_Weel -> zoom\n\nAnimazione:\n Space -> Animazione
         On/Off\n Right_Arrow -> Avanza di un Frame\n\nM -> Abilita Puntamento
         Mouse\n\nTab -> Stampa Benchmark su Terminale");
  });
  b = new Button(panel, "Crediti");
  b->setCallback([&]{
    auto dlg = new MessageDialog(this, MessageDialog::Type::Information,
      "Crediti",
      "Sistema Terra/Luna\n\nUn semplice motore di rendering in tempo reale che
         riproduce le tecniche di Z-Buffer e Rastering di triangoli.\n\n Creato
         da Marco Petreri per Metodi Matematici in Computer Graphics - Scienze e
         Tecnologie per i Media.");
  });

  c = new Canvas(container, {WIDTH, HEIGHT}, w); // creo la canvas, componente
      che mi permette di disegnare su schermo i rendering
  c->setCursor(Cursor::Crosshair);

  performLayout();

  setSize(container->size());


  cbox->popup()->setAnchorPos({panel->size().x()+15,cbox->popup()->anchorPos().y()});

  }

~GUI(){
  delete w;
}

virtual bool keyboardEvent(int key, int scancode, int action, int modifiers){ //
    action listener della tastiera
    if (Screen::keyboardEvent(key, scancode, action, modifiers))
        return true;
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS){
        setVisible(false);
        return true;
    }
    if(key == GLFW_KEY_W && action == GLFW_PRESS){
      w->FRONT = true;
      return true;
    }
    if(key == GLFW_KEY_A && action == GLFW_PRESS){
      w->LEFT = true;
      return true;
    }
```

```
if(key == GLFW_KEY_S && action == GLFW_PRESS){
  w->BACK = true;
  return true;
}
if(key == GLFW_KEY_D && action == GLFW_PRESS){
  w->RIGHT = true;
  return true;
}
if(key == GLFW_KEY_LEFT_ALT && action == GLFW_PRESS){
  w->TOP = true;
  return true;
}
if(key == GLFW_KEY_LEFT_CONTROL && action == GLFW_PRESS){
  w->BOTTOM = true;
  return true;
}
if(key == GLFW_KEY_LEFT_SHIFT && action == GLFW_PRESS){
  w->VEL = w->H_SPEED;
  return true;
}
if(key == GLFW_KEY_W && action == GLFW_RELEASE){
  w->FRONT = false;
  return true;
}
if(key == GLFW_KEY_A && action == GLFW_RELEASE){
  w->LEFT = false;
  return true;
}
if(key == GLFW_KEY_S && action == GLFW_RELEASE){
  w->BACK = false;
  return true;
}
if(key == GLFW_KEY_D && action == GLFW_RELEASE){
  w->RIGHT = false;
  return true;
}
if(key == GLFW_KEY_LEFT_ALT && action == GLFW_RELEASE){
  w->TOP = false;
  return true;
}
if(key == GLFW_KEY_LEFT_CONTROL && action == GLFW_RELEASE){
  w->BOTTOM = false;
  return true;
}
if(key == GLFW_KEY_LEFT_SHIFT && action == GLFW_RELEASE){
  w->VEL = w->L_SPEED;
  return true;
}
if(key == GLFW_KEY_SPACE && action == GLFW_PRESS){
```

```
      w->ANIM = !w->ANIM;
      return true;
    }
    if(key == GLFW_KEY_TAB && action == GLFW_PRESS){
      w->BENCH = !w->BENCH;
      return true;
    }
    if(key == GLFW_KEY_M && action == GLFW_PRESS){
      w->MOUSE = !w->MOUSE;
      return true;
    }
    if(key == GLFW_KEY_RIGHT && action == GLFW_PRESS){
      w->animate(-1);
      return true;
    }
    return false;
}


virtual void draw(NVGcontext *ctx){
  updateMouse(mousePos());

  Screen::draw(ctx);

  updatePerformance();
}

void updateMouse(const Eigen::Vector2i & _mp){ // aggiorna la posizione del
    mouse all'interno della finestra operando anche un cambio di coordinate
  Vec3f mp(_mp.x(), _mp.y(), 0.);
  w->m_l_pos = w->m_pos;
  w->m_pos.x = 2*mp.x/WIDTH - 1;
  w->m_pos.y = 1 - 2*mp.y/HEIGHT;
}

void updatePerformance(){ // aggiorna i valori di fps e frequenza di rendering
    della finestra
  ++count %= step;
  fps += w->timer->getFPS();
  ms += w->timer->getTime();

  if(count == 0){
    fps /= step; ms /= step;
    fps_box->setValue(std::to_string(fps));
    ms_box->setValue(std::to_string(ms));
    fps = ms = 0;
  }
}
```

```
  void resetStats(){ // resetta le statistiche
    count = 0; fps = 0; ms = 0;
  }

  int step = 20, count = 0, fps = 0, ms = 0;
  int scene = 0;
  nanogui::TextBox * fps_box, * ms_box;
  Canvas * c;
  World * w;
};

#endif
```

CAMERAS AND CANVAS:

camera.cpp:

```
/*
  File di definizione della classe Camera, modella una generica videocamera che ha
     il compito di descrivere come l'osservatore guarda la scena
*/

#include "camera.hpp"
#include "../utilities/math.hpp"

using namespace Matrix4D;

Camera::Camera(): width(800), heigth(600), near(1e-5), far(1e5), fov(90)
               , eye(0., 0., 100.), lookAt(0.), up(0., 1., 0.){
  buildWorldToCam();
  buildPerspective();
  buildNDCToRaster();
}

Camera::Camera(int w, int h, float n, float f, float FOV, const Vec3f & e,
             const Vec3f & l, const Vec3f & u): width(w), heigth(h), near(n)
             , far(f), fov(FOV), eye(e), lookAt(l), up(u.hat()){
  buildWorldToCam();
  buildPerspective();
  buildNDCToRaster();
}

Camera::Camera(const Camera & c): width(c.width), heigth(c.heigth), near(c.near),
    far(c.far)
             , fov(c.fov), eye(c.eye), lookAt(c.lookAt), up(c.up){
  worldToCam = c.worldToCam;
  perspective = c.perspective;
}
```

```
Camera::~Camera(){}

Camera & Camera::operator=(const Camera & c){
  width = c.width; heigth = c.heigth; near = c.near; far = c.far; fov = c.fov;
  eye = c.eye; lookAt = c.lookAt; up = c.up;
  worldToCam = c.worldToCam; perspective = c.perspective;
  return *this;
}

void Camera::buildWorldToCam(){ // metodo che costruisce la matrice di cambiamento
    di base tra camera e mondo
  f_dir = (eye - lookAt).hat(); // si parte costruendo il vettore di osservazione
      ribaltato
  Vec3f t = up^f_dir; // si opera un prodotto vettoriale con il vettore del cielo
      per trovare un suo vettore ortonormale dipendente da questi due
  l_dir = (t == Vec3f()) ? (up - Vec3f(1e-6).hat())^w : t; // se il risultato Ăš
      un vettore nullo ( dovuto a una scelta incorretta del vettore up ) si
      perturba l'up con un valore piccolo e si ripete il calcolo
  t_dir = f_dir^l_dir; // si opera l'ultimo prodotto vettoriale tra i due vettori
      calcolati precedentemente
  worldToCam = Mat4f(l_dir,t_dir,f_dir,eye); // costruisco la matrice CamToWorld
  worldToCam.inverse(); // la inverto passando cos\'{i} alla WorldToCam
}

void Camera::buildPerspective(){ // metodo che costruisce la matrice di proiezione
  float s = 1./tan(toRad(fov*.5)); // calcolo il fattore di scaling derivante dal
      fov
  perspective = Mat4f( s, 0, 0, 0, // costruisco la matrice di proiezione
                       0, s, 0, 0,
                       0, 0, -1, 0,
                       0, 0, -1, 0);
}

void Camera::buildNDCToRaster(){ // metodo che costruisce la matrice di
    cambiamento di base tra coordinate normalizzate e coordinate sullo schermo
  ndcToRaster = scaling(width*.5, heigth*.5, 1.)*translation(1., 1,
      0.)*scaling(1.,-1.,1.); // costruisco la matrice tramite composizione
}

std::string Camera::toString() const{
  return "{ Res: " + std::to_string(width) + "x" + std::to_string(heigth)
         + "\nNear: " + std::to_string(near) + ", Far: " + std::to_string(far) +
             ", FOV: " + std::to_string(fov)
         + "\nEye: " + eye.toString() + ", LookAt: " + lookAt.toString() + ", Up:
             " + up.toString() + " }";
}
```

---

camera.hpp:

```
#ifndef _CAMERA
#define _CAMERA

#include <iostream>
#include "../geometry/vec3f.hpp"
#include "../geometry/mat4f.hpp"


class Camera{
public:
  Camera();
  Camera(int, int, float, float, float, const Vec3f &, const Vec3f &, const Vec3f
      &);
  Camera(const Camera &);
  ~Camera();

  Camera & operator=(const Camera &);

  void buildWorldToCam();
  void buildCamToRaster();
  void buildPerspective();
  void buildNDCToRaster();

  std::string toString() const;

  int width, heigth; // larghezza e altezza dell'immagine
  float near, far; // distanza minima e massima di visuale
  float fov; // field of view
  Vec3f eye, lookAt, up; // posizione della camera, punto osservato, vettore
      indicante il cielo
  Mat4f worldToCam; // matrice di cambiamento di base dalle coordinate del mondo a
      quelle della camera
  Mat4f perspective; // matrice di cambiamento di proiezione
  Mat4f ndcToRaster; // matrice di cambiamento di base dalle coordinate
      normalizzate a quelle dello schermo
};

#endif
```

canvas.hpp:

```
/*
  File di definizione della classe Canvas, componente dell'interfaccia
   che ha il compito di disegnare a schermo i frame e gestire la creazione della
       scena.
*/


#ifndef _CANVAS
```

```
#define _CANVAS


#include <nanogui/glcanvas.h>
#include "world.hpp"
#include "utilities/buildFunctions.hpp"

class Canvas : public nanogui::GLCanvas {
public:
  Canvas(Widget *parent, const Eigen::Vector2i & s, World * _w) :
      nanogui::GLCanvas(parent), size(Vec3i(s.x(), s.y(), 0)), w(_w){
      using namespace nanogui;
      setSize(s); // setto la grandezza della finestra

      buffer = w->buildScene(size, &terra_luna); // costruisco la scena di default
          e mi salvo il riferimento al buffer del frame
      w->renderScene(); // renderizzo il primo frame

      // inizio gestione disegno in opengl
      glGenTextures(1, &texture);
      glBindTexture(GL_TEXTURE_2D, texture);
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, size.x, size.y, 0, GL_RGB,
          GL_UNSIGNED_BYTE, buffer);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
      glEnable(GL_TEXTURE_2D);

      mShader.init(
          /* An identifying name */
          "a_simple_shader",

          /* Vertex shader */
          "#version 330\n"
          "in vec3 position;\n"
          "in vec2 texture;\n"
          "out vec2 tex;\n"
          "void main() {\n"
          "    tex = texture;\n"
          "    gl_Position = vec4(position, 1.0);\n"
          "}",

          /* Fragment shader */
          "#version 330\n"
          "uniform sampler2D txt;\n"
          "in vec2 tex;\n"
          "out vec4 color;\n"
          "void main() {\n"
          "    color = texture(txt, tex);\n"
          "}"
```

```
    );

    MatrixXu indices(3, 2);
    indices.col(0) << 0, 1, 2;
    indices.col(1) << 2, 1, 3;

    MatrixXf positions(3, 4);
    positions.col(0) << -1, -1, 0;
    positions.col(1) << 1, -1, 0;
    positions.col(2) << -1, 1, 0;
    positions.col(3) << 1, 1, 0;

    MatrixXf text_uv(2, 4);
    text_uv.col(0) << 0, 1;
    text_uv.col(1) << 1, 1;
    text_uv.col(2) << 0, 0;
    text_uv.col(3) << 1, 0;

    mShader.bind();
    mShader.uploadIndices(indices);
    mShader.uploadAttrib("texture", text_uv);
    mShader.uploadAttrib("position", positions);

    // fine gestione disegno opengl
}

~Canvas() {
    mShader.free();
}

virtual void drawGL() override { // gestisce il disegno dei frame

  w->animate(glfwGetTime()); // calcolo l'animazione del frame
  if(w->RENDER)
    w->renderScene(); // renderizzo il frame

  mShader.bind();
  glBindTexture(GL_TEXTURE_2D, texture);
  glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, size.x, size.y, GL_RGB,
      GL_UNSIGNED_BYTE, buffer);
  mShader.drawIndexed(GL_TRIANGLES, 0, 2);

  w->clearBuffers(); // svuoto i buffers per il nuovo frame
}

virtual bool scrollEvent(const Eigen::Vector2i &p, const Eigen::Vector2f &rel){
    // listener per lo scroll
  w->zoomCamera(-rel.y()); // applica lo zoom alla camera
}
```

```
  GLuint texture;
  nanogui::GLShader mShader;
  unsigned char * buffer;
  Vec3i size;
  World * w;
};


#endif
```

GEOMETRY:

aabb.hpp:

```
/*
  File di definizione della classe AABB, descrive un Axis Aligned Bounding Box.
*/

#ifndef _AABB_
#define _AABB_

#include <iostream>
#include "../geometry/vec3f.hpp"
#include "../utilities/math.hpp"


class AABB{
public:
  AABB(): min(-INFINITY), max(INFINITY), c(0.f){}
  AABB(const Vec3f & v): min(-v), max(v){
    if(min > max){
      min = v;
      max = -v;
    }
    c = min + diag()*.5;
  }
  AABB(const Vec3f & _min, const Vec3f _max): min(_min), max(_max){
    if(min > max){
      min = _max;
      max = _min;
    }
    c = min + diag()*.5;
  }
  AABB(const AABB & b): min(b.min), max(b.max), c(b.c){}

  AABB operator+(const AABB & b) const{ // calcola l'unione insiemistica tra due
      AABB
```

```
    Vec3f tmin, tmax;
    for(int i = 0; i < 3; ++i){
      tmin[i] = b.min[i] < min[i] ? b.min[i] : min[i];
      tmax[i] = b.max[i] > max[i] ? b.max[i] : max[i];
    }
    return AABB(tmin, tmax);
  }
  AABB operator+(const std::initializer_list<Vec3f> & b) const{
    return *this + AABB(*b.begin(), *b.end());
  }
  AABB & operator+=(const AABB & b){
    for(int i = 0; i < 3; ++i){
      min[i] = min[i] < b.min[i] ? min[i] : b.min[i];
      max[i] = max[i] > b.max[i] ? max[i] : b.max[i];
    }
    c = min + diag()*.5;
    return *this;
  }
  AABB & operator+=(const std::initializer_list<Vec3f> & b){
    return *this += AABB(*b.begin(), *b.end());
  }
  AABB & operator=(const AABB & b){min = b.min; max = b.max; c = b.c; return
      *this;}

  Vec3f diag() const{ // calcola la diagonale del AABB
    return max - min;
  }

  std::string toString() const{
    return "[ " + min.toString() + ", " + max.toString() + ", Extention: " +
        diag().toString() + " , Center: " + c.toString() + "]";
  }


  Vec3f min, max, c; // la descrizione del AABB avviene tramite il suo punto di
      minimo e di massimo, centro dell'AABB
};


inline std::ostream & operator<<(std::ostream & os, const AABB & b){
  return os << "AABB : { " << b.min << ", " << b.max << " }";
}

#endif
```

mat4f.hpp:

```
#ifndef _MAT_4D
#define _MAT_4D
```

```
#include "vec3f.hpp"
#include "vec4f.hpp"
#include <string.h>
#include <stdlib.h>
#include <Eigen/Dense>

class Mat4f{
public:
  Mat4f(){memset(m, 0., sizeof(float)*16); m[0] = m[5] = m[10] = m[15] = 1.;}
  Mat4f(float s){memset(m, 0., sizeof(float)*16); m[0] = m[5] = m[10] = m[15] = s;}
  Mat4f(float v[16]): m(v){}
  Mat4f(float m0, float m1, float m2, float m3,
        float m4, float m5, float m6, float m7,
        float m8, float m9, float m10, float m11,
        float m12, float m13, float m14, float m15){
          m[0]  = m0; m[1] = m4; m[2] = m8; m[3] = m12;
          m[4]  = m1; m[5] = m5; m[6] = m9; m[7] = m13;
          m[8]  = m2; m[9] = m6; m[10] = m10; m[11] = m14;
          m[12] = m3; m[13] = m7; m[14] = m11; m[15] = m15;
        }
  Mat4f(const Vec3f & u, const Vec3f & v, const Vec3f & z, const Vec3f & w){
    m[0]  = u.x; m[1] = u.y; m[2] = u.z; m[3] = 0.;
    m[4]  = v.x; m[5] = v.y; m[6] = v.z; m[7] = 0.;
    m[8]  = z.x; m[9] = z.y; m[10] = z.z; m[11] = 0.;
    m[12] = w.x; m[13] = w.y; m[14] = w.z; m[15] = 1.;
  }
  Mat4f(const Vec4f & u, const Vec4f & v, const Vec4f & z, const Vec4f & w){
    m[0]  = u.x; m[1] = u.y; m[2] = u.z; m[3] = u.w;
    m[4]  = v.x; m[5] = v.y; m[6] = v.z; m[7] = v.w;
    m[8]  = z.x; m[9] = z.y; m[10] = z.z; m[11] = z.w;
    m[12] = w.x; m[13] = w.y; m[14] = w.z; m[15] = w.w;
  }
  Mat4f(const Mat4f & n){memcpy(m, n.m, sizeof(float)*16);}
  Mat4f(Mat4f && n){delete [] m; m = n.m; n.m = nullptr;}
  ~Mat4f(){delete [] m;}

  Mat4f operator*(const Mat4f & n) const{ // moltiplicazione tra matrici
    Mat4f t;
    for(int i = 0; i < 4; ++i)
      for(int j = 0; j < 4; ++j)
        t(j,i) = n.m[4*i]*m[j] + n.m[4*i+1]*m[j+4] + n.m[4*i+2]*m[j+8] +
            n.m[4*i+3]*m[j+12];
    return t;
  }
  Mat4f & operator*=(const Mat4f & n){
    Mat4f t;
    for(int i = 0; i < 4; ++i)
      for(int j = 0; j < 4; ++j)
```

```
        t(j,i) = n.m[4*i]*m[j] + n.m[4*i+1]*m[j+4] + n.m[4*i+2]*m[j+8] +
            n.m[4*i+3]*m[j+12];
    return *this = t;
  }
  Vec4f operator*(const Vec4f & v) const{ // moltiplicazione matrice vettore
    Vec4f t(m[0]*v.x + m[4]*v.y + m[8]*v.z + m[12]*v.w,
            m[1]*v.x + m[5]*v.y + m[9]*v.z + m[13]*v.w,
            m[2]*v.x + m[6]*v.y + m[10]*v.z + m[14]*v.w,
            m[3]*v.x + m[7]*v.y + m[11]*v.z + m[15]*v.w);
    return t;
  }
  Mat4f & operator=(const Mat4f & n){memcpy(m, n.m, sizeof(float)*16); return
      *this;}
  Mat4f & operator=(Mat4f && n){delete [] m; m = n.m; n.m = nullptr; return *this;}
  float operator()(int r, int c) const{return m[c*4 + r];} // ritorna l'elemento
      di riga r e colonna c
  float & operator()(int r, int c){return m[c*4 + r];}
  float operator[](int r) const{return m[r*4];}
  float & operator[](int r){return m[r*4];}

  Vec4f row(int i) const{return Vec4f(m[i], m[i+4], m[i+8], m[i+12]);} // ritorna
      la riga i
  Vec4f col(int i) const{return Vec4f(m[i*4], m[i*4+1], m[i*4+2], m[i*4+3]);} //
      ritorna la colonna i
  Mat4f & transpose(){ // transpone la matrice
    Mat4f t;
    for(int i = 0; i < 16; ++i){
      div_t d = div(i,4);
      t.m[i] = m[d.quot + (d.rem)*4];
    }
    return *this = t;
  }
  Mat4f & inverse();
  Mat4f & inverse2();

  float * m = new float[16];
};

std::ostream & operator<<(std::ostream &, const Mat4f &);

namespace Matrix4D{
  Mat4f transpose(const Mat4f &);
  Mat4f inverse(const Mat4f &);
  Mat4f translation(float, float, float);
  Mat4f scaling(float);
  Mat4f scaling(float, float, float);
  Mat4f rotationX(float);
  Mat4f rotationY(float);
  Mat4f rotationZ(float);
```

```
}

#endif
```

objectGroup.hpp:

```
/*
  File di definizione della classe ObjectGroup, modella un oggetto complesso
      costituito da un albero di primitive
*/

#ifndef _OBJECT_GROUP_
#define _OBJECT_GROUP_

#include <vector>
#include "triangle.hpp"
#include "transform.hpp"
#include "aabb.hpp"

class ObjectGroup{
public:
  ObjectGroup(): nPrims(0), nTotPrims(0), nChilds(0), o2w(new Transform()),
      anim(nullptr), bbox(new AABB(Vec3f())){}
  ObjectGroup(Transform * t): nPrims(0), nTotPrims(0), nChilds(0), o2w(t),
      anim(nullptr), bbox(new AABB(Vec3f())){}
  ObjectGroup(const ObjectGroup & og): nPrims(og.nPrims), nTotPrims(og.nTotPrims),
      nChilds(og.nChilds){
    objs.reserve(og.nPrims);
    for(u_int i = 0; i < og.nPrims; ++i){
      *objs[i] = *og.objs[i];
    }
    childs.reserve(og.nChilds);
    for(u_int i = 0; i < og.nChilds; ++i){
      *childs[i] = *og.childs[i];
    }
    *o2w = *og.o2w;
    *bbox = *og.bbox;
  }
  ~ObjectGroup(){
    for(auto & x : objs)
      delete x;
    for(auto & x : childs)
      delete x;
    delete o2w;
    delete anim;
    delete bbox;
  }

  ObjectGroup & operator=(const ObjectGroup & og){
```

```
    for(auto & x : objs)
      delete x;
    for(auto & x : childs)
      delete x;
    objs.clear();
    childs.clear();
    if(og.nPrims > 0)
      objs.reserve(og.nPrims);
    if(og.nChilds > 0)
    childs.reserve(og.nChilds);
    for(u_int i = 0; i < og.nPrims; ++i){
      *objs[i] = *og.objs[i];
    }
    for(u_int i = 0; i < og.nChilds; ++i){
      *childs[i] = *og.childs[i];
    }
    nPrims = og.nPrims;
    nTotPrims = og.nTotPrims;
    nChilds = og.nChilds;
    *o2w = *og.o2w;
    *bbox = *og.bbox;
    return *this;
  }

  void addObject(Triangle * p){ // aggiunge una primitiva all'array di primitive
    objs.push_back(p);
    ++nPrims;
    ++nTotPrims;
  }

  void addChild(ObjectGroup * og){ // aggiunge un ObjectGroup all'array di figli
    childs.push_back(og);
    nTotPrims += og->nPrims;
    ++nChilds;
  }

  void setVelocity(const Vec3f & v){ // metodo che setta la velocitĂ  del corpo
    vel = v;
  }

  void setMass(float _m){ // metodo che setta la massa del corpo
    m = _m;
  }

  void setAnimation(Transform * tr){ // metodo che setta l'animazione del corpo
    anim = tr;
  }
```

```
AABB setBVertex(){ // metodo che calcola ricorsivamente il bounding box
    dell'ObjectGroup
  AABB tbox;
  if(nPrims > 0){ // se ci sono primitive
    objs[0]->setBVertex(); // calcolo il bounding box della prima primitiva
    tbox = AABB(objs[0]->min, objs[0]->max); // creo un AABB dal min e max del
        bounding box della primitiva
    for(u_int i = 1; i < nPrims; ++i){ // per tutte le primitive restanti
      objs[i]->setBVertex(); // calcolo il bounding box della prima primitiva
      AABB primBox(objs[i]->min, objs[i]->max); // creo un AABB dal min e max del
          bounding box della primitiva
      tbox += primBox; // faccio l'unione insiemistica dei due AABB
    }
  }
  else if(nChilds > 0){ // se ha dei figli
    tbox = childs[0]->setBVertex(); // calcolo il bounding box del primo figlio
    for(u_int i = 1; i < nChilds; ++i){ // per tutti i figli restanti
      tbox += childs[i]->setBVertex(); // faccio l'unione insiemistica dei due
          AABB
    }
  }
  *bbox = tbox; // setto l'AABB
  return *bbox; // ritorno l'AABB
}

void applyTransform(Transform * tr){ // metodo che applica ricorsivamente una
    trasformazione a tutto l'albero
  Transform comp = tr ? *tr : Transform(); // se tr non esiste allora uso la
      trasformazione identitĂ
  if(nPrims > 0){ // se ci sono primitive
    for(auto & x : objs){ // per tutte le primitive
      x->applyTransform(&comp); // applico la trasformazione alla primitiva
    }
  }
  else if(nChilds > 0){ // se ci sono figli
    for(auto & x : childs){ // per tutti i figli
      x->applyTransform(&comp); // richiamo la funzione sui figli
    }
  }
}

void animate(){ // metodo che anima il corpo
  applyTransform(anim); // applico la trasformazione di animazione
  setBVertex(); // ricalcolo il boundingbox
}

std::string toString() const{
  if(nPrims > 0){
```

```
      return "Object: [ nTotPrims: " + std::to_string(nTotPrims) + ", nPrims: " +
          std::to_string(nPrims) + ", BBox: " + bbox->toString() +
       ", Vel: " + vel.toString() + ", Mass: " + std::to_string(m) + "]";
    }
    else if(nChilds > 0){
      std::string s = "Group: [ ";
      for(auto & i : childs)
        s += i->toString() + ",\n\t\t";
      s += "]";
      s += ", BBox: " + bbox->toString() + "]";
      return s;
    }
  }

  u_int nPrims, nTotPrims, nChilds; // numero di primitive contenute in questo
      livello, numero di primitive contenute in tutti i livelli sottostanti, numero
      di figli
  std::vector<Triangle *> objs; // vettore di primitive
  std::vector<ObjectGroup *> childs; // vettore di figli
  Transform * o2w, * anim; // trasformazione dell'oggetto e trasformazione di
      animazione
  AABB * bbox; // bounding box dell'oggetto
  Vec3f vel; // velocitĂ  dell'oggetto
  float m; // massa dell'oggetto
};

#endif
```

transform.cpp:

```
/*
  File di dichiarazione della classe Transform, modella una trasformazione
      geometrica nello spazio
*/

#include "transform.hpp"
#include "../utilities/math.hpp"

Transform::Transform(): m(new Mat4f()), mInv(new Mat4f()), mTrInv(new Mat4f()){}

Transform::Transform(const Vec3f & x, const Vec3f & y, const Vec3f & z, const
    Vec3f & w): m(new Mat4f(x, y, z, w)){
  mInv = new Mat4f(x, y, z, w); mInv->inverse();
  mTrInv = new Mat4f(*mInv); mTrInv->transpose();
}

Transform::Transform(const Vec4f & x, const Vec4f & y, const Vec4f & z, const
    Vec4f & w): m(new Mat4f(x, y, z, w)){
  mInv = new Mat4f(x, y, z, w); mInv->inverse();
```

```
  mTrInv = new Mat4f(*mInv); mTrInv->transpose();
}

Transform::Transform(float m0, float m1, float m2, float m3,
                     float m4, float m5, float m6, float m7,
                     float m8, float m9, float m10, float m11,
                     float m12, float m13, float m14, float m15){
                       m = new Mat4f(m0, m1, m2, m3,
                                     m4, m5, m6, m7,
                                     m8, m9, m10, m11,
                                     m12, m13, m14, m15);
                       mInv = new Mat4f(*m);
                       mInv->inverse();
                       mTrInv = new Mat4f(*mInv); mTrInv->transpose();
}

Transform::Transform(Mat4f * _m): m(_m), mInv(new Mat4f(*_m)){
  mInv->inverse();
  mTrInv = new Mat4f(*mInv); mTrInv->transpose();
}

Transform::Transform(Mat4f * _m, Mat4f * _mInv, Mat4f * _mTrInv): m(_m),
    mInv(_mInv), mTrInv(_mTrInv){}

Transform::Transform(const Transform & tr){
  m = new Mat4f(*tr.m); mInv = new Mat4f(*tr.mInv); mTrInv = new Mat4f(*tr.mTrInv);
}

Transform::~Transform(){
  delete m; delete mInv; delete mTrInv;
}

Transform Transform::operator*(const Transform & tr) const{ // composizione di
    trasformazioni
  return Transform(new Mat4f(m->operator*(*tr.m))
                  ,new Mat4f(mInv->operator*(*tr.mInv))
                  ,new Mat4f(mTrInv->operator*(*tr.mTrInv)));
}

Transform & Transform::operator*=(const Transform & tr){
  m->operator*=(*tr.m); mInv->operator*=(*tr.mInv); mTrInv->operator*=(*tr.mTrInv);
  return *this;
}

Transform & Transform::operator=(const Transform & tr){
  *m = *tr.m; *mInv = *tr.mInv; *mTrInv = *tr.mTrInv;
  return *this;
}
```

```
void Transform::inverse(){// inverte la trasformazione
  std::swap(m,mInv);
  *mTrInv = *m; mTrInv->transpose();
}

Transform translation(const Vec3f & v){ // crea una traslazione
  return Transform(new Mat4f(1., 0., 0., v.x
                            ,0., 1., 0., v.y
                            ,0., 0., 1., v.z
                            ,0., 0., 0., 1.)
                  ,new Mat4f(1., 0., 0., -v.x
                            ,0., 1., 0., -v.y
                            ,0., 0., 1., -v.z
                            ,0., 0., 0., 1.)
                  ,new Mat4f(1., 0., 0., 0.
                            ,0., 1., 0., 0.
                            ,0., 0., 1., 0.
                            ,0., 0., 0., 1.)
                  );
}

Transform translation(float a, float b, float c){ // crea una traslazione
  return Transform(new Mat4f(1., 0., 0., a
                            ,0., 1., 0., b
                            ,0., 0., 1., c
                            ,0., 0., 0., 1.)
                  ,new Mat4f(1., 0., 0., -a
                            ,0., 1., 0., -b
                            ,0., 0., 1., -c
                            ,0., 0., 0., 1.)
                  ,new Mat4f(1., 0., 0., 0.
                            ,0., 1., 0., 0.
                            ,0., 0., 1., 0.
                            ,0., 0., 0., 1.)
                  );
}

Transform scaling(float s){ // crea uno scaling
  float d = 1./s;
  return Transform(new Mat4f( s, 0., 0., 0.
                            ,0.,  s, 0., 0.
                            ,0., 0., s, 0.
                            ,0., 0., 0., 1.)
                  ,new Mat4f( d, 0., 0., 0.
                            ,0.,  d, 0., 0.
                            ,0., 0., d, 0.
                            ,0., 0., 0., 1.)
                  ,new Mat4f(1., 0., 0., 0.
                            ,0., 1., 0., 0.
```

```
                        ,0., 0., 1., 0.
                        ,0., 0., 0., 1.)
                );
}

Transform scaling(float a, float b, float c){ // crea uno scaling
  return Transform(new Mat4f( a, 0., 0., 0.
                        ,0.,  b, 0., 0.
                        ,0., 0., c, 0.
                        ,0., 0., 0., 1.)
                ,new Mat4f(1./a, 0.,  0., 0.
                        , 0., 1./b,  0., 0.
                        , 0.,  0., 1./c, 0.
                        , 0.,  0.,  0., 1.)
                ,new Mat4f(1./a, 0.,  0., 0.
                        , 0., 1./b,  0., 0.
                        , 0.,  0., 1./c, 0.
                        , 0.,  0.,  0., 1.)
                );
}

Transform rotationX(float deg){ // crea una rotazione intorno all'asse X
  float sinT = sin(toRad(deg));
  float cosT = cos(toRad(deg));
  return Transform(new Mat4f(1., 0.,  0., 0.
                        ,0., cosT, -sinT, 0.
                        ,0., sinT, cosT, 0.
                        ,0.,  0.,   0., 1.)
                ,new Mat4f(1., 0.,   0., 0.
                        ,0., cosT, sinT, 0.
                        ,0., -sinT, cosT, 0.
                        ,0.,   0.,  0., 1.)
                ,new Mat4f(1., 0.,   0., 0.
                        ,0., cosT, -sinT, 0.
                        ,0., sinT, cosT, 0.
                        ,0.,  0.,   0., 1.)
                );
}

Transform rotationY(float deg){ // crea una rotazione intorno all'asse Y
  float sinT = sin(toRad(deg));
  float cosT = cos(toRad(deg));
  return Transform(new Mat4f( cosT, 0., sinT, 0.
                        ,  0., 1.,  0., 0.
                        ,-sinT, 0., cosT, 0.
                        ,  0., 0.,  0., 1.)
                ,new Mat4f(cosT, 0., -sinT, 0.
                        , 0., 1.,   0., 0.
                        ,sinT, 0., cosT, 0.
```

```
                        , 0., 0.,   0., 1.)
                ,new Mat4f( cosT, 0., sinT, 0.
                        ,   0., 1.,  0., 0.
                        ,-sinT, 0., cosT, 0.
                        ,   0., 0.,  0., 1.)
                );
}

Transform rotationZ(float deg){ // crea una rotazione intorno all'asse Z
  float sinT = sin(toRad(deg));
  float cosT = cos(toRad(deg));
  return Transform(new Mat4f(cosT, -sinT, 0., 0.
                        ,sinT, cosT, 0., 0.
                        , 0.,   0., 1., 0.
                        , 0.,   0., 0., 1.)
                ,new Mat4f( cosT, sinT, 0., 0.
                        ,-sinT, cosT, 0., 0.
                        ,   0.,  0., 1., 0.
                        ,   0.,  0., 0., 1.)
                ,new Mat4f(cosT, -sinT, 0., 0.
                        ,sinT, cosT, 0., 0.
                        , 0.,   0., 1., 0.
                        , 0.,   0., 0., 1.)
                );
}

Transform rotation(const Vec3f & a, float deg){ // crea una rotazione intorno
    all'asse a
  float sinT = sin(toRad(deg));
  float cosT = cos(toRad(deg));
  Mat4f m(a.x*a.x + (1 - a.x*a.x)*cosT, (1 - cosT)*a.x*a.y - sinT*a.z, (1 -
      cosT)*a.x*a.z + sinT*a.y, 0.
        ,(1 - cosT)*a.y*a.x + sinT*a.z, a.y*a.y + (1 - a.y*a.y)*cosT, (1 -
           cosT)*a.y*a.z - sinT*a.x, 0.
        ,(1 - cosT)*a.z*a.x - sinT*a.y, (1 - cosT)*a.z*a.y + sinT*a.x, a.z*a.z + (1
           - a.z*a.z)*cosT, 0.
        , 0., 0., 0., 1.);
  return Transform(new Mat4f(m)
                ,new Mat4f(Matrix4D::inverse(m))
                ,new Mat4f(m)
                );
}
```

---

transform.hpp:

```
#ifndef _TRANSFORM
#define _TRANSFORM

#include "mat4f.hpp"
```

```
class Vec3f;

class Transform{
public:
  Transform();
  Transform(const Vec3f &, const Vec3f &, const Vec3f &, const Vec3f &);
  Transform(const Vec4f &, const Vec4f &, const Vec4f &, const Vec4f &);
  Transform(float, float, float, float,
            float, float, float, float,
            float, float, float, float,
            float, float, float, float);
  Transform(Mat4f *);
  Transform(Mat4f *, Mat4f *, Mat4f *);
  Transform(const Transform &);
  ~Transform();

  Transform operator*(const Transform &) const;
  Transform & operator*=(const Transform &);
  Transform & operator=(const Transform &);

  void inverse();

  Mat4f * m, * mInv, * mTrInv; // matrice, inversa e inversa trasposta
};

Transform translation(const Vec3f &);
Transform translation(float, float, float);
Transform scaling(float);
Transform scaling(float, float, float);
Transform rotationX(float);
Transform rotationY(float);
Transform rotationZ(float);
Transform rotation(const Vec3f &, float);



#endif
```

triangle.hpp:

```
/*
  File di definizione della classe Triangle, modella un triangolo
*/

#ifndef _TRIANGLE
#define _TRIANGLE

#include "vec3f.hpp"
```

```
#include "transform.hpp"
#include "../utilities/material.hpp"
#include "../utilities/constants.hpp"

class Triangle{
public:
  Triangle(): v0(), v1(), v2(){}
  Triangle(const Vec3f & _v0, const Vec3f & _v1, const Vec3f & _v2)
  : v0(_v0), v1(_v1), v2(_v2), n((v1-v0)^(v2-v0)){}
  Triangle(const Vec3f & _v0, const Vec3f & _v1, const Vec3f & _v2, Material *
      _mat)
  : v0(_v0), v1(_v1), v2(_v2), n((v1-v0)^(v2-v0)), mat(_mat){}
  Triangle(const Vec3f & _v0, const Vec3f & _v1, const Vec3f & _v2, const Vec3f &
      _n0, const Vec3f & _n1, const Vec3f & _n2, const Vec3f & _t0, const Vec3f &
      _t1, const Vec3f & _t2, Material * _mat)
  : v0(_v0), v1(_v1), v2(_v2), n((v1-v0)^(v2-v0)), n0(_n0), n1(_n1), n2(_n2),
      t0(_t0), t1(_t1), t2(_t2), mat(_mat){}
  Triangle(const Triangle & t): v0(t.v0), v1(t.v1), v2(t.v2), n(t.n), n0(t.n0),
      n1(t.n1), n2(t.n2), t0(t.t0), t1(t.t1), t2(t.t2){}

  Vec3f operator[](int i){
    return (&v0)[i];
  }

  Triangle & operator=(const Triangle & t){
    v0 = t.v0; v1 = t.v1; v2 = t.v2;
    n = t.n; n0 = t.n0; n1 = t.n1; n2 = t.n2;
    t0 = t.t0; t1 = t.t1; t2 = t.t2;
    min = t.min; max = t.max;
    return *this;
  }

  Vec3f lerpPoint(float c0, float c1, float c2) const{ // interpola i vertici
    return v0*c0 + v1*c1 + v2*c2;
  }

  Vec3f lerpNormal(float c0, float c1, float c2) const{ // interpola le normali
    return n0*c0 + n1*c1 + n2*c2;
  }

  Vec3f lerpTexture(float c0, float c1, float c2) const{ // interpola le texture
    return t0*c0 + t1*c1 + t2*c2;
  }

  void applyTransform(Transform * tr){ // metodo che applica una trasformazione al
      triangolo
    v0 = (tr->m->operator*(Vec4f(v0,1.f))).xyz();
    v1 = (tr->m->operator*(Vec4f(v1,1.f))).xyz();
    v2 = (tr->m->operator*(Vec4f(v2,1.f))).xyz();
```

```
    n0 = (tr->mTrInv->operator*(Vec4f(n0,0.f))).xyz();
    n1 = (tr->mTrInv->operator*(Vec4f(n1,0.f))).xyz();
    n2 = (tr->mTrInv->operator*(Vec4f(n2,0.f))).xyz();
    n = (v1-v0)^(v2-v0);
  }

  void setBVertex(){ // metodo che calcola il bounding box del triangolo
    min = max = v0;
    for(int i = 0; i < 3; ++i)
      for(int j = 1; j < 3; ++j){
        float val = (*this)[j][i];
        min[i] = val < min[i] ? val : min[i];
        max[i] = val > max[i] ? val : max[i];
      }
  }

  std::string toString() const{
    return "Triangle: [ " + v0.toString() + ", " + v1.toString() + ", " +
        v2.toString() + "]";
  }

  friend std::ostream & operator<<(std::ostream &, const Triangle &);

  Vec3f v0, v1, v2; // vertici
  Vec3f n, n0, n1, n2; // normali ai vertici
  Vec3f t0, t1, t2; // texture ai vertici
  Vec3f min, max; // vertici del bounding-box
  Material * mat; // materiale del triangolo

};

inline std::ostream & operator<<(std::ostream & os, const Triangle & t){
  return os << "Triangle: [ " << t.v0 << ", " << t.v1 << ", " << t.v2 << "]";
}

#endif
```

---

vect3f.hpp:

```
/*
  File di dichiarazione della classe Vec3f, modella un vettore tridimensionale di
      float
*/

#ifndef _VEC_3F_
#define _VEC_3F_

#include <iostream>
#include <math.h>
```

```
#include <assimp/vector3.h>

class Vec3f{
public:
  Vec3f(): x(), y(), z(){}
  Vec3f(float s): x(s), y(s), z(s){}
  Vec3f(float _x, float _y, float _z): x(_x), y(_y), z(_z){}
  Vec3f(float * v){x = v[0]; y = v[1]; z = v[2];}
  Vec3f(const Vec3f & v): x(v.x), y(v.y), z(v.z){}
  Vec3f(const aiVector3D & v): x(v.x), y(v.y), z(v.z){}

  Vec3f operator+(const Vec3f & v) const{return Vec3f(x+v.x, y+v.y, z+v.z);} //
     somma tra vettori
  Vec3f & operator+=(const Vec3f & v){x += v.x; y += v.y; z += v.z; return *this;}
  Vec3f operator-(const Vec3f & v) const{return Vec3f(x-v.x, y-v.y, z-v.z);} //
     differenza tra vettori
  Vec3f & operator-=(const Vec3f & v){x -= v.x; y -= v.y; z -= v.z; return *this;}
  Vec3f operator*(float s) const{return Vec3f(x*s, y*s, z*s);} // prodotto per
     scalare
  Vec3f & operator*=(float s){x *= s; y *= s; z *= s; return *this;}
  float operator*(const Vec3f & v) const{return x*v.x + y*v.y + z*v.z;} //
     prodotto scalare
  Vec3f operator^(const Vec3f & v) const{return Vec3f(y*v.z - z*v.y, z*v.x -
     x*v.z, x*v.y - y*v.x);} // prodotto vettoriale
  Vec3f operator%(const Vec3f & v) const{return Vec3f(x*v.x, y*v.y, z*v.z);} //
     prodotto componente per componente
  Vec3f operator/(float s) const{float d = 1./s; return Vec3f(x*d, y*d, z*d);} //
     divisione per scalare
  Vec3f & operator/=(float s){float d = 1./s; x *= d; y *= d; z *= d; return
     *this;}
  Vec3f operator-() const{return Vec3f(-x, -y, -z);} // moltiplicazione per -1
  bool operator==(const Vec3f & v) const{return x == v.x && y == v.y && z == v.z;}
  bool operator!=(const Vec3f & v) const{return x != v.x || y != v.y || z != v.z;}
  bool operator>(const Vec3f & v) const{return x > v.x && y > v.y && z > v.z;}
  bool operator>=(const Vec3f & v) const{return x >= v.x && y >= v.y && z >= v.z;}
  bool operator<(const Vec3f & v) const{return x < v.x && y < v.y && z < v.z;}
  bool operator<=(const Vec3f & v) const{return x <= v.x && y <= v.y && z <= v.z;}
  Vec3f & operator=(const Vec3f & v){x = v.x; y = v.y; z = v.z; return *this;}
  Vec3f & operator=(const aiVector3D & v){x = v.x; y = v.y; z = v.z; return *this;}
  float operator[](int i) const{return (&x)[i];}
  float & operator[](int i) {return (&x)[i];}

  float lengthSq() const{return x*x + y*y + z*z;} // norma quadra
  float length() const{return sqrt(lengthSq());} // norma
  Vec3f hat() const{return (*this)/length();} // ritorna il vettore normalizzato
  Vec3f & normalize(){return (*this)/=length();} // normalizza il vettore
  Vec3f reflect(const Vec3f & n) const{return *this - n*((*this)*n)*2.;} //
     riflette il vettore rispetto alla normale
  Vec3f & reflect(const Vec3f & n){return *this -= n*((*this)*n)*2.;}
```

```
  Vec3f floorVec() const{return Vec3f(floor(x),floor(y),floor(z));}
  Vec3f & floorVec(){x = floor(x); y = floor(y); z = floor(z); return *this;}
  Vec3f roundVec() const{return Vec3f(round(x),round(y),round(z));}
  Vec3f & roundVec(){x = round(x); y = round(y); z = round(z); return *this;}


  std::string toString() const{
    return "{ "+std::to_string(x)+", "+std::to_string(y)+", "+std::to_string(z)+"}";
  }

  float x, y, z;
};

inline Vec3f operator*(float s, const Vec3f & v){return v*s;}
inline std::ostream & operator<<(std::ostream & os, const Vec3f & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << "}";
}

#endif
```

---

vect3i.hpp:

```
/*
  File di dichiarazione della classe Vec3i, modella un vettore tridimensionale di
     interi
*/

#ifndef _VEC_3I_
#define _VEC_3I_

#include <iostream>
#include <math.h>

class Vec3i{
public:
  Vec3i(): x(), y(), z(){}
  Vec3i(int s): x(s), y(s), z(s){}
  Vec3i(int _x, int _y, int _z): x(_x), y(_y), z(_z){}
  Vec3i(const Vec3i & v): x(v.x), y(v.y), z(v.z){}

  Vec3i operator+(const Vec3i & v) const{return Vec3i(x+v.x, y+v.y, z+v.z);} //
     somma tra vettori
  Vec3i & operator+=(const Vec3i & v){x += v.x; y += v.y; z += v.z; return *this;}
  Vec3i operator-(const Vec3i & v) const{return Vec3i(x-v.x, y-v.y, z-v.z);} //
     differenza tra vettori
  Vec3i & operator-=(const Vec3i & v){x -= v.x; y -= v.y; z -= v.z; return *this;}
  Vec3i operator*(int s) const{return Vec3i(x*s, y*s, z*s);} // prodotto per
     scalare
  Vec3i & operator*=(int s){x *= s; y *= s; z *= s; return *this;}
```

```
  int operator*(const Vec3i & v) const{return x*v.x + y*v.y + z*v.z;} // prodotto
      scalare
  Vec3i operator^(const Vec3i & v) const{return Vec3i(y*v.z - z*v.y, z*v.x -
      x*v.z, x*v.y - y*v.x);} // prodotto vettoriale
  Vec3i operator%(const Vec3i & v) const{return Vec3i(x*v.x, y*v.y, z*v.z);} //
      prodotto componente per componente
  Vec3i operator/(int s) const{int d = 1./s; return Vec3i(x*d, y*d, z*d);} //
      divisione per scalare
  Vec3i & operator/=(int s){int d = 1./s; x *= d; y *= d; z *= d; return *this;}
  Vec3i operator-() const{return Vec3i(-x, -y, -z);} // moltiplicazione per -1
  bool operator==(const Vec3i & v) const{return x == v.x && y == v.y && z == v.z;}
  bool operator!=(const Vec3i & v) const{return x != v.x || y != v.y || z != v.z;}
  bool operator>(const Vec3i & v) const{return x > v.x && y > v.y && z > v.z;}
  bool operator>=(const Vec3i & v) const{return x >= v.x && y >= v.y && z >= v.z;}
  bool operator<(const Vec3i & v) const{return x < v.x && y < v.y && z < v.z;}
  bool operator<=(const Vec3i & v) const{return x <= v.x && y <= v.y && z <= v.z;}
  Vec3i & operator=(const Vec3i & v){x = v.x; y = v.y; z = v.z; return *this;}
  int operator[](int i) const{return (&x)[i];}
  int & operator[](int i) {return (&x)[i];}

  int lengthSq() const{return x*x + y*y + z*z;} // norma quadra
  int length() const{return sqrt(lengthSq());} // norma
  Vec3i hat() const{return (*this)/length();} // ritorna il vettore normalizzato
  Vec3i & normalize(){return (*this)/=length();} // normalizza il vettore
  Vec3i floorVec() const{return Vec3i(floor(x),floor(y),floor(z));}
  Vec3i & floorVec(){x = floor(x); y = floor(y); z = floor(z); return *this;}
  Vec3i roundVec() const{return Vec3i(round(x),round(y),round(z));}
  Vec3i & roundVec(){x = round(x); y = round(y); z = round(z); return *this;}


  std::string toString() const{
    return "{ "+std::to_string(x)+", "+std::to_string(y)+", "+std::to_string(z)+"}";
  }

  int x, y, z;
};

inline Vec3i operator*(int s, const Vec3i & v){return v*s;}
inline std::ostream & operator<<(std::ostream & os, const Vec3i & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << "}";
}

#endif
```

---

vect4f.hpp:

```
/*
  File di dichiarazione della classe Vec4f, modella un vettore quadridimensionale
      di float
```

```
*/

#ifndef _VEC_4F_
#define _VEC_4F_

#include "vec3f.hpp"
#include <assimp/vector3.h>

class Vec4f{
public:
  Vec4f(): x(), y(), z(), w(){}
  Vec4f(float s, float _w): x(s), y(s), z(s), w(_w){}
  Vec4f(float _x, float _y, float _z, float _w): x(_x), y(_y), z(_z), w(_w){}
  Vec4f(float * v){x = v[0]; y = v[1]; z = v[2]; w = v[3];}
  Vec4f(const Vec3f & v, float _w): x(v.x), y(v.y), z(v.z), w(_w){}
  Vec4f(const aiVector3D & v, float _w): x(v.x), y(v.y), z(v.z), w(_w){}
  Vec4f(const Vec4f & v): x(v.x), y(v.y), z(v.z), w(v.w){}

  float operator[](int i) const{return (&x)[i];}
  float & operator[](int i) {return (&x)[i];}

  Vec4f operator/(float s){float d = 1./s; return Vec4f(x*d, y*d, z*d, w*d);} //
     divisione per scalare
  Vec4f & operator/=(float s){float d = 1./s; x *= d; y *= d; z *= d; w *= d;
     return *this;}

  Vec4f & perspDivide(){float d = 1./w; x *= d; y *= d; z *= d; w = 1.; return
     *this;} // applica la divisione prospettica
  Vec3f xyz() const{return Vec3f(x,y,z);}

  float x, y, z, w;
};

inline std::ostream & operator<<(std::ostream & os, const Vec4f & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << ", " << v.w << "}";
}

#endif
```

CAPITOLO 6

Radiosità

6.1. Introduzione alla radiosità

Questo modello per la luce diffusa, introdotto in [**19**], sposta l'attenzione dallo studio delle riflessioni dirette ad una analisi accurata dell'effetto cumulativo delle interriflessioni tra gli oggetti della scena. I primi algoritmi in questo campo [**42**] si occupavano della radiazione del calore dentro motori ed altri ambienti chiusi, e da questo ambito furono poi trasportati allo studio della diffusione della luce. Si calcola quanta parte dell'energia emessa o riflessa da ogni superficie raggiunge ogni altra superficie, venendone ulteriormente riflessa o assorbita. Una parte della porzione riflessa raggiunge nuovamente la superficie originale, e quindi ne modifica la luminosità. Questo porta ad una successione di correzioni successive, che si traducono nella risoluzione con metodi iterativi di un appropriato sistema lineare. La soluzione dà la luminosità di ogni superficie della scena in conseguenza dell'illuminazione che la raggiunge direttamente dalle sorgenti di luce ed indirettamente dalla diffusione da parte delle altre superficie. La potenza luminosa totale di una superficie (emessa e/o riflessa) si chiama radiosità. Il metodo di calcolo suddivide le superficie in pezzi più piccoli di luminosità uniforme, chiamati elementi (*patch*), e risolve per iterazione il sistema lineare di cui sopra per trovare la luminosità dei vertici di ciascun elemento generata dalla luce diretta e da quella diffusa dalle altre superficie. Il procedimento iterativo aggiunge a ciascun elemento la luminosità diffusa dalle successive. Quindi nei primi passi iterativi la luminosità globale risulta ridotta, perché si perde la luminosità dovuta ai contributi degli elementi ancora non considerate. Per rendere più rapida la convergenza dell'iterazione, viene quindi eseguito un procedimento di correzione della luminosità globale per recuperare la perdita iniziale.

6.2. L'equazione della radiosità

Le superficie considerate nel metodo di radiosità possono emettere luce propria, a differenza di quelle considerate nel ray tracing. In particolare, le sorgenti di luce in questo metodo diventano superficie (il metodo è ideale, in particolare, per trattare sorgenti di luce estese), ma ciascuna superficie, oltre ad emettere luce propria, diffonde quella che le proviene dalle altre.

Quindi ciascuna superficie della scena è modellata come una sorgente luminosa estesa, avente un'area finita. Suddividiamo l'ambiente in un numero finito di sottosuperficie sufficientemente piccole, che chiameremo elementi, che emettono e riflettono luce in modo uniforme sull'intera area. Se consideriamo ogni elemento come una sorgente di luce che, oltre ad emettere la propria luce, diffonde quella proveniente dalle altre sorgenti (ma non la riflette specularmente: è una superficie opaca), allora, per ogni $i = 1,\ldots,$ n, abbiamo la seguente equazione:

$$A_i b_i = A_i e_i + \rho_i \sum_{j=1,\ldots,n} A_j b_j F_{ji} \; .$$

Qui b_i e b_j sono le radiosità - cioè le potenze totali per unità di area (ossia i flussi di energia emessa per unità di tempo e di area) - uscenti dagli elementi i e j, A_i ed A_j sono le aree degli elementi i e j, e_i la potenza totale per unità di area emessa dall'elemento i, ρ_i la riflettività dell'elemento i e F_{ji} il fattore di forma dell'elemento i rispetto all'elemento j, che misura quale percentuale della potenza emessa dall'elemento j arriva all'elemento i, proiettata nella direzione perpendicolare alla superficie di i (è

necessario proiettare sulla direzione normale per tener conto del fattore di attenuazione angolare rispetto all'angolo di incidenza, illustrato in (2.2.1)). Il fattore di forma misura quindi quanto è grande l'angolo solido coperto dalla porzione visibile dell'elemento i quando si guarda la scena dal centro dell'elemento j: se invece dell'elemento i si prende un suo punto di osservazione, ovvero una piccola area intorno ad un suo punto, diciamo un'area quasi piana, allora per il suo contributo al fattore di forma occorre proiettare perpendicolarmente a j, perché stiamo calcolando la potenza *per unità di area* che l'elemento i riceve da quello j. In effetti, per il calcolo è necessario fare la media degli angoli solidi coperti dall'elemento i al variare del punto di osservazione sull'elemento j, ma questo lo faremo in seguito, per ora il calcolo non ci serve. Il $j-$simo addendo della sommatoria rappresenta l'energia per unità di tempo e di area che l'elemento i riceve da quello j. Questa energia, moltiplicata per il coefficiente di riflettività dell'elemento i, è la frazione proveniente dall'elemento j della energia per unità di tempo e di area emessa dall'elemento i. Riassumendo, questa equazione è

$$b_i = e_i + \rho_i \sum_{j=1,\dots,n} b_j F_{ji} \frac{A_j}{A_i} \,,$$

ed indica semplicemente la relazione esistente tra l'energia emessa per unità di tempo dall'elemento i e l'energia per unita di tempo entrante in questo elemento, vista come somma della componente di luce propria e quella di luce diffusa proveniente dagli altri elementi. È importante notare che il 100% dell'energia emessa da ciascun elemento deve arrivare agli altri elementi (considerando anche lo sfondo come uno o più elementi). In altre parole si ha la seguente equazione della conservazione dell'energia (ovvero dell'angolo solido totale):

$$\sum_i F_{ji} = 1 \,. \tag{6.2.1}$$

Come accennato più sopra, quest'ultima quantità viene determinata moltiplicando per il coefficiente di riflessione dell'elemento i la somma della luce incidente proveniente dagli altri elementi, cioè la somma al variare di j della luce che lascia un'unità di area dell'elemento j e che raggiunge l'elemento i. Come vedremo, vale la seguente relazione di reciprocità tra i fattori di forma (reversibilità dei percorsi ottici):

$$A_i F_{ij} = A_j F_{ji} \,. \tag{6.2.2}$$

(Per una comprensione di questa condizione di normalizzazione in termini di angoli solidi, si veda la successiva Nota 6.3.1).

Essenzialmente questa relazione esprime, in termini della luce inviata da un elemento all'altra, la reversibilità dei percorsi ottici: se un punto di un elemento è visibile da un punto dell'altro, allora è vero anche il viceversa. Pertanto possiamo riscrivere l'equazione precedente così:

$$b_i = e_i + \rho_i \sum_{j=1}^{n} b_j F_{ij} \tag{6.2.3}$$

Riordinando i termini e considerando tutti i possibili valori dell'indice i arriviamo al seguente sistema lineare:

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \vdots & 1-\rho_n F_{nn} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} \tag{6.2.4}$$

Il sistema è quindi il seguente:

$$M\boldsymbol{b} = \boldsymbol{e}, \tag{6.2.5}$$

dove $\boldsymbol{b} = (b_1, \ldots, b_n)$ ed $\boldsymbol{e} = (e_1, \ldots, e_n)$ sono rispettivamente i vettori delle radiosità e delle potenze emesse, ed M è la matrice

$$M_{ij} = \delta_{ij} - \rho_i F_{ij}$$

(come sempre, qui δ_{ij} denota il simbolo di Kronecker, che vale 1 se $i = j$ e 0 altrimenti). È importante rendersi conto per i termini diagonali possiamo supporre

$$M_{ii} > 0$$

dal momento che

$$\rho_i < 1$$

(la radiosità serve a studiare scene diffusive, non perfettamente speculari!). Ma anche se ci fosse qualche elemento i perfettamente speculare, ossia con $\rho_i = 1$, dobbiamo comunque richiedere

$$F_{ii} < 1$$

a meno che l'elemento $i-$simo non sia l'unico elemento della scena incluso lo sfondo, nel qual caso la scena è banale: l'unico elemento ha illuminazione costante, e non occorrono metodi numerici per il rendering. Pertanto, per ogni i, si ha

$$M_{ii} = 1 - \rho_i F_{ii} > 0. \tag{6.2.6}$$

Questo sistema di equazioni, risolvibile usando il metodo iterativo di Gauss–Seidel, porta ad ottenere i valori di radiosità per ogni elemento, i quali, calcolati in funzione della lunghezza d'onda, possono essere interpretati come la distribuzione spettrale dell'intensità di illuminazione di quell'elemento nell'immagine, e sono utilizzabili nell'algoritmo di determinazione delle superficie visibili.

6.2.1. Trasformazione dei valori di radiosità degli elementi di superficie a valori numerici sulla mesh. Per poter visualizzare l'illuminazione della scena occorre alla fine trasportare i valori della radiosità dei singoli elementi a valori associati ai vertici della mesh, ossia della griglia di modellazione tridimensionale.

Nell'articolo **[10]** viene sviluppato un procedimento per determinare la radiosità di un vertice della scena poligonale, utilizzabile poi nello shading interpolato (secondo il metodo di Gouraud: qui non si usa il metodo di Phong perché stiamo studiando un problema di luce solo diffusa, e quindi la direzione dell'osservatore è irrilevante e non ci serve calcolare il versore $\boldsymbol{V}$ ad ogni punto: ma il metodo di interpolazione di Phong invece lo richiede). Il procedimento è il seguente: ad un vertice interno alla superficie viene assegnata la media delle radiosità degli elementi $j_1, j_2, \ldots, j_k$ che condividono quel vertice, pesata con le rispettive aree, ossia $\sum_{m=1}^{k} A_{j_m} b_{j_m} / \sum_{m=1}^{k} A_{j_m}$. Invece, se il vertice appartiene ad un lato, la radiosità di quel vertice viene determinate in due passi. Prima si trova quella del vertice interno più vicino al vertice dato e quelle degli elementi che condividono il vertice dato. Poi la media di questi elementi (pesata con le rispettive aree) viene posta uguale alla media tra il vertice interno più vicino e la radiosità del vertice dato, che è la nostra incognita. Così si arriva ad una equazione lineare, risolvendo la quale otteniamo la quantità cercata.

Rammentiamo che il fatto di trasferire le radiosità ai singoli vertici è particolarmente vantaggioso perché ci permette di inerpolare secondo l'algoritmo di shading di Gouraud, che si basa sull'interpolazione a partire dai dati sui vertici (qui non si usa l'algoritmo di Phong perché la scena è puramente diffusiva, ovvero con coefficienti di riflessione pari a zero).

6.3. Calcolo dei fattori di forma

L'operazione più importante e computazionalmente più gravosa del metodo di radiosità è il calcolo dei fattori di forma, i quali misurano quale frazione dell'energia emessa da un elemento ne raggiunge un'altra. Sono quindi una misura geometrica della visibilità di una superficie da un'altra. In **[10]** viene sviluppato un algoritmo a precisione di immagine per la determinazione delle superficie visibili per approssimare i fattori di forma. Dati due elementi, la geometria che porta a determinare

il fattore di forma da un elemento infinitesimo di area dA_i della prima ad un elemento infinitesimo dA_j della seconda (che chiamiamo $F_{di,j}$) è illustrata nella figura seguente. Occorre moltiplicare le aree per i due rispettivi coseni per proiettarle perpendicolarmente al segmento che le congiunge (in generale sono disposte obliquamente rispetto ad esso, ed in tal modo coprono un angolo solido ridotto di un fattore pari a questo coseno).

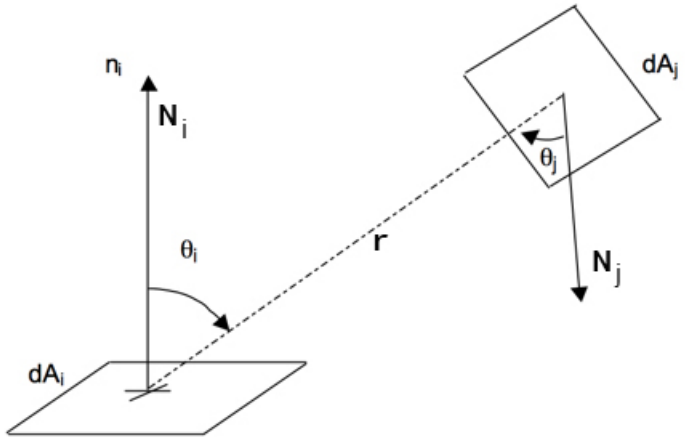


FIGURA 6.3.1. Fattore di forma fra elementi infinitesimi

Qui θ_i è l'angolo che il raggio forma con la normale alla superficie A_i, θ_j è l'angolo che il raggio forma con la normale alla superficie A_j, V_{ij} assume i valori 0 o 1 a seconda che dA_j sia visibile o meno da dA_i, e r è la lunghezza del raggio. Partendo da questo risultato possiamo ottenere il fattore di forma da un elemento infinitesimo di area dA_i all'intero elemento $j-$esimo di area A_j integrando su tutta l'area dell'elemento j-esimo. In tal modo otteniamo che $F_{di,j}$ è dato da

$$F_{di,j} = \int_{A_j} \frac{\cos\theta_i \, \cos\theta_j}{\pi r^2} \, V_{ij} \, dA_j \, . \tag{6.3.1}$$

Nell'integrando, il fattore $\cos\theta_i$ è dovuto al fatto che stiamo calcolando l'area proiettata perpendicolarmente a dA_i. Per verificare che il fattore di normalizzazione dell'integrale sia π consideriamo il caso in cui l'elemento osservato A_j sia un emisfero, guardato dal centro della sua base, senza altri elementi ad occludere la visuale. Allora ogni punto dell'emisfero viene osservato dalla direzione radiale, e quindi $\theta_j = 0$. Ma $\int_{A_j} dA_j$ è l'area dell'emisfero, che vale $2\pi r^2$. Perciò

$$\int_{A_i}\int_{A_j} \frac{\cos\theta_i \, \cos\theta_j}{r^2} \, dA_j \, dA_i = \int_{A_j} \frac{\cos\theta_i}{r^2} \, dA_j = \int_0^{2\pi} d\phi \int_0^{\frac{\pi}{2}} \frac{\cos\theta}{r^2} \, r^2 \sin\theta \, d\theta$$

$$= 2\pi \int_0^{\frac{\pi}{2}} \cos\theta \, \sin\theta \, d\theta = 2\pi \int_0^1 u \, du = \pi$$

(dovremo ripetere questo calcolo nella determinazione della radianza di un emettitore diffusivo, nella Sottosezione 7.4.1).
Ora, integrando sull'elemento $i-$esimo e prendendo la media integrale (cioè dividendo l'integrale per l'area A_i), otteniamo il fattore di forma dall'elemento i all'elemento j:

$$F_{i,j} = \frac{1}{A_i} \int_{A_i}\int_{A_j} \frac{\cos\theta_i \, \cos\theta_j}{\pi r^2} \, V_{ij} \, dA_j \, dA_i \, . \tag{6.3.2}$$

Si osservi che da questa formula segue la relazione di reciprocità (6.2.2). Il fatto che la normalizzazione si ottenga dividendo non per l'area 2π della semisfera unitaria, bensì per l'area π del suo disco di base sarà chiarito fra poco nella Nota 6.3.1.
Se assumiamo che il punto centrale di un elemento rappresenti con buona precisione la luminosità di tutti gli altri suoi punti, possiamo approssimare F_{ij} con $F_{di,j}$, calcolato per dA_i nel centro dall'elemento $i-$esimo.

6.3.1. Calcolo dei fattori di forma con $z-$buffer emisferico. Sono disponibili metodi numerici più semplici per calcolare $F_{di,j}$. Nel libro [**42**]) si attribuisce a Nusselt l'aver notato che la quantità F_{ij} si possa calcolare mediante due proiezioni successive: innanzitutto bisogna proiettare quelle parti di A_k (elemento k-esimo) che sono visibili dal pezzo dA_i dell'elemento A_i (elemento $i-$esimo) su una semisfera unitaria, centrata in dA_i: si osservi che il fattore di forma di questa proiezione (ossia l'angolo solido da essa coperto) coincide con quello dell'elemento A_k. Successivamente quest'area sulla superficie della semisfera viene proiettata ortograficamente sul disco unitario che costituisce la base della semisfera. La percentuale dell'area di questo disco coperta dalla superficie proiettata è il fattore di forma (la proiezione sulla sfera unitaria è data dalla moltiplicazione per $\cos\theta_j/r^2$, la proiezione sul disco unitario dalla moltiplicazione per $\cos\theta_i$.

NOTA 6.3.1. *Poiché l'area del disco di base della semisfera unitaria vale π, questo conferma che la normalizzazione corretta del fattore di forma differenziale si ha dividendo l'integrale per π anziché per l'area 2π della semisfera.*

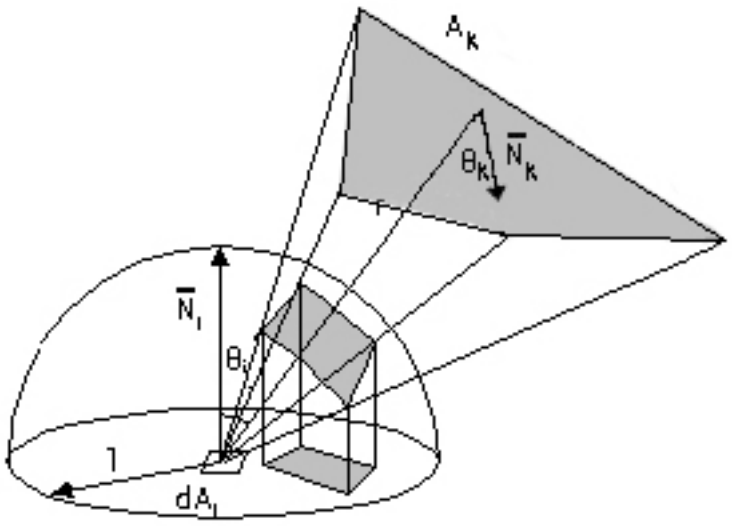


FIGURA 6.3.2. Metodo di Nusselt per il calcolo del fattore di forma infinitesimo

COROLLARIO 6.3.2. *Il fattore di forma infinitesimo di un elemento A visto da un punto $\boldsymbol{p}$ rimane invariante se l'elemento A viene proiettato radialmente (con centro di proiezione nel punto $\boldsymbol{p}$) su*

una superficie sferica centrata in $\boldsymbol{p}$*, od anche se viene proiettato radialmente su una qualunque altra superficie intermedia fra* $\boldsymbol{p}$ *ed* A*, non necessariamente a distanza costante da* $\boldsymbol{p}$*. In altre parole, il fattore di forma infinitesimo di un elemento* A *da un punto* $\boldsymbol{p}$ *rimane invariato se ogni punto* $\boldsymbol{q}$ *di* A *viene spostato in un punto* $\boldsymbol{q'} \neq \boldsymbol{p}$ *allineato con* $\boldsymbol{p}$ *e* $\boldsymbol{q}$*.*
Nota: questo conferma il fatto che il fattore di forma differenziale è la misura dell'angolo solido sotteso, proiettata nella direzione perpendicolare al punto di osservazione.

DIMOSTRAZIONE. L'interpretazione del calcolo della formula (6.3.1) alla luce del metodo di Nusselt rivela (ed anzi equivale a) il fatto che il fattore di forma infinitesimo rimane invariante sotto proiezione radiale su una superficie emisferica centrata in $\boldsymbol{p}$. Nel caso la superficie non fosse emisferica, la si può scomporre come unione di piccole celle, le quali possono approssimarsi con settori di calotte sferiche centrate in $\boldsymbol{p}$: tenendo conto dei coseni che appaiono nella formula (6.3.1) (e del fattore di occlusione, nel caso che la superficie si ripieghi su sé stessa, e quindi si faccia ombra nel corso della proiezione radiale), vediamo che anche in questo caso il fattore di forma differenziale non cambia. □

NOTA 6.3.3. *In particolare, il precedente Corollario 6.3.2 implica che ogni fattore di forma infinitesimo (con centro di vista in qualche punto* $\boldsymbol{p}$*) è invariante sotto dilatazioni radiali* R*, qualunque sia il centro di dilatazione (naturalmente se anche il centro di vista viene dilatato in un nuovo centro* $R\boldsymbol{p}$*). Un modo equivalente ma geometricamente più elegantedi provare questo risultato è il seguente: poiché una traslazione della scena non cambia i fattori di forma, ed ogni dilatazione* $R_{\boldsymbol{p}}$ *con centro* $\boldsymbol{p}$ *differisce da una dilatazione* $R_{\boldsymbol{q}}$ *di pari raggio con centro* $\boldsymbol{q}$ *solo a meno di una traslazione* T *da* $\boldsymbol{p}$ *a* $\boldsymbol{q}$ *(nel senso che* $R_{\boldsymbol{q}} = T^{-1}R_{\boldsymbol{p}}T$*, automaticamente risultano invarianti sotto dilatazione con centro* $\boldsymbol{p}$ *anche i fattori di forma infinitesimi rispetto ad ogni altro punto* $\boldsymbol{q}$*, e quindi anche i fattori di forma non infinitesimi, che sono medie di quelli infinitesimi. Come abbiamo appena visto, più in generale, per lo stesso motivo, rimangono invariati i fattori di forma di elementi ottenuti dilatando radialmente rispetto al punto di osservazione anche se il fattore di dilatazione dipende dalla direzione di osservazione.*

ESEMPIO 6.3.4. Dal Corollario 6.3.2, ed anche dalla Nota 6.3.3, segue ad esempio che i fattori di forma infinitesimi da un cono retto C alla sua base (un disco che chiamiamo B) rimangono invariati se il disco si estrude verso l'esterno, ossia viene sostituito da una calotta sferica S_{out} dall'altra parte di C ed attaccata a C sul bordo di B, e lo stesso vale per i fattori di forma globali: $F_{CB} = F_{CS}$. In questo esempio, questa proprietà segue immediatamente dalla identità di conservazione dell'energia (6.2.1). Infatti, nelle due scene $C \cup B$ e $C \cup S$, il fattore di forma F_{CC} è lo stesso, perché in nessuna delle due c'è alcuna occlusione, quindi $F_{CC} + F_{CB} = 1 = F_{CC} + F_{CS_{\text{out}}}$, da cui $F_{CB} = F_{CS_{\text{out}}}$. La stessa proprietà non vale se invece sostituiamo B con una sfera $F_{S_{\text{out}}}$ intrusa dentro C, perché in questo caso il fattore di occlusione è non nullo per punti di vista in C vicini alla base: quindi F_{CC} diminuisce. Si noti però che il fattore di forma dall'apice del cono rimane invariato, grazie al Corollario 6.3.2. □

6.3.2. Calcolo dei fattori di forma con $z-$buffer emicubico. In **[10]** viene sviluppato un algoritmo a precisione di immagine (infatti basato su una decomposizione in una griglia di pixel, come lo $z-$buffer) nel quale la proiezione avviene sulla parte superiore di un semicubo, centrato in dA_i. Ogni faccia viene suddivisa con una griglia regolare. Tutte gli altri elementi vengono proiettate verso il centro del cubo e quindi si calcola quali celle la proiezione copre sulle cinque facce superiori. Esattamente come nel caso dell'emisfero, il fattore di forma dell'elemento originale A_k coincide con quello della sua proiezione sull'emicubo, e quindi tanto vale rimpiazzare la prima con la seconda (per la quale i vettori normali sono i vettori diretti perpendicolarmente alla faccia dell'emicubo su cui

cade la proiezione). Per ciascuna cella di queste facce si determina quale sia l'elemento A_k più vicina (nella direzione di visuale dal centro del cubo) mediante l'algoritmo di $z-$buffer. L'elemento piu vicina, per una data cella, è l'unico elemento che riflette energia luminosa su dA_i attraverso quella cella, poiché le altre sono schermate dalla prima. Ad ogni cella p viene assegnato un fattore di forma

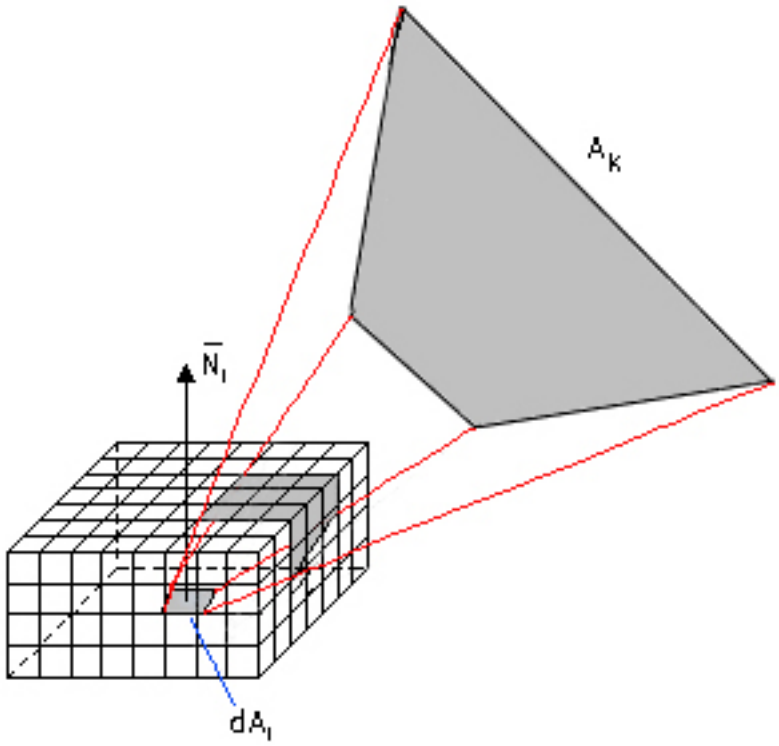


FIGURA 6.3.3. Approssimazione numerica del fattore di forma infinitesimo tramite emicubi

precalcolato del tipo

$$\Delta F_p = \frac{\cos\theta_i \, \cos\theta_p}{\pi r^2} \, \Delta A$$

dove θ_p è l'angolo tra la normale alla cella p e il vettore r che congiunge il centro di dA_i (cioè il centro del cubo) col centro di p, e ΔA è l'area della cella. In un sistema di riferimento per il semicubo con l'origine nel suo centro, per le celle della faccia superiore otteniamo

$$r = \sqrt{1 + x_p^2 + y_p^2}\,,$$

$$\cos\theta_i = \cos\theta_p = \frac{1}{r}\,.$$

dove (x_p, y_p) sono le coordinate del centro della cella p. La geometria è illustrata nella Figura 6.3.4. Ora l'equazione precedente, per una cella appartenente alla faccia superiore, ci dà:

$$\Delta F_p = \frac{1}{\pi(1 + x_p^2 + y_p^2)^2} \, \Delta A\,.$$

Invece, la geometria delle celle in una faccia laterale del semicubo che è perpendicolare, diciamo, all'asse x è illustrata nella Figura 6.3.5. Prendiamo prima in esame il caso di una tale cella allineata

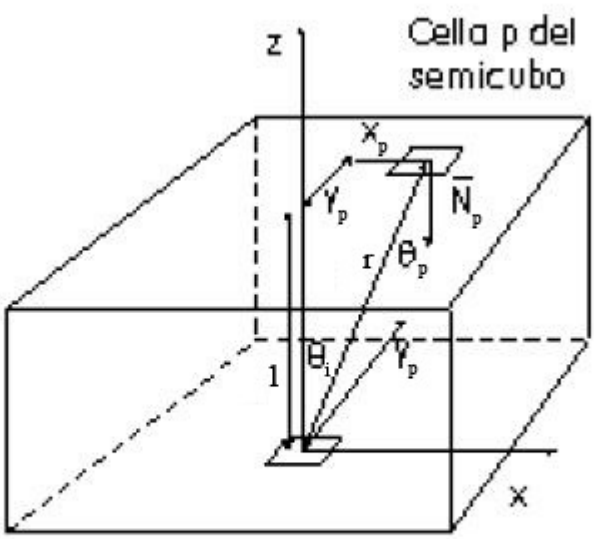


FIGURA 6.3.4. Approssimazione numerica del fattore di forma infinitesimo tramite emicubi: contributo di una cella sulla faccia alta

con il punto $\boldsymbol{p}$, cioè che insiste sul centro della base della faccia laterale a posizione, diciamo, $x_p = -1$, in altre parole con $y_p = 0$. Analogamente, per queste celle si ha $\cos\theta_p = 1/r$, per lo stesso argomento di prima, perché l'angolo θ_p è sotteso dal vettore $\boldsymbol{v}_p$ dall'origine al centro della cella p (di lunghezza r) e dal versore $\boldsymbol{m}_p$ applicato all'origine e perpendicolare alla faccia che contiene la cella p. Quindi $\cos\theta_p$ misura di quanto si accorcia il vettore $\boldsymbol{v}_p$ quando lo proiettiamo sul piano di base: siccome dopo la proiezione la sua lunghezza diventa quella della metà del lato, ossia 1, mentre prima era r, tale accorciamento, e quindi il coseno, vale $1/r$. È ovvio che per questa cella che insiste sul centro della base della faccia laterale i vettori applicati $\boldsymbol{n}_p$, $\boldsymbol{v}_p$ e $\boldsymbol{m}_p$ sono complanari (perché i primi due sono paralleli ed il terzo passa per i loro punti di applicazione). Ma se spostiamo lateralmente la cella, prendendo $y_p \neq 0$, questi tre vettori restano complanari, proprio per lo stesso motivo di incidenza, ed il risultato resta uguale.
Invece $\cos\theta_i = z_p/r$. Infatti l'angolo θ_i è sotteso dal vettore $\boldsymbol{v}_p$ dall'origine al centro della cella p e dal versore $\boldsymbol{n}_z$ dell'asse z, che è verticale; ma questo angolo dipende solo dall'altezza z_p alla quale il raggio di direzione $\boldsymbol{m}_p$ interseca l'emicubo, perché è invariante per rotazione intorno a $\boldsymbol{n}_p$ (si veda di nuovo la nella Figura 6.3.5). Pertanto anche in questo caso, se spostiamo lateralmente la cella ossia prendiamo $y_p \neq 0$, il risultato non cambia.

Quindi troviamo:

$$\Delta F_p = \frac{z_p}{\pi(1 + x_p^2 + z_p^2)^2}\, \Delta A\,.$$

Qui stiamo considerando le facce laterali, ed quindi per due di esse x_p vale rispettivamente 1 o -1, e per le altre due x_p vale rispettivamente 1 o -1.
Ora il fattore di forma $F_{di,k}$ tra l'elemento $i-$esimo e l'elemento $k-$simo può essere approssimato con la somma di tutti i fattori di forma ΔF_p associati ad ogni cella p del semicubo centrato sull'elemento

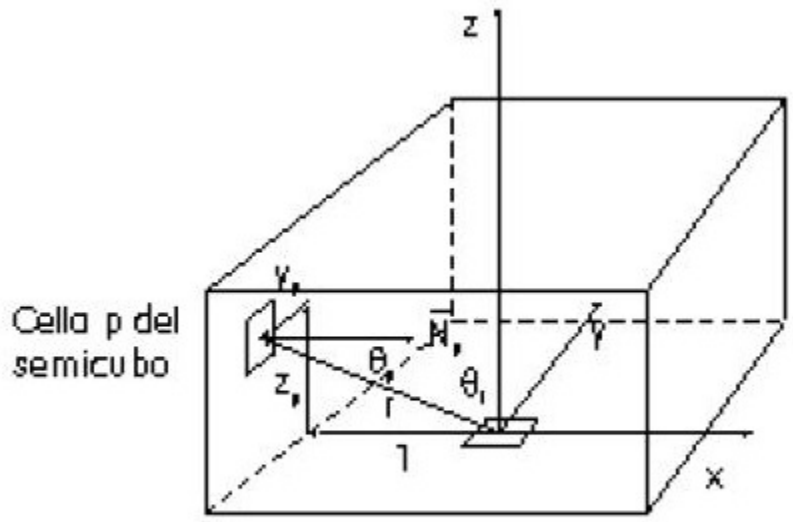


FIGURA 6.3.5. Approssimazione numerica del fattore di forma infinitesimo tramite emicubi: contributo di una cella su una faccia laterale

$i-$esimo coperta dalle proiezioni dell'elemento $k-$simo. Si noti che, poiché questi metodi usano la suddivisione in celle delle facce e lo $z-$buffer, essi sono algoritmi a precisione di immagine, e quindi soggetti al fenomeno dell'aliasing.

Questi fattori di forma precalcolati delle celle dell'emicubo ci forniscono un procedimento rapido per ottenere il fattore di forma infinitesimo dal centro dell'emicubo al generico elemento j (ottenendoli tutti insieme al variare di j con un unico z-buffer emicubo). Si noti che, se il punto di osservazione, ossia l'elemento differenziale di area dA_i, non è posto al centro del pavimento dell'emicubo, diciamo $(0,0)$, bensì al punto (x_0, y_0), allora le due formule appena trovate cambiano in

$$\Delta F_p = \frac{1}{\pi(1+(x_p-x_0)^2+(y_p-y_0)^2)^2}\,\Delta A\,,$$

$$\Delta F_p = \frac{(x_p-x_0)z_p}{\pi(z_p^2+(x_p-x_0)^2+(y_p-y_0)^2)^2}\,\Delta A\,.$$

(Ricordiamo di nuovo che qui stiamo considerando le facce laterali, ed quindi per due di esse x_p vale rispettivamente 1 o -1, e per le altre due y_p vale rispettivamente 1 o -1. Si veda la Figura 6.3.6.)

Questi sono quindi i fattori di forma precalcolati delle celle dell'emicubo da utilizzare quando vogliamo ottenere il fattore di forma globale, non infinitesimo, visto da un elemento i che giace nella base dell'emicubo (o analoghi calcoli, ma con proiezione radiale verso il centro dell'emisfero, ci permettono di ottenere questo stesso risultato dal metodo di Nusselt). Però il risultato non dà una eccellente approssimazione se l'elemento i è grande rispetto alla base dell'emicubo o emisfero invece che essere tutto vicino al suo centro. Nel caso dell'emicubo questo accade perché il metodo proietta radialmente verso il centro dell'emisfero, e quindi la prioiezione non è quella giusta verso piccoli elementi lontani dal centro. Nel caso dell'emicubo, l'errore nasce dal fatto che abbiamo

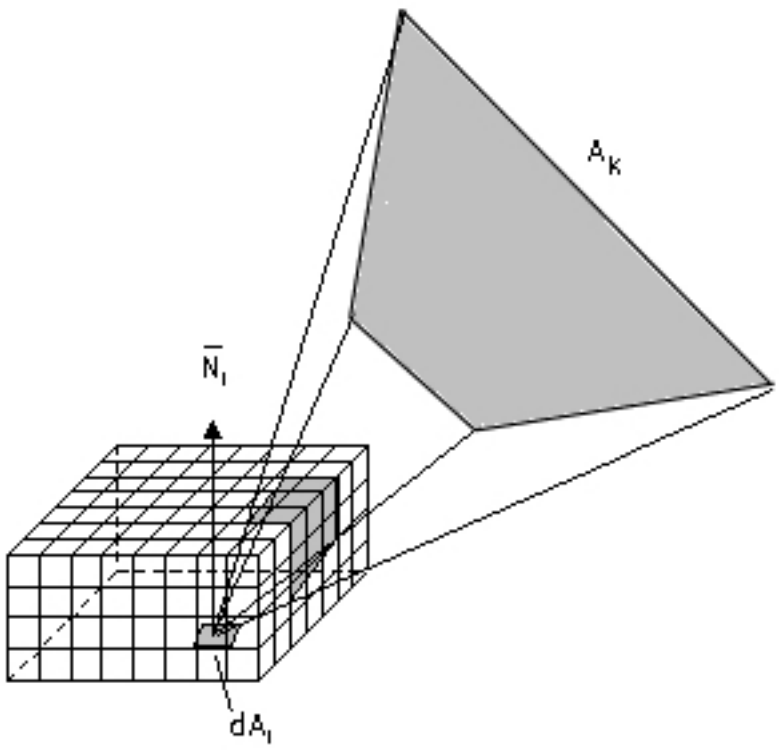


FIGURA 6.3.6. Approssimazione numerica del fattore di forma infinitesimo di un sottoelemento tramite un emicubo non centrato in esso

approssimato i fattori di forma precalcolati discretizzando il problema e calcolando i coseni degli angoli individuati, rispetto alle normali, da segmenti che cominciano e terminano ai centri delle celle. ma se un piccolo elemento i occupa una cella della base che tocca una cella di una parete laterale, allora questa approssimazione diventa scadente per segmenti che cominciano e terminano a coppie di punti, uno sulla base e l'altro sulla parete, arbitrariamente vicini fra loro. In effetti, l'andamento con $1/r_{xy}^2$ dà una divergenza che con questa discretizzazione viene omessa. Occorre invece calcolare il fattore di forma precalcolato verso tali celle laterali scrivendo i coseni e la distanza per ogni coppa di punti nella prima e nella seconda cella e poi integrando. Questo calcolo, che verrà svolto nella Sezione 6.5, è laborioso: ma visto che verrà interamente svolto, lasciamo al lettore il compito di ricavarne il fattore di forma precalcolato giusto.

In alternativa, al fine di calcolare il fattore di forma (visto da un elemento i) globale, invece che infinitesimo, sia con il metodo dell'emicubo sia con il metodo di Nusselt emisferico, è meglio rifare il calcolo spezzando l'elemento i in tanti piccoli sottoelementi dA_i e ricentrando l'emicubo intorno a ciascuno di essi. Di solito si assume che l'emicubo sia abbastanza grande che tutto l'elemento i sia vicino al suo centro, e quindi il fattore di forma F_{ij} coincida con buona approssimazione con il fattore di forma infinitesimo dal suo centro. Se così non è, allora il fattore di forma a partire dall'elemento i che calcoliamo con questi metodi ha una approssimazione via via peggiore quando l'elemento i, invece di essere tutto vicino al centro dell'emisfero o emicubo, si avvicina in alcuni punti alla sua circonferenza di bordo. Questo metodo ci fornisce quindi il fattore di forma in modo non solo soggetto ad aliasing, ma anche ad approssimazione dettata dalla geometria e grandezza degli elementi. Torneremo nella Sezione 6.6 al problema di ottenere una migliore approssimazione in termini di sottoelementi. Un modo di procedere più accurato è quello di accettare piccoli sottoelementi dA_i

non posizionati al centro dell'emicubo, come in Figura 6.3.6, ma utilizzare per ciascuno di essi gli opportuni fattori di forma precalcolati delle celle dell'emicubo (che ora non è più centrato nel piccolo elemento dA_i, quindi il calcolo fatto prima non è più valido). Ma se un elemento si spezza in tanti piccoli sottoelementi, qusto aumenta corrispondentemente la mole di calcoli.

Si noti anche che, se si ingrandisce di un fattore α l'emicubo per fare in modo che tutto l'elemento i si trovi vicino al suo centro, si aumenta il numero delle sue celle di α^2, il che comunque appesantisce il calcolo: ma per fortuna lo z-buffer emicubico è un procedimento rapido, in quanto procede tramite calcoli incrementali.

6.4. Esercizi sul calcolo analitico dei fattori di forma

ESERCIZIO 6.4.1. Una scena consiste di un cono retto di angolo al vertice pari a $2\theta_0$ (ossia di deviazione massima rispetto all'asse di simmetria pari a θ_0), ed altezza h. Quindi la base del cono è un disco, che chiamiamo B. Scegliamo B come un elemento: quanto vale il suo fattore di forma visto dal vertice del cono, considerando il vertice del cono come il centro di un elemento infinitesimo di area disposto su un piano parallelo alla base?

Suggerimento: si usi il metodo di Nusselt.

Svolgimento. Per prima cosa supponiamo che il cono abbia apice in $(0, 0, h)$ e base data dal disco B di raggio r con centro l'origine nel piano $\{z = 0\}$. Per invarianza per dilatazione possiamo prendere $r = 1$. Proiettiamo il disco sulla superficie emisferica di raggio $\rho = \sqrt{h^2 + r^2}$ nel semispazio $\{z \leqslant 0\}$, con centro nell'apice del cono, con proiezione radiale (quindi il centro di proiezione è l'apice del cono, ossia il punto da cui si vuol calcolare il fattore di forma. Per inciso, si osservi che il fattore di forma misura solo l'angolo solido sotteso, e quindi rimarrebbe invariato se la base del cono, invece di un disco piano, fosse una calotta emisferica, ad esempio quella della sfera di raggio h centrata all'apice del cono (o anche di raggio 1, come osservato nel Corollario 6.3.2, ed esemplificato proprio per questa scena geometrica nell'Esempio 6.3.4): questa semplice osservazione non è altro che la prima fase del procedimento di Nusselt, la proiezione radiale.
Quindi possiamo considerare una calotta C di ampiezza angolare θ_0, centrata al punto $x = y = 0$, $z = h$. Il metodo di Nusselt consiste nell'osservare che il fattore di forma richiesto è esattamente la percentuale dell'area del disco di base della sfera (ossia il suo disco equatoriale), che vale πh^2, coperta dalla proiezione verticale D della calotta C sul piano di base $z = 0$. La deviazione angolare massima della calotta rispetto alla direzione verticale è θ_0. Quindi la calotta si proietta sul disco D nel piano $z = 0$ il cui raggio è il raggio della sfera moltiplicato per $\sin\theta_0$, ossia $h \sin\theta_0$. L'area di D è $\pi h^2 \sin^2\theta_0$, e quindi il fattore di forma vale $\pi h^2 \sin^2\theta_0 / \pi h^2 = \sin^2\theta_0$.

ESERCIZIO 6.4.2. Per il cono dell'esercizio precedente, ricalcolare il fattore di forma infinitesimo della sua base B visto dal vertice del cono tramite la formula integrale che lo definisce.

Svolgimento. La formula che dà il fattore di forma è (6.3.1), dove gli angoli θ_i e θ_j sono come nella Figura 6.3.1, ed il fattore di occlusione V_{ij} vale 1 se il punto j è visibile dal vertice da cui vogliamo calcolare il fattore di forma (nella figura questo vertice lo chiamiamo i). Osserviamo che il fattore di occlusione vale sempre 1 perché non ci sono altri elementi che occludono la base del cono quando vista dal vertice (il cono è convesso!). Abbiamo già osservato che il fattore di forma F rimane uguale se si rimpiazza il disco di base B con la calotta sferica C di raggio h incernierata sul cono (ossia interna al cono e tangente al disco B al suo punto centrale). Inoltre sui punti di C l'angolo θ_j vale sempre 0, perché la calotta sferica ha per versore normale esattamente il versore radiale. Quindi chiamiamo più semplicemente θ l'altro angolo θ_i, e notiamo che la deviazione angolare (azimuth) θ verifica $0 \leqslant \theta \leqslant \theta_0$. Allora, fissiamo una misura σ_C su C per calcolare l'integrale di superficie, ad esempio la misura data dalle coordinate sferiche. L'integrale (il cui valore comunque è indipendente

dalla scelta della misura) diventa

$$F = \int_C \frac{\cos\theta}{\pi h^2}\, d\sigma_C = \int_0^{2\pi}\int_0^{\theta_0} \frac{\cos\theta}{\pi h^2}\, h^2 \sin\theta\, d\theta\, d\varphi = \int_0^{\theta_0} \sin 2\theta\, d\theta$$
$$= \frac{1}{2}\int_0^{2\theta_0} \sin\theta\, d\theta = \frac{1}{2}\left(1-\cos(2\theta_0)\right) = \frac{1}{2}\left(1-\cos^2\theta_0+\sin^2\theta_0\right)$$
$$= \sin^2\theta_0\ . \tag{6.4.1}$$

Qui il fattore $\sin\theta$ dentro l'integrale di superficie è il fattore di dilatazione delle aree (ossia il determinante jacobiano) della trasformazione σ da coordinate sferiche θ, φ (angoli di Eulero) a coordinate cartesiane x, y, z, calcolato in (5.2.3):

$$\left\| \frac{\partial\sigma}{\partial\theta} \times \frac{\partial\sigma}{\partial\varphi} \right\| = r\sin\theta.$$

Se invece si desidera svolgere il calcolo direttamente per il disco piano B, basta osservare che la normale al disco B è il versore verticale (diretto verso il basso, se il vertice del cono viene disposto all'origine ed il disco B ad altezza $h > 0$). Perciò ora $\theta_i = \theta_j$: chiamiamo questo angolo θ. Il raggio ρ_B di B è proporzionale al raggio ρ_E della calotta sferica E che B sottende. Attenzione: nella parte precedente di questo esercizio la calotta non era E ma C, ossia non era quella sottesa da B, bensì quella tangente a B al suo centro, che ha raggio $h < \rho_E$, ed infatti è tale che

$$\frac{h}{\rho_E} = \cos\theta_0\ . \tag{6.4.2}$$

Quindi ora il denominatore nell'integrando del fattore di forma è πr^2 invece di πh^2: questa volta r non è costante ma dipende da x e y. Nell'Esercizio 6.4.1 abbiamo visto che $\rho_B = \rho_E \sin\theta_0$. Eliminiamo ρ_E sostituendo (10.2.1a) in questa identità: troviamo la relazione, peraltro geometricamente evidente, $\rho_B = h\,\mathrm{tg}\,\theta_0$. Per la stessa ragione, un punto di B che dista ρ dal centro di B e r dall'origine ha una deviazione angolare θ rispetto alla verticale tale che $\rho = h\,\mathrm{tg}\,\theta = r\sin\theta$, e $h = r\cos\theta$ (come in (10.2.1a)). Pertanto, ponendo $u = \rho/h$, otteniamo $u = \mathrm{tg}\,\theta$, $\cos^2\theta = h^2/r^2$, e, per il teorema di Pitagora, $r^2 = \rho^2 + h^2$. Allora, se scegliamo su B la misura d'area σ_B associata alle coordinate polari nel piano $z = 0$, ne segue:

$$F = \int_B \frac{\cos^2\theta}{\pi r^2}\, d\sigma_B = \int_0^{2\pi}\int_0^{h\,\mathrm{tg}\,\theta_0} \frac{h^2}{\pi r^4}\, \rho\, d\rho\, d\varphi = 2\int_0^{h\,\mathrm{tg}\,\theta_0} \frac{h^2}{(\rho^2+h^2)^2}\, \rho\, d\rho$$
$$= 2\int_0^{\mathrm{tg}\,\theta_0} \frac{h^4}{(h^2u^2+h^2)^2}\, u\, du = 2\int_0^{\mathrm{tg}\,\theta_0} \frac{u\, du}{(1+u^2)^2} = \int_0^{\mathrm{tg}^2\,\theta_0} \frac{dv}{(1+v)^2}$$
$$= \int_1^{1+\mathrm{tg}^2\,\theta_0} \frac{dv}{v^2} = -\left.\frac{1}{v}\right|_1^{1+\mathrm{tg}^2\,\theta_0} = 1 - \frac{1}{1+\mathrm{tg}^2\,\theta_0} = \frac{\mathrm{tg}^2\,\theta_0}{1+\mathrm{tg}^2\,\theta_0}$$
$$= \cos^2\theta_0\ \mathrm{tg}^2\,\theta_0 = \sin^2\theta_0\ .$$

□

Esercizio 6.4.3. (*i*) Una scena consiste dell'intercapedine fra il piano $z = 0$ considerato come un pavimento infinito, che chiamiamo A_0, ed un soffitto infinito dato dal piano $z = 1$, che chiamiamo A_1. Siano x_0, y_0 due numeri reali. Ovviamente il fattore di forma differenziale di A_0 (ossia il fattore di forma F_{dA_1,A_0} del pavimento visto da un elemento infinitesimo di area dA_1 del soffitto) rispetto al punto $V = (x_0, y_0, 1)$ (ossia centrando dA_1 al punto $(x_0, y_0, 1)$) deve valere 1, perché il soffitto copre l'intero angolo solido nel semispazio frontale. Si

verifichi questo fatto usando la formula integrale (6.3.1) per il fattore di forma differenziale F_{dA_1,A_0}.

(*ii*) Ora la scena consiste in una stanza cilindrica, data dal cilindro retto il cui pavimento è il disco di raggio k centrato nell'origine nel piano $z = 0$, che chiamiamo $A_0(k)$, ed il soffitto, che chiamiamo $A_1(k)$, giace parallelamente ad $A_0(k)$ e direttamente sopra di esso nel piano $z = 1$. Si calcoli il fattore di forma differenziale dal centro del soffitto a $A_0(k)$, usando la formula integrale per il fattore di forma differenziale (6.3.1) per il fattore di forma differenziale $F_{dA_1(k),A_0(k)}$. Si calcoli il limite di questo risultato per k che tende ad infinito e si verifichi che coincide con il caso limite della parte precedente (intercapedine infinita).

Svolgimento. Parte (*i*): il fattore di forma differenziale è

$$F_{di,j} = \int_{A_0} \frac{\cos\theta_0 \, \cos\theta_1}{\pi r^2} \, V_{01} \, dA_j \, ,$$

dove il fattore di occlusione V_{01} vale sempre 1 perché non ci sono elementi che si frappongono, r è la lunghezza del segmento dal vertice $V = (x_0, y_0, 1)$ del cono al punto generico $(x, y, 0)$ in A_0, e gli angoli θ_0 e θ_1 sono quelli che questo segmento forma rispettivamente con il versore normale $(0, 0, 1)$ al piano $z = 0$ ed il versore normale $(0, 0, -1)$ al piano $z = 1$ applicati ai suoi rispettivi punti estremi. È quindi chiaro, dal parallelismo di questi versori, che $\cos\theta_0 = \cos\theta_1 = 1/r$, e quindi

$$F = \int_{\{z=0\}} \frac{1}{\pi r^4} \, dx \, dy = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \frac{dx \, dy}{\pi \left(1 + (x - x_0)^2 + (y - y_0)^2\right)^2} \, .$$

Per l'invarianza per traslazione, l'ultimo integrale è costante rispetto a x_0, y_0. Quindi possiamo rimpiazzare (x_0, y_0) con l'origine, e, passando a coordinate polari intorno all'origine otteniamo

$$F = \int_0^{2\pi} \int_0^{\infty} \frac{1}{\pi(1 + \rho^2)^2} \, \rho \, d\rho \, d\theta \, .$$

Ponendo $u = \rho^2$ trasformiamo l'integrale in

$$F = \int_0^{\infty} \frac{du}{(1 + u)^2} = \int_1^{\infty} \frac{du}{u^2} = 1.$$

Parte (*ii*): procedendo come sopra, ora abbiamo

$$F = \int_{A_0(k)} \frac{1}{\pi r^4} \, dx \, dy = \int_0^{2\pi} \int_0^{k} \frac{1}{\pi(1 + \rho^2)^2} \, \rho \, d\rho \, d\theta \, .$$

Ponendo $u = \rho^2$ ora abbiamo

$$F = \int_0^{k^2} \frac{du}{(1 + u)^2} = \int_1^{1+k^2} \frac{du}{u^2} = -\left.\frac{1}{u}\right|_1^{1+k^2} = 1 - \frac{1}{1 + k^2} \, .$$

Chiaramente il limite per k che tende a infinito è 1. □

Esercizio 6.4.4. (1) Una scena consiste di un emisfero S di raggio R con centro nell'origine nel semispazio $\{z \geqslant 0\}$, e del suo disco di base D. Si utilizzi questa scena per dimostrare che il fattore di forma da una ciotola emisferica a sé stessa vale $\frac{1}{2}$.

(2) Si consideri la ciotola S_θ data dalla sezione della superficie sferica di raggio R con apertura angolare θ vista dal centro della sfera: ossia, se il centro della calotta si identifica con il polo Nord della sfera, la calotta consiste di tutti i punto della sfera con deviazione azimutale dal polo Nord minore o uguale a θ. Si usino una scena ed un argomento analoghi a quelli della parte precedente per calcolare il fattore di forma dalla calotta S_θ a sé stessa.

Svolgimento. Parte (i): in base alla relazione di conservazione dell'energia, ovvero dell'angolo solido totale, (6.2.1), si ha $F_{SS} + F_{SB} = 1$, ma $F_{BS} = 1$ perché per la faccia piana B vale $F_{BB} = 0$. Pertanto, dalla relazione di reciprocità (6.2.2), si ottiene

$$F_{SB} = \frac{\operatorname{Area} B}{\operatorname{Area} S}\, F_{BS} = \frac{\operatorname{Area} B}{\operatorname{Area} S}. \tag{6.4.3}$$

D'altra parte, $\operatorname{Area} S = 2\pi R^2$ e $\operatorname{Area} B = \pi R^2$. Quindi l'identità precedente stabilisce che $F_{SB} = \frac{Area B}{\operatorname{Area} S} = \frac{1}{2}$. Perciò $F_{SS} = 1 - F_{SB} = \frac{1}{2}$. Questo prova (i).

Proviamo la parte (ii) con lo stesso ragionamento, utilizzando come nuova scena la calotta sferica S_θ chiusa dal suo disco di base B_θ. A questa scena si applica la stessa relazione di prima, (6.4.3). Calcoliamo anzitutto l'area della calotta:

$$\operatorname{Area} S_\theta = \int_0^{2\pi} d\phi \int_0^{\theta} R^2 \sin\psi \, d\psi = 2\pi R^2(1 - \cos\theta) = 4\pi R^2 \sin^2 \frac{\theta}{2}.$$

Ora osserviamo che, poiché la calotta ha raggio R ed ampiezza θ, il raggio r del suo disco di base B_θ è $r = R\sin\theta$. Quindi $\operatorname{Area} B_\theta = \pi r^2 = \pi R^2 \sin^2\theta$, e

$$\frac{\operatorname{Area} B_\theta}{\operatorname{Area} S_\theta} = \frac{1}{4}\, \frac{\sin^2\theta}{\sin^2 \frac{\theta}{2}} = \frac{1}{4}\, \frac{4\sin^2 \frac{\theta}{2}\, \cos^2 \frac{\theta}{2}}{\sin^2 \frac{\theta}{2}} = \cos^2 \frac{\theta}{2}.$$

Quindi ora segue da (6.4.3) che $F_{S_\theta B} = \cos^2 \frac{\theta}{2}$, e pertanto $F_{S_\theta S_\theta} = 1 - F_{S_\theta B} = \sin^2 \frac{\theta}{2}$. □

Esercizio 6.4.5. Si consideri una ciotola K di forma emisferica di raggio 1 appoggiata nel suo punto più basso all'origine, ossia tangente all'origine al piano $z = 0$ e contenuta nel semispazio superiore $\{z \geqslant 0\}$: quindi il bordo della ciotola è un cerchio J sul piano $z = 1$. Poiché K è convessa, il fattore di forma dalla ciotola a sé stessa è positivo.

- (i) Si mostri che l'equazione che definisce K è $x^2 + y^2 + (z-1)^2 = 1$, con $z \leqslant 1$.
- (ii) Si calcoli l'integrale del fattore di forma differenziale F_{0K} dall'origine (ossia il polo Sud) a K e si mostri che vale $\frac{1}{2}$ (suggerimento: si immagini di completare la scena in modo che sia uno spazio chiuso, ad esempio una sfera intera: se seguiamo questo suggerimento stiamo aggiungendo una semisfera con bordo J, che chiamiamo C).
- (iii) Sia $v = (x_v, y_v, z_v)$ un altro punto di K. Si scriva esplicitamente l'integrale del fattore di forma differenziale da questo punto a K, in modo che l'integrando sia espresso come funzione solo delle coordinate x_v, y_v, z_v di v e x_p, y_p, z_p di un punto p che varia in K, e si dimostri che $F_{vK} = 1/2$ per ogni v.
- (iv) Sia p un punto del bordo J della ciotola. Utilizzando opportuni ragionamenti basati su simmetrie, si dimostri che il fattore di forma differenziale F_{pK} da questo punto alla ciotola è $\frac{1}{2}$.
- (v) Si concluda che il fattore di forma globale F_{KK} vale $1/2$. Poi si ridimostri in modo differente che $F_{KK} = 1/2$ usando argomenti basati non sul calcolo analitico ma sulle proprietà di simmetria dei fattori di forma, come nel precedente Esercizio 6.4.4, ma con una scomposizione della scena diversa, basata su due calotte sferiche concentriche ed estruse.
- (vi) Si ripeta il calcolo, svolto nella seconda parte del precedente Esercizio 6.4.4, del fattore di forma dalla ciotola sferica S_θ di ampiezza angolare θ a sé stessa, ma mediante la stessa scena dell'ultima parte del punto precedente (rammentiamo che S_θ consiste di tutti i punti della sfera il cui angolo di latitudine differisce al massimo di θ dal polo).

Svolgimento. Parte (i). È ovvio: la semisfera è la metà inferiore della sfera di raggio 1 e centro in $(0, 0, 1)$.

Parte (ii): si consideri la restrizione di K al piano x, z. I punti terminali di K in questo piano sono $(1, 1)$. Sia θ l'angolo di deviazione dei punti di K dalla direzione verticale (misurata dall'origine):

quindi per i punti di K la deviazione è $\pi/4 \leqslant \theta \leqslant \pi/2$. Completiamo K ad una sfera intera, e chiamiamo C la semisfera così aggiunta, il cui bordo è J. Chiaramente, per i punti di C, si ha $0 \leqslant \theta \leqslant \pi/4$. Per la identità di conservazione dell'energia (6.2.1), il fattore di forma differenziale è $F_{0K} = 1 - F_{0C}$ (la somma di tutti i fattori di forma differenziali da un punto a tutta la scena è il 100% dell'angolo solido totale, ossia 1). Il calcolo di F_{0C}, o equivalentemente F_{0B}, è analogo a quello svolto nel precedente Esercizio 6.4.1, o nell'Esercizio 6.4.2, e vale $\sin^2 \frac{\pi}{4} = \frac{1}{2}$. Pertanto, $F_{0K} = 1 - F_{0C} = 1 - \sin^2(\pi/4) = 1 - \frac{1}{2} = \frac{1}{2}$.

Parte (*iii*): l'integrale è

$$F_{vK} = \int_K \frac{\cos\theta_v \, \cos\theta_p}{\pi r^2} \, V_{01} \, d\sigma(p) \, .$$

Qui il fattore di occlusione V_{01} vale costantemente 1 perché tutti i punti di K sono visibili da v; r è la distanza fra v e p, ossia la lunghezza del segmento $\boldsymbol{s}$ che li unisce, in altre parole

$$r^2 = (x_p - x_v)^2 + (y_p - y_v)^2 + (z_p - z_v)^2; \tag{6.4.4}$$

θ_v e θ_p sono gli angoli che le normali a K in v e p (ciascuna coincidente con il rispettivo versore radiale, diretto verso il centro $(0, 0, 1)$) formano con il segmento $\boldsymbol{s}$. Chiamiamo o il centro $(0, 0, 1)$ e consideriamo il triangolo $T_K = (o, v, p)$. I due lati che si incontrano in o hanno lunghezza 1 (pari al raggio!): ossia, il triangolo è isoscele. Pertanto ciascun coseno nell'integrando è uguale alla lunghezza della proiezione del corrispondente lato di lunghezza 1 sopra il terzo lato $\boldsymbol{s}$ che ha lunghezza r, ed evidentemente, visto che il triangolo è isoscele, le due proiezioni sono di lunghezza uguale. Quindi si ha

$$\cos\theta_v = \cos\theta_p = \frac{r}{2} \, . \tag{6.4.5}$$

Ora, in base a (6.4.5), l'integrale diventa

$$F_{vK} = \frac{1}{4\pi} \int_K d\sigma(p) = \frac{1}{4\pi} \, \text{Area}\,(K) = \frac{1}{2} \, .$$

Parte (*iv*): basta completare la semisfera ad una sfera, aggiungendo una semisfera superiore. Un punto sul bordo J, per simmetria, ha lo stesso fattore di forma verso la semisfera bassa K e la semisfera alta.

Parte (*v*): Poiché tutti i fattori di forma differenziali $F_{\boldsymbol{v}K}$ valgono 1/2, è chiaro che anche $F_{KK} = 1/2$. Diamo una dimostrazione alternativa basata sulle simmetrie dei fattori di forma.

Si consideri la scena composta dalla ciotola K, di raggio, diciamo R, e dal suo coperchio B (un disco di raggio R). Si scavi da B un disco di raggio $r < R$ centrato nel centro di B: in tal modo, invece di B, rimane una corona circolare D. Si chiuda la scena incernierando sul bordo interno di D una ciotola emisferica C_- di raggio r estrusa, ossia dal lato opposto di K (ovvero sporgente verso la direzione positiva dell'asse z). Scriviamo

$$\alpha := F_{KK}$$
$$\beta := F_{C_- K} \, .$$

Dobbiamo calcolare α.
Notiamo che

(1) D vede solo K, e quindi $F_{DC_-} = F_{DD} = 0$. Allora, in base a (6.2.1), $F_{DK} = 1$. Da quest'ultima identità e dalla relazione di reciprocità (6.2.2), osservando che l'area di K vale $2\pi R^2$ e quella di D vale $\pi(R^2 - r^2)$, ottieniamo

$$F_{KD} = \frac{R^2 - r^2}{2R^2} = \frac{1}{2} \left(1 - \frac{r^2}{R^2}\right) .$$

(2) La ciotola C_- si ottiene da K tramite una dilatazione ed una traslazione. Pertanto, in base alla Nota 6.3.3,
$$F_{C_-C_-} = F_{KK} = \alpha.$$
(3) C_- vede sé stessa e K, ma non D. Quindi, ancora da (6.2.1),
$$1 = F_{C_-C_-} + F_{C_-K} = \alpha + \beta.$$
(4) Infine, una terza volta per (6.2.1), $1 = F_{KK} + F_{KD} + F_{KC_-}$. Ma da (6.2.2) segue che
$$F_{KC_-} = \frac{r^2}{R^2}\, F_{C_-K} = \beta\, \frac{r^2}{R^2}\ .$$
Pertanto l'uguaglianza precedente diventa
$$\alpha + \frac{1}{2}\left(1 - \frac{r^2}{R^2}\right) + \beta\, \frac{r^2}{R^2} = 1.$$

Poichè $\beta = 1 - \alpha$ l'ultima uguaglianza diventa
$$\alpha\left(1 - \frac{r^2}{R^2}\right) = 1 - \frac{r^2}{R^2} - \frac{1}{2}\left(1 - \frac{r^2}{R^2}\right) = \frac{1}{2}\left(1 - \frac{r^2}{R^2}\right).$$
Da qui, poiché si è scelto $r < R$, segue $\alpha = \frac{1}{2}$. Pertanto anche $\beta = 1/2$, e $F_{KC_-} = \frac{r^2}{2R^2}$.

Si osservi che questo argomento funziona solo grazie all'aver dissimetrizzato le ciotole inferiore e superiore rendendo una delle due più piccola dell'altra. Infatti, se il pavimento D non c'è e $r = R$, allora le informazioni ottenute da (6.2.1) sono il sistema lineare
$$F_{KK} + F_{KC_-} = 1,$$
$$F_{C_-C_-} + F_{C_-K} = 1.$$
Ma $F_{KC_-} = F_{C_-K}$, e l'identità (6.2.2) diventa $F_{KC_-} = F_{C_-K}$. Quindi, nel sistema lineare, le due righe sono la stessa equazione, ed il problema non si risolve.

Parte (vi): Si consideri la calotta sferica S_θ con $\theta \leqslant \frac{\pi}{2}$ di raggio di base r (ossia il cui disco di base B ha raggio r). Si noti che il raggio della sfera da cui è ritagliata la calotta è $R \geqslant r$; se $R = r$ allora $\theta = \frac{\pi}{2}$ e la calotta è un emisfero, mentre se $R \to \infty$ la calotta si approssima al disco di base B, e $\theta \to 0$).
Si chiuda la scena con una calotta emisferica $S = S_{\frac{\pi}{2}}$ che protrude nella direzione opposta, il cui fattore di forma vale $1/2$, come visto nella parte (v). La conservazione dell'energia (6.2.1) assicura che
$$F_{SS} + F_{SS_\theta} = 1, \tag{6.4.6a}$$
$$F_{S_\theta S} + F_{S_\theta S_\theta} = 1. \tag{6.4.6b}$$
Poiché $F_{SS} = 1/2$, la prima identità mostra che $F_{SS_\theta} = \frac{1}{2}$ (si noti che questo fattore di forma non dipende da θ: è immediato che questo segue anche dal Corollario 6.3.2 (lo si deduca esplicitamente). Sia $A_S = 2\pi r^2$ l'area dell'emisfero S, e A_{S_θ} l'area della calotta sferica S_θ. Calcoliamo A_{S_θ}:
$$A_{S_\theta} = \int_0^{2\pi}\int_0^{\theta} R^2 \sin\theta'\, d\theta'\, d\phi = 2\pi R^2(1 - \cos\theta)$$
(qui si è usato il valore $R^2 \sin\theta$ dello jacobiano delle coordinate sferiche: si veda la Nota 5.2.5). Quindi
$$\frac{A_S}{A_{S_\theta}} = \frac{r^2}{R^2}\ \frac{1}{1 - \cos\theta}\ .$$

Poiché R è il raggio della sfera da cui è ritagliata la calotta S_θ e r è il raggio della base della calotta, è immediato che l'angolo θ di apertura polare della calotta verifica $\sin\theta = \frac{r}{R}$. Quindi

$$\frac{A_S}{A_{S_\theta}} = \frac{\sin^2\theta}{(1-\cos\theta)} .$$

Ora, dall'equazione di reciprocità (6.2.2) sappiamo che $A_{S_\theta} F_{S_\theta S} = A_S F_{SS_\theta}$, e quindi

$$F_{S_\theta S} = \frac{A_S}{A_{S_\theta}}\, F_{SS_\theta} = \frac{1}{2}\, \frac{A_S}{A_{S_\theta}} = \frac{\sin^2\theta}{2(1-\cos\theta)} .$$

Pertanto, dalla seconda identità (6.4.6) si trova

$$F_{S_\theta S_\theta} = 1 - F_{S_\theta S} = 1 - \frac{\sin^2\theta}{2(1-\cos\theta)} = \frac{(1-\cos\theta)^2}{2(1-\cos\theta)} = \frac{1}{2}\,(1-\cos\theta).$$

Si noti che, se $\theta = \frac{\pi}{2}$, allora S_θ è l'emisfero $S = S_{\frac{\pi}{2}}$ e sappiamo che $F_{SS} = \frac{1}{2}$, mentre se $\theta = 0$ allora S_θ è il disco di base B, che è piatto, e quindi $F_{S_0 S_0} = 0$. Entrambi questi risultati sono verificati dalla formula precedente. □

Esercizio 6.4.6. Si consideri la variante della scena nella risoluzione della Parte (vi) del precedente Esercizio 6.4.5 nella quale si incernierano una calotta sferica di raggio R ed apertura polare θ (quindi di raggio di base r che soddisfa $\sin\theta = \frac{R}{r}$) ed un cono retto C di altezza h estruso dalla parte opposta, incernierato alla calotta sulla circonferenza di base di raggio r. Il ragionamento della parte (vi) del precedente Esercizio 6.4.5 per permette di calcolare $F_{S_\theta S_\theta}$ una volta noto F_{CC}. Per calcolare il fattore di forma F_{CC}, si osservi, come in quell'esercizio, che dalla conservazione dell'energia segue $F_{CC} + F_{CS} = 1$, $F_{SC} + F_{SS} = 1$, dove qui S è la ciotola S_θ o una ciotola emisferica (non importa, perché di entrambe abbiamo ormai calcolato l'autofattore di forma nell'esercizio precedente). Questo ci dà F_{SC}, ma allora otteniamo F_{CS} dalla relazione di reciprocità (6.2.2): $A_C F_{CS} = A_S F_{SC}$, pur di calcolare l'area A_C del cono (quella della ciotola emisferica o della calotta sferica le abbiamo già calcolate nel precedente Esercizio 6.4.5). Quest'area si calcola con un facile integrale in coordinate cilindriche, e vale $\pi h a = \pi h\sqrt{h^2+r^2}$, dove a è la lunghezza dell'apotema e r quella del raggio di base (si scriva e si svolga questo integrale).

(oppure si usi il risultato di quella parte dell'esercizio per calcolare direttamente F_{CC} tramite le usuali considerazioni di simmetria). □

6.5. Esempio: calcolo analitico del fattore di forma fra due pareti adiacenti di una stanza cubica

Vogliamo dimostrare che il fattore di forma F fra due quadrati con lo stesso lato, incernierati su un lato e disposti perpendicolarmente, vale $\frac{1}{5}$ (e di conseguenza, per la relazione (6.2.1) di conservazione dell'energia, anche il fattore di forma fra due pareti frontali di un cubo vale $\frac{1}{5}$).

Osserviamo che, fra i fattori di forma in una stanza i cui sei elementi sono le sei pareti, questo è quello numericamente più delicato, in quanto tale fattore di forma è un integrale improprio quadridimensionale con ordine di divergenza $1/r^4$, dove r è la distanza fra punti del primo e del secondo quadrato (si veda l'identità (6.5.1) nel seguito); ma il suo calcolo, che si conclude in (6.5.10), conferma che l'integrale improprio è convergente.

Introduciamo un opportuno sistema di coordinate cartesiane. Disponiamo i due quadrati uno sul piano $\{z = 0\}$ (lo chiamiamo base, o pavimento, e lo indichiamo con $\boldsymbol{b}$) e l'altro sul piano $\{x = 0\}$ (lo chiamiamo parete, od altezza, e lo indichiamo con $\boldsymbol{a}$). Il fattore di forma rimane lo stesso se dilatiamo i due quadrati di uno stesso fattore, perché l'angolo solido differenziale che uno dei due copre quando visto dai punti dall'altro rimane invariato sotto la dilatazione. Quindi possiamo, per semplicità e senza perdita di generalità, assumere che il lato dei quadrati sia lungo 1. Per fissare le

notazioni, diciamo che l'intersezione dei due elementi è il segmento $[0,1]$ sull'asse y. Quindi la base ha coordinate $\{0 \leqslant x \leqslant 1,\ 0 \leqslant y \leqslant 1\}$ e la parete ha coordinate $\{0 \leqslant y \leqslant 1,\ 0 \leqslant z \leqslant 1\}$.
Ricordando che $\boldsymbol{b}$ è un quadrato di lato 1, e quindi di area 1, scriviamo l'integrale del fattore di forma dalla base alla parete nel modo seguente come integrale quadruplo rispetto alle variabili (y_a, z) dei punti $\boldsymbol{p_a} \in \boldsymbol{a}$ ed alle variabili (x, y_b) dei punti $\boldsymbol{p_b} \in \boldsymbol{b}$). Osserviamo che ora gli angoli θ_a (rispettivamente θ_b) sono sottesi dai segmenti $\boldsymbol{p_b} - \boldsymbol{p_a}$ e $\boldsymbol{n_{p_a}}$ (rispettivamente $\boldsymbol{n_{p_a}}$), e quindi non sono angoli su piani parallei agli assi coordinati (tranne il caso $y_b = y_a$), vediamo che

$$F := \frac{1}{\pi}\,\frac{1}{\mathrm{Area}(\boldsymbol{b})}\int_{\boldsymbol{b}}\int_{\boldsymbol{a}} \frac{\cos\theta_{\boldsymbol{b}}\,\cos\theta_{\boldsymbol{a}}}{r^2}\,d\sigma_{\boldsymbol{a}}\,d\sigma_{\boldsymbol{b}}$$

$$= \frac{1}{\pi}\int_0^1\int_0^1\int_0^1\int_0^1 \frac{xz}{(x^2+(y_a-y_b)^2+z^2)^2}\,dx\,dy_a\,dy_b\,dz\,. \quad (6.5.1)$$

Il cambiamento di variabili

$$s = \frac{y_a - y_b}{\sqrt{x^2+z^2}}$$

trasforma l'integrale al lato di destra di (6.5.1) in

$$F = \frac{1}{\pi}\int_0^1\int_0^1\int_0^1\int_{-y_b/\sqrt{x^2+z^2}}^{(1-y_b)/\sqrt{x^2+z^2}} \frac{xz\sqrt{x^2+z^2}}{(x^2+z^2)^2(1+s^2)^2}\;ds\,dx\,dy_b\,dz \quad (6.5.2)$$

Calcoliamo separatamente l'integrale rispetto alla variabile s nel prossimo esercizio elementare di Calcolo:

Esercizio 6.5.1. La primitiva di $1/(1+s^2)^2$ è $\frac{1}{2}\left(\mathrm{arctg}\,s + \frac{s}{1+s^2}\right) + C$.

Svolgimento. Si osservi che

$$\frac{1}{(1+s^2)^2} + \frac{s^2}{(1+s^2)^2} = \frac{1}{1+s^2}$$

e quindi

$$\int \frac{ds}{(1+s^2)^2} = \int \frac{ds}{1+s^2} - \int \frac{s^2\,ds}{(1+s^2)^2} \;= \mathrm{arctg}\,s - \int s\,\frac{s\,ds}{(1+s^2)^2} \quad (6.5.3)$$

Osserviamo che grazie alla sostituzione $u = s^2$, si calcola la primitiva del secondo fattore dell'ultimo integrando:

$$\int \frac{s\,ds}{(1+s^2)^2} = \frac{1}{2}\int \frac{du}{(1+u)^2} = -\frac{1}{2}\frac{1}{1+u} + C = -\frac{1}{2(1+s^2)} + C\,.$$

Quindi l'ultimo integrale in (6.5.3) si può integrare per parti come segue:

$$\int s\,\frac{s\,ds}{(1+s^2)^2} = -\frac{s}{2(1+s^2)} + \int \frac{ds}{2(1+s^2)} = \frac{1}{2}\left(-\frac{s}{1+s^2} + \mathrm{arctg}\,s\right) + C.$$

Pertanto (6.5.3) diventa

$$\int \frac{ds}{(1+s^2)^2} = \mathrm{arctg}\,s - \frac{1}{2}\left(\mathrm{arctg}\,s - \frac{s}{1+s^2}\right) + C$$

$$= \frac{1}{2}\left(\mathrm{arctg}\,s + \frac{s}{1+s^2}\right) + C\,.$$

□

Grazie al risultato del precedente esercizio, l'integrale al lato di destra di (6.5.2) diventa:

$$F = \frac{1}{\pi}\int_0^1\int_0^1\int_0^1 \frac{xz\sqrt{x^2+z^2}}{2(x^2+z^2)^2}\left[\operatorname{arctg}\left(\frac{1-y_b}{\sqrt{x^2+z^2}}\right)\right.$$
$$+\frac{1-y_b}{\sqrt{x^2+z^2}}\frac{1}{1+\left(\frac{1-y_b}{1+\sqrt{x^2+z^2}}\right)^2}+\operatorname{arctg}\left(\frac{y_b}{\sqrt{x^2+z^2}}\right)+$$
$$\left.\frac{y_b}{\sqrt{x^2+z^2}}\frac{1}{1+\left(\frac{y_b}{1+\sqrt{x^2+z^2}}\right)^2}\right]dx\,dy_b\,dz\,. \quad (6.5.4)$$

Spezziamo l'ultimo integrale come la somma di $I_1 + I_2$, dove

$$I_1 := \frac{1}{\pi}\int_0^1\int_0^1\int_0^1 \frac{xz\sqrt{x^2+z^2}}{2(x^2+z^2)^2}\left[\operatorname{arctg}\left(\frac{1-y_b}{\sqrt{x^2+z^2}}\right)\right.$$
$$\left.+\operatorname{arctg}\left(\frac{y_b}{\sqrt{x^2+z^2}}\right)\right]dx\,dy_b\,dz \quad (6.5.5)$$

e

$$I_2 := \frac{1}{\pi}\int_0^1\int_0^1\int_0^1 \frac{xz\sqrt{x^2+z^2}}{2(x^2+z^2)^2}\left[\frac{1-y_b}{\sqrt{x^2+z^2}}\frac{1}{1+\left(\frac{1-y_b}{1+\sqrt{x^2+z^2}}\right)^2}\right.$$
$$\left.+\frac{y_b}{\sqrt{x^2+z^2}}\frac{1}{1+\left(\frac{y_b}{1+\sqrt{x^2+z^2}}\right)^2}\right]dx\,dy_b\,dz\,. \quad (6.5.6)$$

Al fine di calcolare I_1 premettiamo, come richiamo, il ben noto calcolo (per parti) della primitiva dell'arcotangente:

Esercizio 6.5.2.

$$\int_0^x \operatorname{arctg} s\,ds = x\,\operatorname{arctg} x - \frac{1}{2}\,\ln(1+x^2)\,.$$

Svolgimento. Immaginiamo l'integrando a primo membro come il prodotto dell'arcotangente e della funzione costante 1. Integrando per parti otteniamo

$$\int_0^x \operatorname{arctg} s\,ds = x\,\operatorname{arctg} x - \int_0^x \frac{s}{1+s^2}\,ds = x\,\operatorname{arctg} x - \frac{1}{2}\int_0^{x^2}\frac{1}{1+u}\,du$$
$$= x\,\operatorname{arctg} x - \frac{1}{2}\ln(1+x^2)\,.$$

□

Adesso possiamo calcolare l'integrale I_1 in (6.5.5). Applicando al primo integrale in (6.5.5) il cambiamento di variabili $s = (y_b - 1)/(x^2+z^2)$, ed al secondo $s = y_b/(x^2+z^2)$, otteniamo, grazie alla

disparità della arcotangente ed al precedente esercizio:

$$
\begin{aligned}
I_1 &= \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz\sqrt{x^2+z^2}}{2(x^2+z^2)}\left(\int_{-1/(x^2+z^2)}^0 \operatorname{arctg}\left(s(-\sqrt{x^2+z^2})\right)\right.\\
&\left.+\int_0^{1/(x^2+z^2)} \operatorname{arctg}\left(s\sqrt{x^2+z^2}\right)\right) ds\,dx\,dz\\
&= \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz}{x^2+z^2}\int_0^{1/\sqrt{x^2+z^2}} \operatorname{arctg} u\,du\,dx\,dz\\
&= \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz}{x^2+z^2}\left[\frac{1}{\sqrt{x^2+z^2}}\ \operatorname{arctg}\frac{1}{\sqrt{x^2+z^2}}\right. \qquad (6.5.7)\\
&\left.-\frac{1}{2}\ln\left(1+\frac{1}{x^2+z^2}\right)\right] dx\,dz\,. \qquad (6.5.8)
\end{aligned}
$$

Ora, grazie agli stessi due cambiamenti di variabile che abbiamo applicato a I_1, calcoliamo l'integrale I_2 in (6.5.6):

$$
\begin{aligned}
I_2 &= \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz\sqrt{x^2+z^2}}{2(x^2+z^2)^2}\left(\int_{1/\sqrt{x^2+z^2}}^0 \frac{s}{1+s^2}\left(-\sqrt{x^2+z^2}\right)\right.\\
&\left.+\int_0^{1/\sqrt{x^2+z^2}} \frac{s}{1+s^2}\left(\sqrt{x^2+z^2}\right)\, ds\,dx\,dz\right)\\
&= \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz}{x^2+z^2}\int_0^{1/\sqrt{x^2+z^2}} \frac{s}{1+s^2}\,ds\,dx\,dz\\
&= \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz}{x^2+z^2}\left[\frac{1}{2}\ \ln\left(1+\frac{1}{x^2+z^2}\right)\right]\, dx\,dz\,. \qquad (6.5.9)
\end{aligned}
$$

Nel riunire i risultati di (6.5.7) e (6.5.9) i termini con il logaritmo si cancellano e si ottiene:

$$
F = I_1 + I_2 = \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz}{x^2+z^2}\left[\frac{1}{\sqrt{x^2+z^2}}\ \operatorname{arctg}\frac{1}{\sqrt{x^2+z^2}}\right] dx\,dz.
$$

Quest'ultimo integrale si calcola passando a coordinate cilindriche, $x = \rho\cos\theta$, $z = \rho\sin\theta$, ossia $\rho = \sqrt{x^2+z^2}$, come ora mostriamo. Prima, però, osserviamo che per $\rho \sim 0$ l'integrando ha ordine di infinito $1/\rho$, e quindi, come sapevamo che doveva succedere, l'integrale doppio converge (dopo il passaggio a coordinate cilindriche, il fattore ρ nell'elemento di integrazione elimina la singolarità).

Ecco il calcolo:

$$
\begin{aligned}
F &= \frac{1}{\pi}\int_0^1\int_0^1 \frac{xz}{x^2+z^2}\left(\frac{1}{\sqrt{x^2+z^2}}\ \operatorname{arctg}\frac{1}{\sqrt{x^2+z^2}}\right)\,dx\,dz \\
&= \frac{2}{\pi}\int_0^{\frac{\pi}{4}}\int_0^{1/\cos\theta} \sin\theta\,\cos\theta\,\operatorname{arctg}\frac{1}{\rho}\,d\rho\,d\theta \\
&= \frac{2}{\pi}\int_0^{\frac{\pi}{4}} \sin\theta\cos\theta\left[\rho\,\operatorname{arctg}\frac{1}{\rho}+\frac{1}{2}\ln(1+\rho^2)\right]_0^{1/\cos\theta} d\theta \\
&= \frac{2}{\pi}\int_0^{\frac{\pi}{4}} \sin\theta\,\cos\theta\,\left[\frac{1}{\cos\theta}\,\operatorname{arctg}\cos\theta+\frac{1}{2}\ln\left(1+\frac{1}{\cos^2\theta}\right)\right]\,d\theta \\
&= \frac{2}{\pi}\int_1^{\sqrt{2}/2} t\sqrt{1-t^2}\left[\frac{1}{t}\,\operatorname{arctg}\,t+\frac{1}{2}\,\ln\left(1+\frac{1}{t^2}\right)\right]\left(-\frac{1}{\sqrt{1-t^2}}\right)\,dt \\
&= \frac{2}{\pi}\int_{\sqrt{2}/2}^1 \operatorname{arctg}\,t+\frac{1}{2}t\ln\left(1+\frac{1}{t^2}\right)\,dt \\
&= \frac{2}{\pi}\left[t\operatorname{arctg}\,t-\frac{1}{2}\ln(1+t^2)\right]_{\sqrt{2}/2}^1 \\
&\qquad\qquad + \frac{1}{2\pi}\left[t^2\ln\left(1+\frac{1}{t^2}\right)+\ln(1+t^2)\right]_{\sqrt{2}/2}^1 \\
&= \frac{2}{\pi}\left(\frac{\pi}{4}-\frac{1}{2}\,\ln 2-\frac{\sqrt{2}}{2}\,\operatorname{arctg}\frac{\sqrt{2}}{2}+\frac{1}{2}\,\ln\frac{3}{2}\right) \\
&\qquad\qquad + \frac{1}{2\pi}\left(2\ln 2-\frac{1}{2}\,\ln 3-\ln\frac{3}{2}\right) \\
&= \frac{1}{2}-\frac{\sqrt{2}}{\pi}\,\operatorname{arctg}\,\frac{\sqrt{2}}{2}-\frac{2}{\pi}\,\ln 2+\frac{3}{2\pi}\,\ln 2+\frac{1}{\pi}\,(1-\frac{3}{4})\,\ln 3 \\
&= \frac{1}{2}-\frac{\sqrt{2}}{\pi}\,\operatorname{arctg}\,\frac{\sqrt{2}}{2}-\frac{1}{2\pi}\,\ln 2+\frac{1}{4\pi}\,\ln 3\,. \qquad (6.5.10)
\end{aligned}
$$

Sorprendentemente, l'ultima espressione vale quasi esattamente $\frac{1}{5}$, come si può verificare con una calcolatrice (la Figura 6.5.1 è il listato del calcolo eseguito in Matlab). In realtà il valore di $\frac{1}{2}-\frac{\sqrt{2}}{\pi}\,\operatorname{arctg}\,\frac{\sqrt{2}}{2}-\frac{1}{2\pi}\,\ln 2+\frac{1}{4\pi}\,\ln 3$, preciso a 50 cifre decimali, è

$$0.20004377607540315423960018720173281797044005757744,$$

che approssimeremo a $0.2 = 1/5$ nel resto di questo Capitolo. Quindi, in una scena cubica, tutte e cinque le facce, viste dalla sesta faccia, hanno lo stesso fattore di forma.

```
Fattore_di_forma =

1/2-(sqrt(2)/pi)*atan(sqrt(2)/2)-(1/(2*pi))*(log(2)-log(3)/2)


Valore =

    0.2000
```

FIGURA 6.5.1. Valore numerico calcolato in Matlab dell'espressione in (6.5.10)) riguardo il fattore di forma fra quadrati perpendicolari adiacenti.

6.6. Sottostrutturazione

Più fine è la suddivisione in elementi, migliore è il risultato ottenuto, ma il prezzo da pagare è un notevole incremento del tempo di calcolo perché il numero di fattori di forma è il quadrato del numero degli elementi, ed il calcolo di ciascun fattore di forma può essere laborioso. Per evitare questo incremento quadratico del numero dei fattori di forma, alcuni studiosi hanno proposto una suddivisione variabile degli elementi, basata sul ripartirli ulteriormente in sottoelementi laddove la radiosità varia più rapidamente. Per velocizzare il procedimento, i sottoelementi non vengono trattate come gli elementi originari. Infatti, per ciascun sottoelemento s si calcola, usando il metodo del semicubo, solamente il fattore di forma F_{sj} da s verso ciascuno degli elementi j originari (che misura quanta luce arriva al sottoelemento s a partire da j), mentre i fattori di forma dagli elementi ai sottoelementi, $F_{js} = F_{sj}A_s/A_j$, non vengono utilizzati nel calcolo. In tal modo miglioriamo il calcolo della distribuzione di energia che arriva agli elementi spezzati in sottoelementi, considerando la geometria della ricezione di luce in modo più fine, spezzando le aree di ricezione laddove l'intensità è elevata, mentre non raffiniamo la distribuzione dell'emissione. Dopo che un elemento i è stato suddiviso, il suo fattore di forma verso un qualsiasi altro elemento, precedentemente calcolato con approssimazione emicubica, viene sostituito con una stima più accurate, e precisamente con la media dei fattori di forma dei suoi sottoelementi, pesate in proporzione alle aree:

$$F_{ij} = \frac{1}{A_i}\sum_{s=1}^{m} F_{sj}A_s$$

Si noti invece che il calcolo di F_{ij} tramite z-buffer emicubico è preciso solo se si assume che l'elemento i sia piccolo rispetto alla base dell'emicubo e collocato vicino al centro della base (si veda la fine della Sezione 6.3), ma in tale situazione questo fattore di forma di fatto è il fattore di forma infinitesimo dal centro dell'emicubo.

A questo punto si calcola come prima la radiosità b_j di tutti gli elementi originali, in base al sistema lineare della radiosità (6.2.4). Una volta che sia già stata calcolata la radiosità degli elementi originali, la radiosità di ogni sottoelemento s di un elemento i viene calcolata riscrivendo il sistema lineare (6.2.4) rispetto ai sottoelementi, ma mantenendo per ciascun sottoelemento il coefficiente di riflessione dell'elemento da cui proviene (i coefficienti di riflessione per gli elementi originari sono assegnati in fase di modellazione: non abbiamo a disposizione una stima più precisa per quelli dei sottoelementi). Di solito viene anche mantenuta per il sottoelemento la componente e_i di energia luminosa creata internamente assegnata all'elemento da cui esso proviene, perché si è interessati ad una migliore distribuzione dell'energia ricevuta e non di quella emessa (e quindi normalmente non si sottostrutturano le sorgenti di luce). Comunque, se si decide di spezzare in sottoelementi anche

le sorgenti di luce, si può usare, per ciascun elemento s, un suo valore specifico e_s di energia creata, col vincolo che la media pesata per area degli e_s al variare di s in i deve coincidere con e_i.

Quindi abbiamo ora una stima più raffinata anche della radiosità dei sottoelementi, come media pesata in proporzione alle aree:

$$b_s = e_i + \rho_i \sum_{j=1}^{n} b_j F_{sj} \, .$$

Questa stima è stata ottenuta a partire dai valori già noti di b_j e quindi non richiede la risoluzione di un sistema lineare della radiosità a dimensione più elevata: in altre parole, quando si calcola b_s, nel lato di destra dell'equazione i b_j non sono più incognite. Peraltro, essa tiene conto in modo più accurato di come la illuminazione si distribuisce sui sottoelementi, ma non di come questi la irradiano a propria volta in modo più appropriato: per il calcolo della radiosità degli altri elementi si usano sempre i b_i originali, non i b_s della sottostrutturazione. L'algoritmo procede per iterazione, suddividendo gli elementi nelle zone dove la radiosità è maggiore, fino a quando il miglioramento ottenuto dalla suddivisione non raggiunge una soglia sufficientemente piccola prefissata. Gli elementi ottenuti da questo processo vengono infine usati per determinare la radiosità nei vertici, nel modo spiegato nella Sottosezione 6.2.1.

6.7. Soluzione con metodi iterativi

6.7.1. Metodo iterativo di Jacobi. Il sistema lineare della radiosità (6.2.5) è del tipo $M\boldsymbol{b} = \boldsymbol{e}$, dove $\boldsymbol{e}$ è il vettore dell'energia (o meglio potenza) luminosa creata dai singoli elementi (luce propria) e $\boldsymbol{b}$ è il vettore della radiosità, cioè dell'energia (meglio, potenza) luminosa totale diffusa (quella creata o riflessa).

Se ρ_i indica il coefficiente di riflettività dell'elemento numero i e F_{ii} è il fattore di forma dall'elemento i a se stesso (che misura quale frazione dell'energia emessa dall'elemento i ritorna a quell'elemento), si ha:

$$0 \leqslant \rho_i < 1 \qquad \text{e} \qquad 0 \leqslant F_{ii} \leqslant 1.$$

Abbiamo visto in (6.2.4) che la matrice M del sistema lineare della radiosità verifica:

$$m_{ii} = 1 - \rho_i F_{ii} \qquad (i = 1 \ldots, n).$$

Abbiamo poi postulato in (6.2.6) che si abbia $m_{ii} > 0$ per ogni i. Quindi $0 < m_{ii} \leqslant 1$ per ogni i, ed il sistema $M\boldsymbol{b} = \boldsymbol{e}$ diventa

$$b_i = -\sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_{ij}}{m_{ii}} \, b_j + \frac{e_i}{m_{ii}} \qquad (i = 1 \ldots, n). \tag{6.7.1}$$

Questo sistema si può risolvere con metodi iterativi.

Il più semplice metodo iterativo è quello di Jacobi: si parte con qualche vettore iniziale $\boldsymbol{b}^{(0)}$ al posto della soluzione esatta incognita $\boldsymbol{b}$, e la si sostituisce nel lato destro di (6.7.1), per ottenere una prima approssimazione iterativa:

$$b_i^{(1)} = -\sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_{ij}}{m_{ii}} \, b_j^{(0)} + \frac{e_i}{m_{ii}} \qquad (i = 1 \ldots, n). \tag{6.7.2}$$

Le componenti di $\boldsymbol{b}^{(1)}$ vengono così determinate una dopo l'altra, in un primo ciclo (per i da 1 a n) che chiameremo la prima iterazione, ovvero il primo ciclo di iterazione. Poi si procede in modo

analogo, ciclo dopo ciclo. Alla $k-$sima iterazione determiniamo $\boldsymbol{b}^{(k)}$ da

$$b_i^{(k)} = -\sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{m_{ij}}{m_{ii}}\, b_j^{(k-1)} + \frac{e_i}{m_{ii}} \tag{6.7.3}$$

Si dice che il metodo è convergente se gli approssimanti $\boldsymbol{b}^{(k)}$ convergono ad un vettore limite $\boldsymbol{b}$, che in tal caso è necessariamente soluzione del sistema (6.7.1).

Si noti che tutte le componenti di $\boldsymbol{b}^{(k)}$ vengono aggiornate insieme nello stesso ciclo, utilizzando tutte le componenti di $\boldsymbol{b}^{(k-1)}$: quindi il metodo di Jacobi richiede di conservare in memoria simultaneamente due diversi aggiornamenti del vettore della radiosità: $\boldsymbol{b}^{(k)}$ e $\boldsymbol{b}^{(k-1)}$.

Metodi iterativi di questo tipo si chiamano anche metodi di rilassamento.

6.7.2. La matrice di iterazione di un metodo di rilassamento. Rivediamo la relazione fra il sistema lineare $M\boldsymbol{b} = \boldsymbol{e}$ ed il sistema iterativo (6.7.3) in forma matriciale. In generale, il problema lineare $M\boldsymbol{b} = \boldsymbol{e}$ si trasforma in un metodo iterativo se decomponiamo

$$M = A - B. \tag{6.7.4}$$

Allora il sistema lineare diventa $A\boldsymbol{b} = B\boldsymbol{b} + \boldsymbol{e}$, ossia

$$\boldsymbol{b} = A^{-1}B\boldsymbol{b} + A^{-1}\boldsymbol{e}\,, \tag{6.7.5}$$

da cui la procedura iterativa

$$\boldsymbol{b}^{(k+1)} = A^{-1}B\boldsymbol{b}^{(k)} + A^{-1}\boldsymbol{e}\,. \tag{6.7.6}$$

Nota 6.7.1. Osserviamo che, se esiste $\lim_k \boldsymbol{b}^{(k)} = \boldsymbol{b}$, allora da (6.7.3) segue che $\boldsymbol{b} = \boldsymbol{b} = A^{-1}B\boldsymbol{b} + A^{-1}\boldsymbol{e}$, e quindi $\boldsymbol{b}$ è soluzione del problema (6.7.5). □

Nota 6.7.2. La parte più onerosa del calcolo della prima iterazione consiste nel trovare la matrice inversa A^{-1}: osserviamo però che il calcolo dell'inversa è necessario solo una volta, alla prima iterazione: basta memorizzare l'inversa ed alle iterazioni successive la si può usare senza rideterminarla. □

6.7.3. Il metodo di Jacobi. Allora, spezziamo la matrice M come $M = L + D + U$, somma della sua parte diagonale D, la parte triangolare (strettamente) superiore U e quella triangolare (strettamente) inferiore L, e per comodità poniamo $P = -U$. Per tradizione, nei metodi numerici la matrice triangolare inferiore si indica con L (da *lower triangular*, e quella triangolare superiore con U (da *upper triangular*), ma qui, per comodità, spesso porremo scriveremo $U = -P$. Allora il sistema lineare della radiosità diventa

$$D\boldsymbol{b} = (-L - U)\boldsymbol{b} + \boldsymbol{e}\,. \tag{6.7.7}$$

ed il sistema lineare (6.8.5) diventa

$$D\boldsymbol{b}^{(k+1)} = -L\boldsymbol{b}^{(k)} - U\boldsymbol{b}^{(k)} + \boldsymbol{e} = -L\boldsymbol{b}^{(k)} + P\boldsymbol{b}^{(k)} + \boldsymbol{e}.$$

D'altra parte, sappiamo da (6.2.6) che, nel problema della radiosità, la matrice D, che consiste della parte diagonale della matrice M della radiosità, ha tutti i termini diagonali non nulli, e quindi è invertibile. Pertanto si ottiene

$$\boldsymbol{b}^{(k+1)} = -D^{-1}(L + U)\boldsymbol{b}^{(k)} + D^{-1}\boldsymbol{e}\,. \tag{6.7.8}$$

Chiamiamo questo procedimento iterativo il *metodo di Jacobi*. La sua matrice di iterazione è

$$Q = -D^{-1}(L + U). \tag{6.7.9}$$

e quindi il procedimento iterativo di Jacobi ha la forma seguente:

$$\boldsymbol{b}^{(k+1)} = Q\boldsymbol{b}^{(k)} + D^{-1}\boldsymbol{e}\,. \tag{6.7.10}$$

Ora ne vedremo una variante più efficiente (Gauss–Seidel), ed in seguito un'altra ancora più efficiente (Southwell).

6.8. Rilassamento di Gauss–Seidel; predominanza diagonale stretta

Il metodo iterativo di Gauss–Seidel è simile al metodo di Jacobi, ma con la seguente variante. Il sistema iterativo (6.7.7) viene riscritto come

$$(D + L)\boldsymbol{b} = -U\boldsymbol{b} + \boldsymbol{e}\,. \tag{6.8.1}$$

Lo schema iterativo diventa

$$(D + L)\boldsymbol{b}^{(k+1)} = -U\boldsymbol{b}^{(k)} + \boldsymbol{e}\,, \tag{6.8.2}$$

in cui la parte sotto-diagonale $N = D+L$ (comprensiva dei termini diagonali) della matrice M viene applicata alla nuova approssimazione di ordine $k + 1$ e la parte strettamente sopra-diagonale alla approssimazione k. Però questo comporta calcolare $(D+L)^{-1}$. Allora, in termini di questa matrice, riscriviamo il sistema 6.8.1 così:

$$\boldsymbol{b} = -(D + L)^{-1}U\boldsymbol{b} + (D + L)^{-1}\boldsymbol{e} = -(\mathbb{I} + D^{-1}L)^{-1}D^{-1}U\boldsymbol{b} + (D + L)^{-1}\boldsymbol{e}\,, \tag{6.8.3}$$

perché $D + L = D(\mathbb{I} + D^{-1}L)$, e quindi $(D + L)^{-1} = (\mathbb{I} + D^{-1}L)^{-1}D^{-1}$.
Riverifichiamo: in questo procedimento iterativo la matrice di iterazione è diventata

$$Q := -(\mathbb{I} + D^{-1}L)^{-1}D^{-1}U = -(D + L)^{-1}U = N^{-1}P. \tag{6.8.4}$$

Questo è il meccanismo di Gauss e Seidel. Segue da (6.8.1) che questo equivale a $(D + L)\boldsymbol{b}^{(k+1)} = P\boldsymbol{b}^{(k)} + \boldsymbol{e}$. Questo spiega la differenza fra lo schema iterativo di Jacobi e quello di Gauss–Seidel: essa consiste nel fatto che, quando si svolge il ciclo di iterazione (6.7.3), le n equazioni vengono anche qui risolte una dopo l'altra in ordine progressivo (ossia per i da 1 a n), ma, nella $i-$esima equazione, si utilizzano ove possibile le componenti già precedentemente aggiornate (nelle equazioni precedenti del ciclo, da 1 a $i - 1$) del vettore $\boldsymbol{b}^{(k+1)}$ invece che quelle di $\boldsymbol{b}^{(k)}$. Per le componenti non ancora aggiornate, cioè quelle da $i + 1$ a n, si continua a utilizzare $\boldsymbol{b}^{(k)}$. Quindi, invece di (6.7.3), ossia invece di $b_i^{(k+1)} = -\sum_{j=1}^{i-1} \frac{m_{ij}}{m_{ii}}\, b_j^{(k)} - \sum_{j=i+1}^{n} \frac{m_{ij}}{m_{ii}}\, b_j^{(k)} + \frac{e_i}{m_{ii}}$, ora scriviamo:

$$b_i^{(k+1)} + \sum_{j=1}^{i-1} \frac{m_{ij}}{m_{ii}}\, b_j^{(k+1)} = -\sum_{j=i+1}^{n} \frac{m_{ij}}{m_{ii}}\, b_j^{(k)} + \frac{e_i}{m_{ii}} \tag{6.8.5}$$

Riassumendo, abbiamo spezzato la matrice M come $M = L+D+U$, somma della sua parte diagonale D, la parte triangolare (strettamente) superiore U e quella triangolare (strettamente) inferiore L, ed abbiamo posto $P = -U$. Allora il sistema lineare (6.8.5) diventa

$$D\boldsymbol{b}^{(k+1)} = -L\boldsymbol{b}^{(k+1)} - U\boldsymbol{b}^{(k)} + \boldsymbol{e} = -L\boldsymbol{b}^{(k+1)} + P\boldsymbol{b}^{(k)} + \boldsymbol{e},$$

ossia

$$N\boldsymbol{b}^{(k+1)} = P\boldsymbol{b}^{(k)} + \boldsymbol{e}, \tag{6.8.6}$$

(dove $N = D + L$ è la parte inferiore di M inclusi i termini diagonali), ovvero ancora

$$\boldsymbol{b}^{(k+1)} = N^{-1}P\boldsymbol{b}^{(k)} + N^{-1}\boldsymbol{e}. \tag{6.8.7}$$

NOTA 6.8.1. Dato un operatore lineare M su $\mathbb{C}^n$, i metodi di rilassamento sono mirati a trasformare il problema lineare $M\boldsymbol{b} = \boldsymbol{e}$ in un problema equivalente di punto fisso che sia risolvibile in maniera rapida ed efficiente. Fissata una base in $\mathbb{C}^n$, l'operatore lineare M si identifica con una matrice $n \times n$. La trasformazione del problema si ottiene attraverso un qualsiasi spezzamento della matrice M, come in (6.7.5)

$$M = A - B$$

con A invertibile. Così si ottiene:

$$A\boldsymbol{b} = B\boldsymbol{b} + \boldsymbol{e} \tag{6.8.8}$$

ovvero

$$\boldsymbol{b} = A^{-1}B\boldsymbol{b} + A^{-1}\boldsymbol{e} \tag{6.8.9}$$

Il problema $M\boldsymbol{b} = \boldsymbol{e}$ è quindi trasformato nel problema di punto fisso

$$\boldsymbol{b} = Q\boldsymbol{b} + \boldsymbol{a} \tag{6.8.10}$$

dove

$$\boldsymbol{a} = A^{-1}\boldsymbol{e} \tag{6.8.11a}$$

$$Q = A^{-1}B. \tag{6.8.11b}$$

L'operatore Q è chiamato l'operatore (o la matrice) di iterazione: lo schema iterativo è

$$\boldsymbol{b}^{(k+1)} = Q\boldsymbol{b}^{(k)} + \boldsymbol{a}. \tag{6.8.12}$$

Però questo approccio è numericamente vantaggioso solo se, alla $(k+1)-$sima iterazione, questo sistema lineare, che equivale al sistema lineare (6.8.8)

$$A\boldsymbol{y} = P\boldsymbol{b}^{(k)} + \boldsymbol{e}$$

(in cui il termine di destra è un vettore noto e $\boldsymbol{y}$ è un vettore incognito) è risolvibile per $\boldsymbol{y}$ in maniera rapida, cioè con un tempo di calcolo trascurabile rispetto a quello richiesto per la risoluzione del problema originale $M\boldsymbol{b} = \boldsymbol{e}$. Per questo è particolarmente utile il caso in cui l'operatore invertibile A sia espresso da una matrice triangolare inferiore, perché in questo caso la soluzione si ottiene in maniera rapida ed immediata: per sostituzioni successive di ogni equazione del sistema lineare nella precedente. Perciò si sceglie $A = N$. □

6.9. Analisi della convergenza dei metodi di Jacobi e di Gauss–Seidel

6.9.1. Convergenza di metodi di rilassamento.

DEFINIZIONE 6.9.1 (Raggio spettrale). Il raggio spettrale di un operatore lineare T su $\mathbb{C}^n$ (munito della norma euclidea, ossia la norme ℓ^2) è definito da

$$\rho_T = \limsup_{k\to\infty} \|T^k\|^{\frac{1}{k}}.$$

È ovvio che $\rho_T \leqslant \|T\|$, perché $\|UV\| \leqslant \|U\|\,\|V\|$ per ogni coppia di operatori lineari U, V su $\mathbb{C}^n$, ed in particolare $\|T^k\| \leqslant \|T\|^k$ per ogni k. In particolare, se il raggio spettrale di T è minore di 1 e T è diagonalizzabile, allora $\|T\|$ è il massimo modulo degli autovalori (per vederlo basta cambiare base e metterci nella base in cui T ha forma diagonale: il raggio spettrale e la norma non dipendoo dalla scelta della base). Ne segue che anche $\|T\| < 1$ e T è un operatore di contrazione, ossia $\|T(\boldsymbol{u} - \boldsymbol{v})\| \leqslant \|u - v\|$ per ogni coppia di vettori $\boldsymbol{u} \neq \boldsymbol{v}$. Viceversa, se T è una contrazione allora $\|T\| \leqslant 1$ e quindi $\rho_T \leqslant 1$.
Invece, in generale non è vero che $\|(UV)^k\| \leqslant \|U^k\, V^k\|$ (a meno che gli operatori U e V commutino), e quindi che $\rho_{UV} \leqslant \rho_U \rho_V$. Analogamente, di solito non è vero che $\rho_{U+V} = \rho_U + \rho_V$ (basta prendere $V = -U$).

TEOREMA 6.9.2 (Convergenza ad un punto fisso di metodi iterativi). *Condizione necessaria e sufficiente affinché il metodo iterativo* (6.8.12) *con matrice di iterazione Q converga per ogni scelta di vettore iniziale è che $\rho_Q < 1$.*
Per specifiche scelte del vettore iniziale, l'errore tende a zero anche senza che Q abbia raggio spettrale minore di 1. Ciò accade quando definitivamente il nucleo delle matrici Q^k include $\boldsymbol{b}^{(0)} - \boldsymbol{b}$, dove $\boldsymbol{b}$ è il punto fisso: di solito si sceglie $\boldsymbol{b}^{(0)} = 0$, e quindi la condizione diventa che il punto fisso $\boldsymbol{b}$ appartenga a $\ker(Q^k)$ da un certo k in poi (in particolare, basta che $\boldsymbol{b} \in \ker Q$).

DIMOSTRAZIONE. Assumiamo $\rho_Q < 1$ e consideriamo l'incremento alla $k-$sima iterazione ($k > 0$), $\boldsymbol{s}^{(k)} = \boldsymbol{b}^{(k)} - \boldsymbol{b}^{(k-1)}$. Da (6.8.12) e (6.8.10) si ottiene per questi incrementi la relazione di ricorrenza

$$\boldsymbol{s}^{(k+1)} = \boldsymbol{b}^{(k+1)} - \boldsymbol{b}^{(k)} = Q\boldsymbol{b}^{(k)} - Q\boldsymbol{b}^{(k-1)} = Q\boldsymbol{s}^{(k)},$$

e quindi

$$\boldsymbol{s}^{(k+1)} = Q\boldsymbol{s}^{(k)} = Q^2\boldsymbol{s}^{(k-1)} = \cdots = Q^k\boldsymbol{s}^{(1)}. \tag{6.9.1}$$

Se $\rho_Q < 1$, si ha $\|\boldsymbol{s}^{(k+1)}\| = \|Q^k\boldsymbol{s}^{(1)}\| \leqslant \|Q^k\|\|\boldsymbol{s}^{(1)}\|$, che è asintoticamente equivalente a ρ_Q^k se il limite superiore nella Definizione 6.9.1 è un limite, ed in generale è maggiorata da $C\rho_Q^k$, dove C è una qualunque costante maggiore di 1. In altre parole,

$$\|\boldsymbol{s}^{(k+1)}\| = \|Q^k\boldsymbol{s}^{(1)}\| \leqslant C\,\rho_Q^k$$

per qualche costante $C > 1$. Allora, poiché $\rho_Q < 1$, la serie $\sum_k \boldsymbol{s}^{(k)}$ converge in norma, essendo maggiorata in norma da una serie geometrica di ragione minore di 1, quindi convergente.. D'altra parte, la somma parziale $\sum_{k=1}^m \boldsymbol{s}^{(k)}$ è la somma telescopica

$$\sum_{k=1}^{k} \boldsymbol{s}^{(m)} = \sum_{k=1}^{k} (\boldsymbol{b}^{(m)} - \boldsymbol{b}^{(m-1)}) = \boldsymbol{b}^{(k)} - \boldsymbol{b}^{(0)}, \tag{6.9.2}$$

e quindi la convergenza della serie equivale all'esistenza del limite (finito) $\boldsymbol{b} = \lim_{k\to\infty} \boldsymbol{b}^{(k)}$. Questo dimostra che la condizione $\rho(Q) < 1$ è sufficiente per la convergenza del metodo iterativo.

Si osservi che, indicando l'errore alla $k-$sima iterazione ($k > 0$) con $\boldsymbol{\eta}^{(k)} = \boldsymbol{b}^{(k)} - \boldsymbol{b}$, per questi errori, analogamente a prima, grazie a (6.8.10) e (6.8.12), abbiamo $\boldsymbol{b}^{(k+1)} = Q\boldsymbol{b}^{(k)} + \boldsymbol{a}$ e $\boldsymbol{b} = Q\boldsymbol{b} + \boldsymbol{a}$ (per l'ultima identità si veda anche la Nota 6.7.1), e quindi otteniamo la relazione di ricorrenza

$$\boldsymbol{\eta}^{(k+1)} = \boldsymbol{b}^{(k+1)} - \boldsymbol{b} = Q\boldsymbol{b}^{(k)} - Q\boldsymbol{b} = Q\boldsymbol{\eta}^{(k)} = Q^k\boldsymbol{\eta}^{(1)}.$$

In particolare $\boldsymbol{\eta}^{(1)} = Q\boldsymbol{\eta}^{(0)}$. Quindi

$$\boldsymbol{\eta}^{(k+1)} = Q^{(k+1)}\boldsymbol{\eta}^{(0)}.$$

Da qui che segue la seconda asserzione dell'enunciato: se $\boldsymbol{\eta}^{(0)} = \boldsymbol{b}^{(0)} - \boldsymbol{b}$ è nel nucleo di $\cap_{j>k} \ker(Q^j)$, allora $0 = \boldsymbol{\eta}^{(k+1)} = \boldsymbol{\eta}^{(k+2)} = \boldsymbol{\eta}^{(k+3)} = \ldots$, e quindi $\boldsymbol{b}^{(j)} = \boldsymbol{b}$ per ogni $j > k$.

Viceversa, diamo un'idea della dimostrazione che la condizione è necessaria. Assumiamo allora $\rho_Q \geqslant 1$. Rammentiamo che il raggio spettrale di un operatore su uno spazio vettoriale a dimensione finita è il massimo dei moduli dei suoi autovalori. Sia $\boldsymbol{v}$ un autovettore di Q corrispondente all'autovalore di massimo modulo: questo modulo vale quindi $\rho_Q \geqslant 1$, e $Q^k\boldsymbol{v} = \rho_Q^k\,\boldsymbol{v}$ non converge. Più precisamente, se $\rho_Q > 1$, la successione dei vettori $Q^k\boldsymbol{v}$ diverge in norma al crescere di k. Se invece $\rho_Q = 1$, l'autovalore di modulo massimo è del tipo e^{it} per qualche $t \in \mathbb{R}$, e quindi il corrispondente autovettore $\boldsymbol{v}$ viene mandato dalla matrice Q^k in $e^{ikt}\boldsymbol{v}$, una successione di norma costante che ad ogni passo ruota di una fase e^{it} e quindi non converge (tranne nel caso banale $t = 0$, in cui l'autovalore di modulo massimo vale 1 ed il suo autovettore $\boldsymbol{v}$ viene lasciato fisso: ma allora $\boldsymbol{v}$ si trova nel nucleo di $\cap_{j>k} \ker(Q - \mathbb{I})^j$ ed è un punto fisso, ossia una soluzione banale del sistema iterativo).
Riassumendo, se scegliamo $\boldsymbol{b}^{(0)} = \boldsymbol{v}$, vediamo che il procedimento iterativo non converge quando

$\rho_Q \geqslant 1$ se applicato ad un qualunque vettore iniziale che non sia lasciato fisso dalla matrice Q. Questo completa la dimostrazione. □

6.9.2. Convergenza del metodo di Gauss–Seidel.

DEFINIZIONE 6.9.3. Una matrice $n \times n$ M è a predominanza diagonale stretta se, per ogni $i = 1, \ldots, n$ si ha $|m_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |m_{ij}|$.

LEMMA 6.9.4. *Nell'ipotesi che la scena non abbia elementi speculari, ossia che ρ_i sia minore di 1 per ciascun i, la matrice M del sistema della radiosità* (6.7.1) *è a predominanza diagonale stretta.*

DIMOSTRAZIONE. Abbiamo visto nella Sezione (6.2) che, per $1 \leqslant i \leqslant n$, si ha $m_{ii} = 1 - \rho_i F_{ii}$. Inoltre, per $1 \leqslant j \leqslant n$, si ha $\sum_{j=1}^{n} F_{ij} = 1$ (qui supponiamo che anche lo sfondo sia un elemento, e quindi che l'area totale coperta da tutti gli elementi vista dall'elemento numero i sia l'intero angolo solido del semispazio anteriore, di π steradianti). Sempre nella Sezione (6.2) abbiamo anche visto che, per $i \neq j$, si ha $m_{ij} = -\rho_i F_{ij}$. Perciò $m_{ii} = 1 - \rho_i F_{ii} = \sum_{j=1}^{n} F_{ij} - \rho_i F_{ii}$. Ma $\rho_i < 1$ (e quindi $m_{ii} > 0$) per ogni i. Quindi

$$|m_{ii}| \geqslant \sum_{j=1}^{n} F_{ij} - \rho_i F_{ii} > \sum_{j=1}^{n} \rho_i F_{ij} - \rho_i F_{ii} = \sum_{\substack{j=1 \\ j \neq i}}^{n} \rho_i F_{ij} = \sum_{\substack{j=1 \\ j \neq i}}^{n} |m_{ij}|,$$

il che completa la dimostrazione. □

TEOREMA 6.9.5. *Consideriamo un sistema lineare $M\boldsymbol{b} = \boldsymbol{e}$. Se la matrice M è a predominanza diagonale stretta, allora il metodo di Gauss–Seidel è convergente.*

DIMOSTRAZIONE. Consideriamo le iterazioni di $M\boldsymbol{b} = \boldsymbol{e}$ con le sostituzioni successive nel senso di (6.8.5) (dove la matrice, come in (6.8.5), è la matrice M della radiosità). Scriviamo m_{ij} per i coefficienti della matrice M. Rammentiamo che $m_{ii} > 0$ per ogni i. Quindi, se il procedimento iterativo converge, la soluzione $\boldsymbol{b} = (b_1, \ldots, b_n)$ verifica il sistema lineare (6.7.1), ovvero, per ogni $i = 1, \ldots, n$,

$$b_i = -\sum_{j=1}^{i-1} \frac{m_{ij}}{m_{ii}}\, b_j - \sum_{j=i+1}^{n} \frac{m_{ij}}{m_{ii}}\, b_j + \frac{e_i}{m_{ii}}\,,$$

e quindi l'approssimazione di Gauss–Seidel di ordine $k+1$ verifica (6.7.5):

$$b_i^{(k+1)} = -\sum_{j=1}^{i-1} \frac{m_{ij}}{m_{ii}}\, b_j^{(k+1)} - \sum_{j=i+1}^{n} \frac{m_{ij}}{m_{ii}}\, b_j^{(k)} + \frac{e_i}{m_{ii}}\,.$$

Quindi gli errori

$$s_i^{(k)} := b_i^{(k)} - b_i^{(k-1)}$$

risolvono ad ogni iterazione (come già osservato in (6.9.1)), il seguente sistema lineare:

$$s_i^{(k+1)} = -\sum_{j=1}^{i-1} \frac{m_{ij}}{m_{ii}}\, s_j^{(k+1)} - \sum_{j=i+1}^{n} \frac{m_{ij}}{m_{ii}}\, s_j^{(k)}\,.$$

Perciò, per ogni $i = 1, \ldots, n$, si ha

$$\left| s_i^{(k+1)} \right| \leqslant \sum_{j=1}^{i-1} \left| \frac{m_{ij}}{m_{ii}} \right| \, |s_j^{(k+1)}| + \sum_{j=i+1}^{n} \left| \frac{m_{ij}}{m_{ii}} \right| \, |s_j^{(k)}|\,, \tag{6.9.3}$$

e quindi:

$$\left|s_i^{(k+1)}\right| \leqslant \sum_{j=1}^{i-1} \left|\frac{m_{ij}}{m_{ii}}\right| \ \|\boldsymbol{s}^{(k+1)}\| + \sum_{j=i+1}^{n} \left|\frac{m_{ij}}{m_{ii}}\right| \ \|\boldsymbol{s}^{(k)}\| \,. \tag{6.9.4}$$

Qui come norma dei vettori in $\mathbb{C}^n$ abbiamo scelto la norma ℓ^∞: $\|\boldsymbol{s}^{(k)}\| = \max\{|s_i^{(k)}| : 1 \leqslant i \leqslant n\}$. Rammentiamo comunque che in spazi a dimensione finita tutte le norme sono equivalenti.
Detto i_k l'indice i per il quale il secondo termine della disuguaglianza è massimo, e scrivendo per brevità:

$$R_{i_k} = \sum_{j=1}^{i_k-1} \left|\frac{m_{i_k j}}{m_{i_k i_k}}\right|,$$

$$S_{i_k} = \sum_{j=i_k+1}^{n} \left|\frac{m_{i_k j}}{m_{i_k i_k}}\right|.$$

si ottiene una disuguaglianza uniforme rispetto all'indice i, ossia tale che il secondo membro non dipende più da i: per ogni $i = 1, \ldots, n$,

$$\left|s_i^{(k+1)}\right| \leqslant R_{i_k}\|\boldsymbol{s}^{(k+1)}\| + S_{i_k}\|\boldsymbol{s}^{(k)}\|.$$

Quindi, scegliendo come norma la norma ℓ^∞, definita da $\|s^{(k)}\| = \max_{\{1\leqslant i\leqslant n\}} |s_i^{(k)}|$, otteniamo

$$\|\boldsymbol{s}^{(k+1)}\| \leqslant R_{i_k}\|\boldsymbol{s}^{(k+1)}\| + S_{i_k}\|\boldsymbol{s}^{(k)}\|. \tag{6.9.5}$$

Pertanto

$$(1 - R_{i_k})\|\boldsymbol{s}^{(k+1)}\| \leqslant S_{i_k}\|\boldsymbol{s}^{(k)}\|.$$

Ora, se supponiamo che M sia a predominanza diagonale stretta, cioè se

$$|m_{ii}| > \sum_{j\neq i} |m_{ij}|,$$

si ottiene

$$R_{i_k} + S_{i_k} < 1.$$

Ne segue che $R_{i_k} < 1$ e $S_{i_k} < 1$, e quindi

$$\frac{S_{i_k}}{1 - R_{i_k}} \ < 1. \tag{6.9.6}$$

In questa disuguaglianza, l'indice i_k dipende dalla scelta di k. Però i_k varia solo nell'insieme finito da 1 a n, e quindi la costante $\frac{S_{i_k}}{1-R_{i_k}}$ varia in un insieme finito di numeri tutti minori di 1, in base all'ultima disuguaglianza. Indichiamo il suo valore massimo ancora con $\frac{S_C}{1-R_C} < 1$. Allora

$$\|\boldsymbol{s}^{(k+1)}\| \leqslant \frac{S_C}{1 - R_C} \ \|\boldsymbol{s}^{(k)}\| \leqslant \cdots \leqslant \left(\frac{S_C}{1 - R_C}\right)^{k+1} \ \|\boldsymbol{s}^{(0)}\|. \tag{6.9.7}$$

Ora, grazie alla disuguaglianza (6.9.6), segue da (6.9.7) che

$$\lim_{k\to\infty} \|\boldsymbol{s}^{(k)}\| = 0,$$

e la convergenza a zero è a velocità esponenziale. Pertanto, per lo stesso argomento usato dopo 6.9.2 (ossia la convergenza della serie geometrica di ragione minore di 1), questo dimostra l'enunciato. È anche importante osservare che da (6.9.7) e (6.9.6), in base alla Definizione 6.9.1 di raggio spettrale, segue che il raggio spettrale della matrice Q del Teorema 6.9.2 è minore di 1. □

COROLLARIO 6.9.6. *Il metodo iterativo di Gauss–Seidel, applicato al problema della radiosità, è convergente.*

PROBLEMA 1. Il metodo iterativo di Jacobi, applicato al problema della radiosità, è convergente?
Abbiamo dimostrato nel Lemma 6.9.4 che la matrice della radiosità è a predominanza diagonale stretta, e nel Teorema 6.9.2 che se la matrice di iterazione del metodo di Jacobi ha raggio spettrale minore di 1 allora il metodo di Jacobi converge. Però la matrice di iterazione nel metodo di Jacobi non è $N^{-1}P$, ma $\mathbb{I} - D^{-1}M_0$, dove abbiamo posto $M_0 := M - D$ (si veda (6.7.9)). Quale è il suo raggio spettrale?
Ovviamente si ha

$$D^{-1}M_0 = D^{-1}(M - D) = D^{-1}M - \mathbb{I} = D^{-1}(L + D + U) - \mathbb{I} \tag{6.9.8}$$

(mentre la matrice di iterazione di Gauss–Seidel è $N^{-1}P$, come visto in (6.8.4)). Poiche' la norma della somma di due matrici è minore o uguale alla somma delle norme (per la proprietà triangolare della norma), ne segue subito che lo stesso vale per il raggio spettrale. Pertanto $\rho_{\mathbb{I}-D^{-1}M} \leqslant \rho_{\mathbb{I}} + \rho_{D^{-1}M)} = 1+\rho_{D^{-1}M}$. Il lato destro è maggiore di 1, e quindi questa stima non ci dà la maggiorazione necessaria. Da questo quindi non segue che il metodo di Jacobi sia convergente. Per dimostrare a sua convergenza, occorre assumere (o fornire ipotesi che assicurino) che si abbia $\rho_{\mathbb{I}-D^{-1}M} < 1$. In effetti, vedremo nella prossima Sezione che la predominanza diagonale stretta di M porta anche a questa conclusione, e che quindi anche il metodo di Jacobi per la radiosità converge qualunque sia l'approssimazione iniziale.

6.10. Convergenza del metodo di Jacobi e confronto delle velocità di convergenza di Jacobi e Gauss–Seidel

Consideriamo la velocità di convergenza del procedimento iterativo di Jacobi (6.7.10). Il calcolo della matrice inversa D^{-1} è banale: D^{-1} è la matrice diagonale che sulla diagonale ha i reciproci degli elementi corrispondenti di D. Questo velocizza la prima iterazione, ma vedremo che a lungo andare il procedimento diventa più lento: la velocità di convergenza degli approssimanti è più lenta. Quindi questo approccio è sconsigliabile nella pratica numerica, ma è comodo per risolvere problemi in cui sono richieste solo le prime iterazioni.

Ora mostriamo che, comunque, il metodo di Jacobi (6.7.10) converge. Abbiamo visto in (6.7.9) che la sua matrice di iterazione è $Q := -D^{-1}(L + U)$. La condizione necessaria e sufficiente per la convergenza, data dal Teorema 6.9.2, è che il raggio spettrale della matrice $D^{-1}(L + U)$ sia minore di 1. Per dimostrarlo utilizziamo un risultato di analisi numerica, il teorema di Stein e Rosenberg [**44**, Teorema 8.2.14], che asserisce che, se una matrice $A+B$ è non negativa, allora il raggio spettrale $\rho_{(\mathbb{I}-A)^{-1}B}$ di $(\mathbb{I}-A)^{-1}B$ è minore del raggio spettrale ρ_{A+B} di $A+B$, purché quest'ultimo sia inferiore a 1, e che queste due matrici si collocano in uno delle seguenti tre casi: o hanno entrambe raggio spettrale minore di 1, o entrambe maggiore di 1, o entrambe uguale a 1. Poniamo $A := -D^{-1}L$ e $B := -D^{-1}U$, e, come in (6.8.4), $(\mathbb{I} - A)^{-1}B = -(\mathbb{I} + D^{-1}L)^{-1}D^{-1}U = -N^{-1}U = N^{-1}P$, e quest'ultima (che a parte il segno è la matrice di iterazione del metodo di Gauss–Seidel) ha raggio spettrale minore di 1: pertanto siamo nel primo caso. Allora, in base a questo teorema, il raggio spettrale della matrice $A + B = D^{-1}(L + U)$, ossia $\rho_{D^{-1}(L+U)}$, verifica

$$\rho_{(\mathbb{I}-D^{-1}L)^{-1}D^{-1}U} = \rho_{N^{-1}P} < \rho_{D^{-1}(L+U)} < 1\,.$$

Cautela: si tenga conto che questo riferimento bibliografico chiama L e U le matrici che noi qui indichiamo con $D^{-1}L$ e $D^{-1}U$).

Per inciso, osserviamo che la matrice $L+U$ ha elementi diagonali tutti nulli e quindi certamente non è a predominanza diagonale stretta; la matrice $D^{-1}(L + U)$ si ottiene dalla precedente moltiplicando ciascuna riga per il corrispondente numero sulla diagonale di D^{-1}, e quindi neanche essa

è a predominanza diagonale stretta. Pertanto non si applica neppure la condizione sufficiente di convergenza per il metodo di rilassamento di Gauss–Seidel.

La seconda osservazione è che le iterazioni così ottenute *non* sono le stesse del metodo di Gauss–Seidel discusso precedentemente. Però, visto che il metodo converge, ossia il raggio spettrale $\rho_{D^{-1}(L+U)}$ è minore di 1, allora il ragionamento che porta all'identità (6.9.1) prova che la differenza fra le iterate k-sime del metodo di Jacobi ed il punto fisso $\boldsymbol{b}$ tende a zero come $\rho^k_{D^{-1}(L+U)}$. Ma questo ragionamento provava che la differenza fra le iterate $k-$sime del metodo di Gauss–Seidel ed il punto fisso $\boldsymbol{b}$ tende a zero come $\rho^k_{(\mathbb{I}-D^{-1}L)^{-1}D^{-1}U} = \rho^k_{N^{-1}P}$, che, come visto sopra, è più piccolo. Quindi il metodo di Jacobi converge allo stesso punto fisso, ma più lentamente. Negli esempi comunque noi non diamo preminenza, in questo libro, al metodo di iterazione di Jacobi.

6.11. Rilassamento di Southwell applicato al sistema lineare della radiosità

6.11.1. Rilassamento di Southwell. Abbiamo visto nella sezione precedente che in ogni ciclo di iterazione, da ognuna delle n equazioni nell'ordine $j = 1, \ldots, n$, il metodo di Gauss–Seidel calcola la componente j di una approssimazione del valore di radiosità di ogni singolo elemento. Ora presentiamo il metodo di Southwell, che procede in ordine diverso: anch'esso calcola le varie componenti della nuova approssimazione risolvendo una equazione lineare alla volta, ma la scelta di j (cioè di quale equazione usare) è fatta in modo da rendere massima l'efficienza.
Quando si studiano scambi di radiosità questi paragoni sono più significativi se compiuti su valori di energia (emessi da un singolo elemento) piuttosto che su valori di radiosità (energia per unità di area). Riscriviamo l'equazione della radiosità in termini dei valori di energia per elemento e definiamo nuovi vettori $\boldsymbol{\beta}$ e $\boldsymbol{\varepsilon}$ come

$$\beta_i = A_i b_i \qquad \text{(energia totale che lascia l'elemento } i\text{, emessa internamente o riflessa)}, \tag{6.11.1a}$$

$$\varepsilon_i = A_i e_i \qquad \text{(energia emessa internamente dall'elemento } i\text{)}. \tag{6.11.1b}$$

Quindi riscriviamo l'equazione della radiosità, trasformando la matrice M del sistema lineare definita in (6.2.5) in una nuova matrice K, che rappresenta il sistema lineare dell'energia:

$$K\boldsymbol{\beta} = \boldsymbol{\varepsilon}, \tag{6.11.2}$$

con

$$K_{ij} = \frac{A_i}{A_j}\, M_{ij} \tag{6.11.3}$$

Grazie alla relazione di reversibilità (6.2.2), $A_i F_{ij} = A_j F_{ji}$, nel calcolo degli elementi della matrice K si possono far apparire i termini F_{ji} invece che gli F_{ij} che abbiamo usato nella matrice M. Osservando che

$$K_{ij} = \frac{A_i}{A_j}\, M_{ij} = \frac{A_i}{A_j}\, (\delta_{ij} - \rho_i F_{ij}),$$

per $i \neq j$ otteniamo, grazie alla relazione di reversibilità,

$$K_{ij} = -\rho_i F_{ji}, \tag{6.11.4}$$

mentre, per $i = j$, abbiamo $K_{ii} = 1 - \rho_i F_{ii}$. Pertanto la matrice K è assomiglia alla trasposta della matrice M: in effetti, si ha $K_{ii} = M_{ii}$, e per $j \neq i$ si ha $K_{ij} = -\rho_i F_{ji}$ e $M_{ij} = -\rho_j F_{ji}$, quindi se tutti i coefficienti di riflessione ρ_i sono uguali le due matrici sono trasposta l'una dell'altra. Si noti che

$$F_{ii} = 0 \longrightarrow K_{ii} = 1, \tag{6.11.5}$$

ed inoltre, per ogni i, j, se $\rho_i = \rho_j$ si ottiene

$$K_{ij} = \delta_{ij} - \rho_i F_{ji} = \delta_{ji} - \rho_i F_{ji} = M^T_{ij} \tag{6.11.6}$$

(qui indichiamo con M^T la matrice trasposta di M). Il sistema lineare (6.2.5) della radiosità si trasforma quindi nel seguente sistema lineare, che chiamiamo il *sistema della potenza:*

$$\beta_i = \varepsilon_i + \rho_i \sum_{j=1}^{n} \beta_j F_{ji}. \tag{6.11.7}$$

La matrice K è a predominanza diagonale stretta sulle colonne: per ogni colonna $j = 1, \ldots, n$, la somma dei termini non sulla diagonale verifica $\sum_{i:\, i\neq j} K_{ij} < K_{jj}$. In effetti,

$$\sum_{i:\, i\neq j} |K_{ji}| = \sum_{i:\, i\neq j} \rho_j F_{ij} \leqslant \sum_{i:\, i\neq j} F_{ij} = \Big(\sum_{i=1}^{n} F_{ij}\Big) - F_{jj}$$

$$= 1 - F_{jj} \leqslant 1 - \rho_j F_{jj} = K_{jj}\,.$$

Anche l'algoritmo di Southwell calcola, in modo simile al metodo di Jacobi, ma diversamente da Gauss–Seidel, una serie di approssimazioni $\boldsymbol{\beta}^{(k)}$ della soluzione $\boldsymbol{\beta}$. In vista di (6.11.2), è naturale definire il vettore degli errori, o vettore dei resti (che ci dà la precisione dell'approssimazione) come

$$\boldsymbol{\sigma}^{(k)} = \boldsymbol{\varepsilon} - K\boldsymbol{\beta}^{(k)} \tag{6.11.8}$$

Il sistema lineare (6.11.2) si può riscrivere nella forma

$$K_{ii}\beta_i = \epsilon_i - \sum_{j=1,\ldots,n;\ j\neq i} K_{ij}\beta_j \qquad \text{i=1,}\ldots,\text{n.}$$

Questo tenere un solo elemento a primo membro è esattamente il modo di riscrivere il sistema lineare che porta all'iterazione di Jacobi (si riveda l'identità (6.7.2)). Pertanto il procedimento iterativo diventa

$$\beta_i^{(k+1)} = \frac{1}{K_{ii}} \Big(\epsilon_i - \sum_{j=1,\ldots,n;\ j\neq i} K_{ij}\beta_j^{(k)}\Big) \qquad \text{i=1,}\ldots,\text{n.} \tag{6.11.9}$$

Il fatto che $\boldsymbol{\beta}^{(k)}$ converga alla soluzione esatta $\boldsymbol{\beta}$ equivale al fatto che il vettore resto $\boldsymbol{\sigma}^{(k)}$ tenda a 0. Notiamo che

$$(K\boldsymbol{\beta}^{(k)})_i = \sum_j K_{ij}\beta_j^{(k)} = \sum_j \frac{A_i}{A_j}\, M_{ij}\beta_j^{(k)} = A_i \sum_j M_{ij} b_j^{(k)} = A_i (M\boldsymbol{b}^{(k)})_i\,.$$

Ora dalla definizione di $\boldsymbol{\varepsilon}$ e $\boldsymbol{\beta}$ e da (6.11.3) segue che la $i-$sima componente del vettore resto alla $k-$sima iterazione è data da

$$\sigma_i^{(k)} = A_i \Big(e_i - (Mb^{(k)})_i\Big) \tag{6.11.10}$$

La nuova approssimazione $\boldsymbol{\beta}^{(k+1)}$ data ad ogni passo dall'algoritmo di Southwell viene ottenuta con lo stesso schema di Jacobi, ma non applicando l'intera matrice di iterazione, bensì una sua sola riga, cioè modificando una singola componente del vettore $\boldsymbol{\beta}^{(k)}$, in modo tale che la corrispondente componente di $\boldsymbol{\sigma}^{(k)}$ diventi nulla: resto 0 per quella componente. La componente risolta ad una data iterazione si dice *rilassata*.

L'equazione (6.11.10) mostra che le norme dei vettori resto del metodo di Southwell sono maggiorate da quelle del metodo di Jacobi (la cui iterazione è identica) moltiplicate per il valore massimo delle aree degli elementi: quindi, se gli elementi hanno aree limitate, la convergenza del metodo di Jacobi implica quella del metodo di Southwell. Nella prossima Sottosezione 6.11.2 dimostreremo la convergenza in generale, con una dimostrazione più elegante.

A tal fine, vediamo cosa comporta il fatto che ad ogni iterazione l'algoritmo si limiti a risolvere (ovvero a rilassare) solo una delle n equazioni del sistema. Supponiamo che alla iterazione $k+1$ venga risolta la equazione numero i (ossia la riga i del sistema lineare): questo vuol dire, per (6.11.9),

che si trova un approssimante $\boldsymbol{\beta}^{(k+1)}$ tale che $\beta_i^{(k+1)} = \frac{1}{K_{ii}} \left(\epsilon_i - \sum_{j=1,\dots,n;\, j\neq i} K_{ij}\beta_j^{(k)}\right)$; invece; per le altre componenti,

$$\beta_j^{(k+1)} = \beta_j^{(k)} \quad \forall j \neq i, \tag{6.11.11}$$

Ma allora

$$\beta_i^{(k+1)} = \frac{1}{K_{ii}} \left(\varepsilon_i - \sum_{j=1,\dots,n;\, j\neq i} K_{ij}\beta_j^{(k+1)}\right) \tag{6.11.12}$$

ossia $(K\boldsymbol{\beta}^{(k+1)})_i = \epsilon_i$, ossia, da (6.11.8),

$$\sigma_i^{(k+1)} = \varepsilon_i - (K\boldsymbol{\beta}^{(k+1)})_i = 0. \tag{6.11.13}$$

Nel metodo di Southwell, l'equazione da rilassare ad ogni iterazione viene scelta come quella che alla iterazione precedente dava luogo alla componente del vettore resto maggiore delle altre (ossia, diciamo così, viene rilassata la componente con l'errore più grande).

NOTA 6.11.1. Sottolineamo che il metodo di Southwell si basa su una procedura differente da quella di Jacobi. Esso rilassa solo una riga per ogni iterazione. Fino a questo punto il procedimento non fa differenza rispetto a quello di Jacobi. Però il metodo di Southwell rilassa ad ogni iterazione la componente di resto massimo, e per questo calcola il vettore resto $\boldsymbol{\sigma}^{(k+1)}$ e pone uguale a zero la sua componente i, ma rilassa solo quella e non anche le altre. Quindi cambia l'ordine delle operazioni, in modo da ottimizzare il procedimento. □

Quindi la differenza dal metodo di Jacobi (ed in qualche modo anche di Gauss–Seidel) è che ad ogni iterazione questo aggiorna in maniera progressiva tutte le componenti del vettore della soluzione approssimata alla iterazione precedente, mentre il metodo di Southwell sceglie nel vettore resto $\boldsymbol{\sigma}^{(k)}$ la coordinata i alla quale corrisponde la componente più grande e rilassa la componente i-sima $\beta_i^{(k)}$ del vettore approssimante in modo da annullare la corrispondente componente del resto, $\sigma_i^{(k)}$.
Poiché alla iterazione $k+1$ modifichiamo soltanto la componente i, si ha (6.11.11), mentre per la componente i abbiamo (6.11.12). Ossia, sempre da (6.11.13),

$$\beta_i^{(k+1)} = \frac{1}{K_{ii}} \left(\varepsilon_i - \sum_{j=1}^{n} K_{ij}\beta_j^{(k)} + K_{ii}\beta_i^{(k)}\right) = \beta_i^{(k)} + \frac{\sigma_i^{(k)}}{K_{ii}}\,. \tag{6.11.14}$$

Da (6.11.8) segue che

$$\boldsymbol{\sigma}^{(k+1)} - \boldsymbol{\sigma}^{(k)} = -K(\boldsymbol{\beta}^{(k+1)} - \boldsymbol{\beta}^{(k)}). \tag{6.11.15}$$

Da (6.11.15), grazie a (6.11.11) e (6.11.14), nella moltiplicazione riga per colonna all'ultimo membro entra in gioco solo la colonna $i-$esima della matrice K (perche solo la componente $i-$esima del vettore $\boldsymbol{\beta}^{(k+1)} - \boldsymbol{\beta}^{(k)}$ è non nulla), e ne segue che i resti si ridistribuiscono in questo modo:

$$\sigma_j^{(k+1)} = \sigma_j^{(k)} - \frac{K_{ji}}{K_{ii}}\,\sigma_i^{(k)}\,. \tag{6.11.16}$$

Ripetiamo quanto sopra in un breve sommario. L'algoritmo di Southwell calcola per iterazione una soluzione approssimata $\boldsymbol{\beta}^{(k)}$ e stima l'errore di approssimazione, cioè il vettore resto $\boldsymbol{\sigma}^{(k)}$. A questo punto l'algoritmo trova quale è la componente del vettore resto più grande in valore assoluto ed aggiorna quella componente in maniera che il resto corrispondente diventi nullo. In questo modo cambiano anche le altre componenti; l'algoritmo ripete l'iterazione su quella diventata di grandezza maggiore, e così via.
Durante ogni iterazione viene usata una sola colonna della matrice K, quindi occorre calcolare una sola riga della matrice dei fattori di forma (in effetti, grazie a (6.11.4), la matrice K, al di fuori della diagonale, è legata alla trasposta della matrice dei fattori di forma: ogni sua colonna è un multiplo della corrispondente riga della matrice dei fattori di forma). La scelta del vettore iniziale $\boldsymbol{\beta}^{(0)}$ è

arbitraria: infatti nella prossima Sottosezione dimostreremo che l'algoritmo converge per qualunque scelta iniziale.

6.11.2. Convergenza del metodo di Southwell. Consideriamo la norma del vettore resto. Dobbiamo dimostrare che essa converge a zero quando iteriamo il procedimento all'infinito. Al posto della norma Euclidea $\|\boldsymbol{\sigma}^{(k)}\|_2 := \left(\sum_{i=1}^n (\sigma_i^{(k)})^2\right)^{\frac{1}{2}}$, è equivalente usare la norma $\ell^1(\mathbb{C}^n)$ data da

$$\|\boldsymbol{\sigma}^{(k)}\|_1 := \sum_{i=1}^n |\sigma_i^{(k)}|,$$

perché

$$\|\boldsymbol{\sigma}^{(k)}\|_2 \leqslant \|\boldsymbol{\sigma}^{(k)}\|_1 \leqslant n^{\frac{1}{2}} \|\boldsymbol{\sigma}^{(k)}\|_2$$

grazie alla disuguaglianza di Cauchy-Schwartz. (Rammentiamo comunque di nuovo che in dimensione finita tutte le norma sono equivalenti). Quindi vogliamo provare il seguente enunciato.

TEOREMA 6.11.2. *Il metodo di Southwell, applicato ad una matrice a predominanza diagonale stretta, converge per ogni scelta dell'approssimazione iniziale:*

$$\lim_{k\to\infty} \|\boldsymbol{\sigma}^{(k)}\|_1 = \lim_{k\to\infty} \sum_{i=1}^n |\sigma_i^{(k)}| = 0.$$

DIMOSTRAZIONE. Da (6.11.16) abbiamo

$$\|\boldsymbol{\sigma}^{(k+1)}\|_1 = \sum_{j=1}^n \left| \sigma_j^{(k)} - \frac{K_{ji}}{K_{ii}} \sigma_i^{(k)} \right| . \tag{6.11.17}$$

Poiché stiamo supponendo che alla iterazione $k-$sima la componente più grande del vettore $\boldsymbol{\sigma}^{(k)}$ sia la $i-$esima, cioè che sia questa la componente che viene rilassata alla iterazione $k+1$, segue che $\sigma_i^{(k+1)} = 0$, in base a (6.11.8) o anche a (6.11.16). Inoltre, al lato destro il termine $i-$simo è nullo, e quindi in realtà la somma in (6.11.17) si può limitare ai soli indici j diversi da i: adotteremo quindi questo range di somma.
Ora segue dalla disuguaglianza triangolare che

$$\begin{aligned} \|\boldsymbol{\sigma}^{(k+1)}\|_1 &\leqslant \sum_{j\neq i} \left|\sigma_j^{(k)}\right| + \sum_{j\neq i} \left| \frac{K_{ji}}{K_{ii}} \sigma_i^{(k)} \right| \\ &= \|\boldsymbol{\sigma}^{(k)}\|_1 - |\sigma_i^{(k)}| + |\sigma_i^{(k)}| \sum_{j\neq i} \left| \frac{K_{ji}}{K_{ii}} \right| . \end{aligned} \tag{6.11.18}$$

Poiché la matrice K è a predominanza diagonale stretta sulle colonne, per ogni i, $1 \leqslant i \leqslant n$, si ha:

$$K_{ii} > \sum_{j\neq i} K_{ji} . \tag{6.11.19}$$

Per $i = 1, \ldots, , n$, poniamo

$$t := \max_{i=1,\ldots,n} \sum_{j\neq i} \left| \frac{K_{ji}}{K_{ii}} \right| .$$

Allora $0 < t < 1$ per (6.11.19), e la definizione di t equivale a per $1 \leqslant i \leqslant n$,

$$\sum_{j\neq i} \left| \frac{K_{ji}}{K_{ii}} \right| \leqslant t. \tag{6.11.20}$$

Pertanto segue da (6.11.18) che, per ogni $1 \leqslant i \leqslant n$,

$$\|\boldsymbol{\sigma}^{(k+1)}\|_1 \leqslant \|\boldsymbol{\sigma}^{(k)}\|_1 - (1-t)\,|\sigma_i^{(k)}|. \tag{6.11.21}$$

Per l'ipotesi del metodo di Southwell, all'iterazione $k-$sima la componente più grande del vettore resto è la $i-$esima. Perciò:

$$\|\boldsymbol{\sigma}^{(k)}\|_1 \leqslant n\,|\sigma_i^{(k)}|. \tag{6.11.22}$$

Quindi

$$\|\boldsymbol{\sigma}^{(k+1)}\|_1 \leqslant \left(1 - \frac{1-t}{n}\right)\,\|\boldsymbol{\sigma}^{(k)}\|_1. \tag{6.11.23}$$

Osserviamo che, se si pone

$$T := 1 - \frac{1-t}{n}\ ,$$

applicando iterativamente (6.11.23) si ottiene

$$\|\boldsymbol{\sigma}^{(k+1)}\|_1 \leqslant T^{k+1}\,\|\boldsymbol{\sigma}^{(0)}\|_1 \tag{6.11.24}$$

Ma $T < 1$ poiché $0 < t < 1$, e quindi $\lim_{k\to\infty} T_{k+1} = 0$. Ne segue che $\lim_{k\to\infty} \|\boldsymbol{\sigma}^{(k)}\|_1 = 0$, e perciò l'algoritmo converge per qualsiasi scelta delle condizioni iniziali. □

6.11.3. Interpretazione fisica del procedimento di Southwell: luce emessa nell'ambiente. È interessante interpretare i vari passi del metodo di Southwell quando esso viene applicato al problema della radiosità e vedere il significato fisico del rilassamento.
Per comodità, iniziamo il processo iterativo di Southwell assegnando al vettore approssimante valore zero (cioè tutte le componenti uguali a zero). In questo caso, in base a (13), il vettore iniziale dei resti è uguale al vettore dell'energia creata dall'elemento. La prima iterazione seleziona quindi la componente che corrisponde all'elemento che emette più luce propria. Questo primo passaggio porta ad una nuova stima dell'energia dell'elemento, calcolata in modo che il valore del resto diventi nullo per quell'elemento.
Assumendo per semplicità che tutti i fattori di forma da un elemento su sé stesso siano nulli, se indichiamo con i l'indice dell'elemento che viene rilassata durante l'iterazione $k+1$ (cioè quella con il resto più grande dopo l'iterazione k), la variazione del resto per ciascun elemento può essere espressa grazie a (19), (10) e (9) come

$$\sigma_j^{(k+1)} = \sigma_j^{(k)} - \frac{K_{ji}}{K_{ii}}\,\sigma_i^{(k)} = \sigma_j^{(k)} + \rho_j F_{ij} \sigma_i^{(k)}, \tag{6.11.25}$$

dove $j = 1, \ldots, n$, e i è l'indice dell'elemento di massima energia residua $\sigma_i^{(k)}$ dopo l'iterazione k. In altre parole, all'iterazione $k+1$, il resto $\sigma_i^{(k)}$ dopo l'iterazione k viene distribuito fra tutti gli altri elementi in funzione dell'appropriato fattore di forma e del coefficiente di riflettività. Poiché questi resti sono differenze di energia, ogni iterazione modella il trasferimento di energia da un elemento all'ambiente, dovuto ad un'ulteriore riflessione della luce sui vari elementi: all'iterazione $k+1$ l'elemento i di massimo resto, ossia di massima energia immagazzinata, si libera della propria energia distribuendola nell'ambiente in proporzione ai fattori di forma. Come conseguenza di queste iterazioni di irraggiamento di energia, ciascuna degli altri elementi riceve ed accumula energia luminosa, fino a che non viene il suo turno di emetterla: il suo turno arriva quando l'energia accumulata diventa così elevata da far sì che quel particolare elemento sia quello fra tutti con la massima quantità di energia luminosa immagazzinata.

6.12. Raffinamento progressivo

6.12.1. Emissione di energia nell'ambiente invece di assorbimento. La mole di calcoli dell'algoritmo di radiosità, se eseguito così come presentato fino a questo momento, è molto elevata. Sembra naturale domandarsi se è possibile procedere per approssimazioni successive, creando una prima immagine non accurata e migliorandola in seguito mediante un algoritmo incrementale. Con lo schema di calcolo che abbiamo stabilito per la radiosità questo procedimento non è realizzabile nel metodo di Southwell. Infatti i processi di iterazione richiedono che la nuova luminosità dell'elemento sia calcolata sommando tutte le energie entranti provenienti dagli altri elementi: non basta tener conto dell'emissione da parte di un solo elemento. Quindi, per progredire di un ordine di approssimazione nella stima della radiosità di un singolo elemento, deve essere eseguito l'intero ciclo di iterazione. Se ci sono n elementi, ogni ciclo di iterazione richiede la moltiplicazione di una matrice $n \times n$ con un vettore colonna di dimensione n, un procedimento che richiede n^2 moltiplicazioni, e quindi un tempo di esecuzione dell'ordine di $O(n^2)$. Inoltre, le necessità di spazio di memoria per lo storage sono ugualmente elevate. Infatti il sistema lineare della potenza (6.11.7) richiede una somma su j che coinvolge la matrice trasposta F_{ji}, la quale quindi va calcolata, per ogni riga i, al variare di j, ossia usando per ogni elemento j un corrispondente $z-$buffer semicubico centrato sull'elemento j e non lo stesso per tutta la riga (se non dovessimo usare la matrice trasposta ma quella originale, allora un unico $z-$buffer semicubico centrato sull'elemento i ci darebbe in un colpo solo tutti i fattori di forma F_{ij} della riga i). Pertanto i fattori di forma tra tutte gli elementi devono essere precalcolati all'inizio ed essere memorizzati prima dell'esecuzione: questo richiede sia un tempo sia uno spazio di memoria dell'ordine di $O(n^2)$.
Però, quest'ultimo problema dello spazio di memoria si può aggirare grazie alla equazione di reversibilità (6.2.2). Infatti, utilizzando questa equazione, Cohen, Chen, Wallace e Greenberg **[9]** hanno sviluppato un algoritmo per il raffinamento progressivo del calcolo della radiosità, senza la necessità di precalcolare e memorizzare tutti i fattori di forma: il sistema lineare viene risolto in modo progressivo, una riga per volta. Vediamo come l'equazione di reversibilità evita la memorizzazione di tutti i fattori di forma..
Prima riassumiamo: nel metodo iterativo fin qui presentato, il calcolo della radiosità dell'elemento $i-$esimo richiede il calcolo dei contributi di energia ricevuti da tutti gli altri elementi. Ogni termine della $i-$sima equazione del sistema della radiosità esprime l'effetto sulla radiosità dell'elemento i dovuta all'emissione dell'elemento j per tutti i valori di j da 1 a n; l'energia (o meglio potenza) contribuita all'elemento i dalla radiosità b_j dell'elemento j e poi riflessa nell'ambiente è

$$\rho_i b_j F_{ij} \tag{6.12.1}$$

I metodi iterativi calcolano l'incremento di radiosità di un elemento sommando queste energie luminosa che esso raccoglie da ciascun altro. Ora abbiamo compreso che tale procedimento è troppo oneroso.

La revisione del procedimento, che porta ad un raffinamento progressivo molto meno oneroso dell'immagine generata iterazione dopo iterazione, porta a reinterpretare il sistema lineare della potenza (6.11.7) considerando la potenza emanata da un elemento verso il resto dell'ambiente, piuttosto che quella ricevuta: ed in effetti, scambiando i ruoli di i e j rispetto a quanto fatto sopra nella Sezione 6.2, vediamo proprio che il contributo alla radiosità b_j dell'elemento j dovuto all'emissione di radiosità dell'elemento i e poi riflesso nell'ambiente è $\rho_j b_i F_{ji}$, analogamente a (6.12.1): questo ci ha portato al sistema lineare trasposto di quello della radiosità.
Sfortunatamente abbiamo visto che questo procedimento richiede una maggiore mole di calcoli e di occupazione di memoria di quello originale (6.2.5) della radiosità, perché, fissato l'elemento i, esso richiede la conoscenza del fattore di forma F_{ji} dall'elemento j all'elemento i, ed al variare di j ciascuno degli F_{ji} richiede un calcolo diverso, fatto su un diverso semicubo, quindi non con lo stesso $z-$buffer. Però qui possiamo utilizzare la relazione di reversibilità fra i fattori di forma (6.2.2) e

quindi scrivere il contributo alla radiosità b_j dovuto a b_i (cioè il contributo di emissione di energia dall'elemento j che consiste nella frazione di energia emessa dall'elemento i verso j e poi da questo riflessa indietro verso il resto dell'ambiente) come

$$\rho_j b_i F_{ij} \frac{A_i}{A_j} \ .$$

In effetti, ora basta scambiare gli indici i e j per verificare che, con questa rinormalizzazione, dal sistema della potenza (6.11.7) si ritorna al sistema della radiosità (6.2.3), ovvero, in forma compatta, (6.2.5). Ora però ne abbiamo reinterpretato il significato fisico: nel corso del $k-$simo ciclo di iterazione, la $i-$sima riga del sistema lineare della radiosità rappresenta l'emissione di energia lanciata nell'ambiente nel corso di quella iterazione dall'elemento i verso ciascuno degli altri elementi (incluso sé stesso se F_{ii} è non nullo).
Il calcolo degli addendi a secondo membro di quest'ultima equazione al variare di j richiede solo la conoscenza dei fattori di forma F_{ij} calcolati usando un unico semicubo centrato al centro dell'elemento $i-$esimo. Se i fattori di forma dall'elemento i possono essere calcolati velocemente (ad esempio utilizzando uno $z-$buffer hardware), allora il tempo necessario per questo calcolo è breve; inoltre lo spazio di memoria occupata da F_{ij} può essere rilasciata non appena la radiosità emanata dall'elemento i verso l'elemento j viene lanciata nell'ambiente (cioè appena calcolato l'addendo $j-$simo). In questo modo, ad ogni passo dobbiamo avere a disposizione spazio di memoria solo per un singolo semicubo e per i suoi fattori di forma. L'algoritmo, la cui convergenza è stata dimostrata nel Teorema 6.9.5, viene iterato fino a quando non si raggiunge una tolleranza prefissata.
Inizialmente la maggior parte dell'energia luminosa è concentrata in pochi elementi (le sorgenti luminose). Se il vettore di partenza $\boldsymbol{b}^{(0)}$ dell'iterazione si pone uguale a 0, allora, calcolando l'emissione di tutte le sorgenti simultaneamente, al primo passo otteniamo il primo iterato $\boldsymbol{b}^{(1)}$ uguale al vettore $\boldsymbol{e}$ dell'energia creata dalle sorgenti: ossia vediamo solo le sorgenti ed il resto è buio. Al passo successivo dobbiamo cominciare ad aggiungere gli addendi della somma che rappresentano, per ciascun elemento i, l'energia addizionale emessa dall'elemento i verso ciascuno degli altri e da questi riflessa indietro: in questa passo di iterazione, quindi, cominciano ad illuminarsi, oltre alle sorgenti di luce, anche gli elementi direttamente illuminati da esse. Comprendiamo allora la logica del metodo di Southwell: è chiaro che la convergenza diventa un pò più rapida se sviluppiamo il calcolo dapprima per la sorgente che ha più energia da emettere, e così via: questo significa che al primo passo ordiniamo gli elementi che sono sorgenti di luce in base alla luminosità (le altre, che non emettono luce propria, al primo passo non intervengono). Poi, via via, ad ogni ciclo di iterazione successivo scegliamo analogamente l'elemento da rilassare ordinando gli elementi in base alla quantità di energia immagazzinata perché ricevuta dagli altri che hanno precedentemente irradiato luce. In ciascuna fase di iterazione diventano visibili gli elementi che ricevono luce direttamente dagli oggetti che erano illuminati alla fase precedente. In altre parole, non procediamo secondo l'ordine progressivo degli elementi, per i da 1 a n, bensì scegliamo volta per volta l'elemento di massima energia immagazzinata. Questo significa procedere in maniera più efficiente utilizzando, invece del metodo di Gauss–Seidel, il metodo di Southwell.

6.12.2. Stima della radiosità residua da emettere. Nella Sottosezione 6.11.1 abbiamo studiato il vettore $\boldsymbol{\beta}^{(k)}$ dell'energia irradiata e le sue iterazioni $\boldsymbol{\beta}$ nella soluzione iterativa del sistema lineare (6.11.2) dello scambio di energia fra gli elementi. Supponiamo che l'elemento con maggiore energia dopo la $k-$sima iterazione (cioè quella che nel corso dell'iterazione $k+1$ rilascia la sua energia accumulata) sia l'elemento $i-$esimo, e per semplicità facciamo l'ipotesi che non rifletta luce su sé stesso, cioè che il fattore di forma F_{ii} sia zero. Allora, per (6.11.5), $K_{ii} = 1$, e quindi, nel ciclo di iterazione $k+1$, l'incremento del vettore $\boldsymbol{\beta}$ dell'energia totale irradiata è dato dall'equazione (6.11.14), che ora diventa

$$\beta_i^{(k+1)} = \beta_i^{(k)} + \sigma_i^{(k)} \tag{6.12.2}$$

e, per j diverso da i, dall'equazione (6.11.11):

$$\beta_j^{(k+1)} = \beta_j^{(k)} .$$

Corrispondentemente si ha $\sigma_i^{(k+1)} = 0$ (grazie a (6.11.16), perché stiamo rilassando l'elemento i) e, per j diverso da i,

$$\sigma_j^{(k+1)} = \sigma_j^{(k)} - K_{ji}\sigma_i^{(k)} = \sigma_j^{(k)} + \rho_j F_{ij}\sigma_i^{(k)} \tag{6.12.3}$$

sempre in base all'equazione (6.11.25).
L'energia totale irradiata dall'elemento j al passo di iterazione $k+1$ è quindi la stessa che al passo precedente (questo equivale a $\beta_j^{(k+1)} = \beta_j^{(k)}$), ma c'è una quantità $\sigma_j^{(k+1)} - \sigma_j^{(k)}$ di energia addizionale pervenuta all'elemento j durante l'iterazione $k+1$ in conseguenza del rilascio, in quella iterazione, dell'energia immagazzinata nell'elemento i fino alla iterazione precedente. Perciò è opportuno introdurre un nuovo vettore $\boldsymbol{\beta'}$ dell'energia totale già irradiata da ciascun elemento o in attesa di esserlo al successivo rilassamento:

$$\beta_j'^{(k)} = \beta_j^{(k)} + \sigma_j^{(k)} \qquad (j = 1, \ldots, n). \tag{6.12.4}$$

Questo vettore, un passo di iterazione dopo l'altro, dà una stima progressivamente più precisa dell'energia totale che verrà irradiata dagli elementi al termine del processo iterativo.
Segue ora da (6.12.4), (6.11.11) e (6.12.3) che, per ogni j diverso da i,

$$\beta_j'^{(k+1)} = \beta_j^{(k)} + \sigma_j^{(k+1)} = \beta_j'^{(k)} + \sigma_j^{(k+1)} - \sigma_j^{(k)} = \beta_j'^{(k)} + \rho_j F_{ij}\sigma_i^{(k)} . \tag{6.12.5}$$

Ora ritorniamo alle radiosità, che sono le energie irradiate divise per le aree dei rispettivi elementi. Vogliamo una stima iterativa $b_i'^{(k)}$ della radiosità finale b_m, per ogni elemento m. Essa è la somma della radiosità effettivamente rilasciata da quell'elemento fino al ciclo di iterazione k (che in base a (6.11.1) è data da $b_m^{(k)} = \beta_m^{(k)}/A_m$), e della radiosità immagazzinata ed ancora in attesa di essere rilasciata,

$$\Delta b_m^{(k)} = \frac{\sigma_m^{(k)}}{A_m} .$$

Pertanto

$$b_m'^{(k)} := \frac{\beta_m'^{(k)}}{A_m} = \frac{\beta_m^{(k)} + \sigma_m^{(k)}}{A_m} = b_m^{(k)} + \Delta b_m^{(k)} . \tag{6.12.6}$$

Ma poiché $b_m'^{(k)}$ è una stima via via più precisa della radiosità finale b_m, ora $\Delta b_m^{(k)}$ è una stima della quantità residua di radiosità che all'elemento i resta da irradiare dopo il passo k. A causa di questa radiosità residua, le immagini ottenute durante i primi passi di iterazione sono scure, ed a poco a poco diventano più chiare quando ci si avvicina all'immagine finale. Purtroppo, però, questa stima $b_m'^{(k)}$ tiene sì conto dell'energia residua $\Delta b_m^{(k)} = \sigma_m^{(k)}/A_m$ immagazzinata nell'elemento m al passo k di iterazione, ma in modo impreciso: al termine del ciclo iterativo il residuo di energia immagazzinata non contribuisce solo alla radiosità dell'elemento i che la immagazzinava, ma anche a quella di tutti gli altri, in seguito alle ulteriori emissioni e ricezioni ed in proporzione alle loro riflettività ed ai fattori di forma. Occorre quindi una stima più accurata che tenga conto di questa distribuzione.

6.12.3. Correzione ambientale della luminosità nel corso delle iterazioni. uando l'algoritmo di radiosità è formulato in termini dell'emissione di energia (o radiosità) dagli elementi, invece che della loro ricezione di energia dagli altri elementi, e l'approssimazione iniziale della soluzione è il vettore nullo, allora la sola illuminazione dopo il primo ciclo di iterazione è quella proveniente direttamente dalle sorgenti di luce. Nelle iterazioni successive si considerano una riflessione, due riflessioni e così via. Chiaramente, nelle immagini ottenute dalle prime fasi di iterazione manca l'effetto di molteplici riflessioni successive, e quindi queste immagini sono scure.
Vogliamo usare la stima (6.12.6) per determinare un termine di correzione ambientale da aggiungere

alla radiosità di tutte gli elementi al fine di compensare la bassa luminosità delle immagini iniziali: grazie ad esso si possono ottenere approssimazioni iterative già buone dell'immagine finale dopo non molte iterazioni, e quindi si velocizza considerevolmente il preocedimento di calcolo della radiosità. Come accennato prima, questo algoritmo è stato introdotto in [**9**].
Poiché questa correzione deve basarsi su una stima dell'effetto delle riflessioni successive, dobbiamo calcolarla stimando il coefficiente di riflettività media della scena, che si può definire come la media dei coefficienti di riflettività ρ_i degli elementi, pesata rispetto alla proporzione dell'area del singolo elemento rispetto all'area totale:

$$\rho_{\text{media}} = \frac{\sum_{m=1}^{n} \rho_m A_m}{\sum_{m=1}^{n} A_m} \tag{6.12.7}$$

Cosa si troverebbe se si cercasse una stima della luminosità ambientale residua già al passo iniziale di iterazione? A quel passo l'energia inviata all'ambiente per ogni elemento è data dal vettore $\boldsymbol{e}$ (la luce propria emessa dalle sorgenti). Allora, dopo la prima riflessione avremmo un contributo addizionale $\rho_{\text{media}}\|\boldsymbol{e}\|$, dopo la seconda un ulteriore contributo $\rho^2_{\text{media}}\|\boldsymbol{e}\|$, e così via: l'incremento di luminosità è quindi $R\boldsymbol{e}$, dove il numero R è il *fattore di interriflessione globale*

$$R = 1 + \rho_{\text{media}} + \rho^2_{\text{media}} + \rho^3_{\text{media}} + \cdots = \frac{1}{1 - \rho_{\text{media}}} \,. \tag{6.12.8}$$

Osserviamo che $0 < \rho_{\text{media}} < 1$, e quindi $R > 1$. L'incremento di luminosità dà quindi l'impressione falsa che l'energia non si conservi: ma in realtà qui stiamo misurando non l'energia ma il suo flusso temporale. Questi effetti di maggiore luminosità sono dovuti a riflessioni successive della luce, che avvengono in istanti successivi del tempo: in ogni intervallo temporale le lampade riemettono lo stesso numero di fotoni ed il flusso di energia totale rimane sempre quello iniziale, ma il fattore di correzione somma insieme gli istanti successivi e quindi amplifica l'energia erogata in una unità di tempo. Stiamo contando il contributo dello stesso fotone più volte, una per ogni riflessione. In questo modo peró possiamo ottenere la ridistribuzione dell'energia che ci dà una illuminazione di fondo più precisa, se applichiamo il ragionamento non alla prima iterazione ma via via a quelle successive, salvo poi rinormalizzare alla fine per imporre che l'energia sia la stessa di quella emessa dalle sorgenti. Vediamo come.
Una stima dell'energia totale H_k ancora non irradiata nell'ambiente dopo la $k-$sima iterazione (quindi dopo k interriflessioni) è la somma delle energie residue immagazzinate dagli elementi:

$$H_k = \sum_{m=1}^{n} \sigma_m^{(k)} \,. \tag{6.12.9}$$

Ma a noi serve la correzione ambientale delle radiosità, non delle energie. Perciò dobbiamo utilizzare il flusso per unità di area di questa energia totale non irradiata. L'area totale dell'ambiente è la somma A delle aree degli elementi:

$$A = \sum_{m=1}^{n} A_m \,.$$

Quindi il flusso per unità di area dell'energia totale non irradiata è

$$\Phi_k = \frac{RH_k}{A} \,. \tag{6.12.10}$$

Questo flusso ambientale, effetto globale di tutte le interriflessioni successive, colpisce tutte gli elementi, e le illumina in proporzione al loro coefficiente di riflettività: pertanto al termine della $k-$sima iterazione la stima della radiosità con incremento ambientale è data da

$$b_m'^{(k)} = b_m^{(k)} + \rho_m \, \Phi_k \,. \tag{6.12.11}$$

Al crescere di k, $\sigma_m^{(k)}$ tende a 0 per ogni $i = 1, \ldots, n$, perché il procedimento di Southwell converge ad un punto fisso, e quindi i resti tendono a zero. Quindi H_k tende a 0. Pertanto Φ_k tende a 0, e

l'effetto di correzione si attenua progressivamente col procedere delle iterazioni. Esso è intenso solo all'inizio, e permette di simulare con buona approssimazione quanto devono essere schiarite le immagini approssimate iniziali perché la loro luminosità totale (ma ovviamente non la loro distribuzione di luce) sia accettabile.

6.13. Accuratezza nel calcolo dei fattori di forma, uso del Ray Tracing per il loro calcolo, e mappa di occlusione

6.13.1. Accuratezza. L'utilizzo di uno $z-$buffer hardware rende il metodo basato sulla proiezione sul semicubo per il calcolo dei fattori di forma un algoritmo estremamente efficiente per quanto riguarda il tempo di calcolo. Non vale la stessa asserzione per quanto riguarda l'accuratezza. Ecco alcuni problemi:

- A causa del modo in cui funziona lo $z-$buffer, ogni pixel del semicubo viene associato ad un solo elemento, anche nel caso in cui in quel pixel siano parzialmente visibili due o più elementi. Questo può portare a fenomeni di aliasing per il calcolo dei fattori di forma.
- Inoltre, si assume che il punto centrale di un elemento sia rappresentativo della ubicazione dell'intero elemento quando da esso si guardano gli altri (e viceversa). Questo è vero se l'elemento è tutto vicino al centro dell'emicubo: se ciò non è vero, l'elemento può essere suddiviso in sottoelementi, ma questo si fa una volta per tutte, e quindi una volta sola: se il punto centrale non è adeguato, allora procediamo a suddividere, ma allora i sottoelementi distanti dal centro dell'emicubo sono inadeguati, ed occorre creare nuovi emicubi centrati su di essi.
- Infine osserviamo che, affinché l'intero procedimento basato sul semicubo dia un risultato corretto, gli elementi devono essere lontani fra di loro. La stima dei fattori di forma è approssimata per difetto per elementi vicini o adiacenti: sulle zone di contatto il calcolo dei fattori di forma precalcolati per i pixel del semicubo è sottostimato, perché è relativo al centro del pixel (dove i due elementi sono un po' più lontani), e non al punto di contatto che corrisponde invece di solito al bordo del pixel.

6.13.2. Calcolo dei fattori di forma via Ray Tracing. Il calcolo progressivo della radiosità è stato anche effettuato mediante l'impiego, nella valutazione dei fattori di forma, del Ray Tracing (non ricorsivo) al posto della proiezione sul semicubo [**50**]. Questo si fa nel modo seguente. Quando un elemento diffonde nell'ambiente la propria radiosità, da ogni vertice della scena vengono tracciati raggi verso quell'elemento. Si suddivide l'elemento in un numero finito di sottoelementi, ognuno dei quali è attraversato da un raggio uscente dal vertice. Se il raggio non viene interrotto lungo il suo cammino, allora il sottoelemento dell'elemento è visibile e viene quindi calcolato il fattore di forma infinitesimo tra il vertice ed il sottoelemento, o in modo analitico grazie alla formula (6.3.1), oppure in modo numerico o probabilistico. I procedimenti numerico e probabilistico sono attuati tracciando raggi a partire dal vertice, attraverso un viewport (o meglio un emisfero o un emicubo) suddiviso in pixel. I raggi sono centrati al centro dei vari pixel oppure distribuiti aleatoriamente (quale sia la migliore scelta di distribuzione di probabilità lo vedremo nella seconda parte del libro, Parte 2, dove presenteremo metodi probabilistici di illuminazione globale). Essi vengono tracciati e si conta quanti di essi colpiscono il sottoelemento: la frazione degli *hits*, ossia di quelli che lo colpiscono, misura la proporzione dell'angolo solido sotteso dal sottoelemento rispetto a quel vertice, e quindi il fattore di forma infinitesimo del sottoelemento rispetto al vertice. A questo punto il fattore di forma tra il vertice e l'intero elemento viene calcolato come somma dei fattori di forma infinitesimi dei sottoelementi visti dal vertice. In tal modo si ottengono direttamente i fattori di forma degli elementi visti dai vertici, il che è un vantaggio perché evita la fase di postprocessing nella quale le radiosità delle facce vengono trasformate in radiosità dei vertici [**10**]: qui si ottengono direttamente le radiosità dei vertici. È chiaro che, nel procedimento probabilistico, di fatto non occorre spezzare

l'elemento in sottoelementi, ma, se lo si fa, ciò corrisponde ad un campionamento stratificato, nel senso della successiva Sottosezione 8.3.2.

6.13.3. Mappa di occlusione. Si noti che il fattore di forma di un elemento visto da un punto è una misura dell'occlusione frapposta dall'elemento all'illuminazione del punto dovuta al resto dell'ambiente, ossia del suo livello di ombra. In questo senso, il calcolo del fattore di forma tramite Ray Tracing dei vari elementi rispetto ad un punto equivale alla mappa di occlusione, che ora definiamo e spieghiamo, a costo di una breve digressione. Consideriamo un punto, ad esempio appartenente ad una faccia piana di un elemento, ed il semispazio ad esso frontale. La frazione di angolo solido coperta dagli altri elementi, misurata attraverso il numero di hits (intersezioni) di raggi proiettori uscenti dal punto, misura l'intensità di ombra a quel punto come conseguenza dell'occlusione della luce dovuta agli altri elementi. Essa si può reinterpretare nel modo seguente. Supponiamo dapprima che il punto stia su un elemento da illuminare con luce puramente ambientale, non riflettente né diffusiva secondo il modello di Lambert. Immaginiamo che il semispazio frontale sia racchiuso in un emisfero uniformemente illuminato e non riflettente, che funge da sfondo luminoso, ossia da sorgente di luce diffusa. Emaniamo, dal punto in oggetto, un gran numero di raggi proiettori, approssimativamente angolarmente equidistribuiti nell'emisfero, e tracciamoli nella scena finché colpiscono un altro elemento o l'emisfero. Ciascun raggio proiettore che non viene bloccato dagli altri elementi colpisce l'emisfero, e, nel consueto meccanismo del Ray Tracing, ogni hit fornisce un contributo di energia luminosa. La somma di questi contributi è l'illuminazione del punto causata dallo sfondo luminoso: se si preferisce, è la luce ambientale che illumina il punto, detratta però di quella bloccata dagli altri elementi, ossia attenuata a causa dell'ombra che essi proiettano. In tal modo la mappa di occlusione definisce ombre, e quindi differisce dal modello banale della luce ambientale perché ha aspetti direzionali: tiene conto delle direzioni in cui il punto vede gli altri elementi.
Se invece si intende simulare l'illuminazione, e quindi il livello di ombra, di un punto che giace in un elemento diffusivo secondo Lambert, anche in tal caso è possibile calcolare l'occlusione della scena mediante Ray Tracing. In questo caso dobbiamo immaginare che l'emisfero uniformemente luminoso si scomponga in tante piccole sorgenti di luce puntuali che emettono luce isotropicamente, tutte alla stessa intensità. In tal caso, ogni raggio proiettore, in base al modello di Lambert, colpisce l'emisfero in una delle piccole sorgenti di luce, e fornisce un contributo di illuminazione proporzionale al coseno dell'angolo di deviazione dal polo Nord: infatti questo è esattamente l'angolo fra la direzione della piccola sorgente di luce verso cui punta il raggio proiettore ed il versore normale al punto di emanazione, che è nient'altro che il versore verticale, ossia diretto verso il polo Nord. Pertanto basta, in questo caso, emanare raggi proiettori non più equidistribuiti, ma pesati secondo il coseno dell'angolo di deviazione dal polo Nord. Questo meccanismo pesa diversamente le sorgenti puntiformi di cui idealmente si compone l'emisfero. Vengono pesate di più le aree dell'emisfero frontali rispetto al punto di emanazione, cioè quelle in un intorno del polo Nord (dove la deviazione è zero ed il coseno vale 1), le quali inducono una illuminazione più intensa proprio perché frontali, in perfetto accordo con la legge empirica di Lambert. Queste aree vengono campionate maggiormente perché un maggior numero di raggi vengono emessi in tali direzioni, e quindi contribuiscono di più all'illuminazione del punto di emissione, naturalmente se esse non sono occluse.
Infine, se la scena è illuminata non isotropicamente, l'illuminazione e l'ombra si possono ancora calcolare come sopra. Supponiamo che la distribuzione di luci della scena sia tale che, ad esempio, ci sia una direzione dalla quale proviene principalmente la luce, mentre nelle altre direzioni l'emisfero è relativamente buio. Allora i raggi devono essere distribuiti in maniera da privilegiare la direzione di provenienza della luce: la distribuzione angolare dei raggi deve essere più fitta in tale direzione, e deve decrescere come il coseno dell'angolo di deviazione. L'analisi precedente dell'occlusione Lambertiana precedente corrisponde al caso da cui la luce proviene principalmente sia quello frontale,

ossia il polo Nord. Questo modello è ideale per simulare le ombre in una scena data da una giornata nuvolosa con il cielo coperto ed il Sole dietro lo strato di nubi in un dato punto del cielo.

Approfittiamo di queste osservazioni per riformularle in un senso che diventerà cruciale quando, nel terzo volume di quest'opera, lo riconsidereremo nell'ambito dell'illuminazione globale. Abbiamo già osservato che il calcolo dei fattori di forma, ovvero della mappa di occlusione, tramite Ray Tracing richiede, come tipico del Ray Tracing, che vengano emessi molti raggi. C'è però un modo di evitare questo proliferare di raggi da emettere con una distribuzione che può essere anisotropa: invece di emettere molti raggi deterministicamente, ne emettiamo pochi ma stocasticamente, ovvero con una distribuzione di probabilità corrispondente alla distribuzione angolare voluta. Il senso di questa corrispondenza ed il modo in cui viene realizzata la distribuzione di probabilità verranno studiati nella Parte 2, nel Capitolo **??** sui metodi probabilistici per il calcolo di integrali e la loro applicazione all'illuminazione globale.

6.14. Riflettività bidirezionale e metodi multipass

6.14.1. Riflettività bidirezionale. Il metodo di radiosità sviluppato fino a questo momento tratta solo la riflessione diffusa. In tal modo, la radiosità uscente da un elemento è influenzata dalla radiosità totale e non dalle direzioni attraverso cui viene ricevuta energia. In [**23**] l'algoritmo della radiosità è stato esteso in modo da modellare anche la riflessione speculare. Invece di calcolare un singolo valore di radiosità per ogni elemento, viene calcolato un valore per ogni direzione. Più precisamente, si partiziona l'emisfero sopra l'elemento in un insieme finito di angoli solidi, ognuno dei quali individua una direzione per l'energia entrante o uscente (ovviamente, quindi, si limita l'attenzione ad un numero finito di direzioni: il problema viene discretizzato). A questo punto si calcola, in base ai modelli fisici o anche solo al modello euristico di Phong,, la percentuale della luce che entra dalla direzione entrante ed esce nella direzione di uscita. Pesando i contributi mediante questo coefficiente di riflettività bidirezionale ora possiamo calcolare la radiosità uscente in ogni direzione come somma dei contributi di radiosità entranti dalle varie direzioni (discretizzate, quindi in numero finito) ed emesse verso quella data direzione di uscita, più la radiosità creata in quel dato elemento ed emessa verso quella direzione di uscita. L'immagine infine viene resa interpolando le intensità determinate per ogni vertice tramite media pesata delle radiosità direzionali uscenti dall'elemento a cui il vertice appartiene, interpolate a partire da quelle delle direzioni discretizzate più vicine alla direzione dal vertice all'osservatore. Questo tipo di approccio però porta ad una mole di calcoli estremamente onerosa.

6.14.2. Metodi di rendering a due passi che combinano radiosità e Ray Tracing ricorsivo. L'algoritmo di radiosità si adatta bene alla riflessione diffusa, ma non a quella speculare, poiché l'intensità della riflessione speculare dipende dall'angolo con il quale l'osservatore guarda la superficie, ma questo angolo non viene considerato nell'algoritmo di radiosità, che non è direzionale. D'altra parte il procedimento di Ray Tracing è ideale per il calcolo della riflessione speculare, ma non per la diffusione globale dell'ambiente.

È quindi opportuno trovare il modo di combinare i due metodi in modo da sfruttare i vantaggi di entrambi. Purtroppo però non è sufficiente applicare entrambi i metodi e poi sommare i risultati, perché le componenti della luminosità dovute alla riflessione speculare contribuiscono anche alla illuminazione diffusa e viceversa.
In [**49**] è stato sviluppato un approccio a due passi che combina il metodo di radiosità (indipendente dall'angolo di visuale) con quello di Ray Tracing, che invece ne dipende. Il primo passo, che non dipende dalla direzione di osservazione, consiste di un metodo ampliato di radiosità che tiene conto anche della riflessione speculare, perché amplia in maniera virtuale la scena, come se gli elementi riflettenti fossero finestre su un mondo speculare. I fattori di forma da un elemento generico ad uno di questi elementi virtuali riflessi dall'altro lato di un elemento speculare rappresentano il contributo

alla diffusione globale dato dalla riflessione speculare dello specchio. Purtroppo, se le superficie speculari sono ampie, questa fase aumenta quasi di un fattore due i fattori di forma da calcolare, e quindi rende quattro volte più lento il tempo di calcolo.
Nel secondo passo, dipendente dal punto di visuale, si considera ogni punto visibile di ogni elemento (compresi quelli virtuali). Del raggio proiettore proveniente a questo punto dall'osservatore si calcola la direzione di riflessione speculare, e si costruisce, anziché il raggio riflesso del Ray Tracing, una piramide di riflessione, cioè uscente in tale direzione e di apertura piccola, perché limitata all'angolo solido che include solo le deviazioni dalla direzione di riflessione speculare per le quali l'intensità riflessa è sufficientemente elevata (per determinare quanto elevata sia l'intensità in ciascuna direzione uscente si calcola il coefficiente di riflettività bidirezionale accennato nella precedente Sottosezione 6.14.1. Sul piano perpendicolare all'asse della piramide (cioè alla direzione di riflessione speculare) si introduce un piccolo rettangolo che viene usato per eseguire uno $z-$buffer di piccole dimensioni: così si determina quali altri elementi siano visibili attraverso ciascun pixel, e di ciascuno di essi si considera l'illuminazione diffusa calcolata nella prima fase tramite la radiosità estesa (la radiosità era stata riportata ai vertici della scena, ed ora la si riporta ai punti osservati attraverso i centri dei pixel mediante interpolazione di Gouraud). Se c'è più di un elemento visibile attraverso la piramide, si calcola il contributo totale risultante tramite interpolazione, ossia la combinazione pesata in proporzione alle aree visibili): si osservi che questa fase è analoga al calcolo dei fattori di forma differenziali. Inoltre si tiene conto se la piramide attraverso il pixel vede altri elementi speculari: per ciascuno di essi ripetiamo il procedimento in maniera ricorsiva (per un numero prefissato di generazioni) emettendo un'altra piramide riflessa (esattamente come nel procedimento di generazione iterativa di raggi riflessi del Ray Tracing ricorsivo). Si totalizzano i contributi di illuminazione ottenuti, come spiegato prima, da ognuna delle piramidi così generate, ed in tal modo si ottiene il contributo della riflessione speculare al vertice della piramide iniziale. Anche il calcolo di questa fase, ovviamente, è piuttosto oneroso.

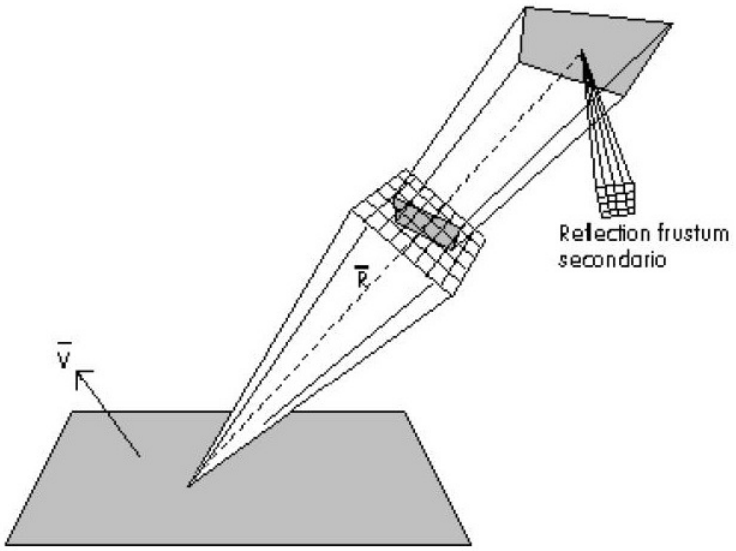


FIGURA 6.14.1. Piramidi di riflessione, primaria e secondaria

Nella Figura 6.14.1 viene mostrata una piramide di riflessione, ed un'altra, secondaria, generata dalla prima riflessione. Riassumiamo il procedimento. Quale sia per ogni pixel l'elemento speculare visibile si calcola col metodo di $z-$buffer; si utilizza l'interpolazione di Gouraud per calcolare per

quel pixel le intensità di riflessione diffusa del primo passo. Se l'elemento visibile ad un dato pixel di questo $z-$buffer è un elemento che dà luogo a riflessione speculare, allora si traccia il raggio dal vertice della piramide attraverso il centro del pixel e si determina l'angolo di riflessione speculare al punto in cui colpisce l'elemento: continuando iterativamente così si disegnano nuove piramidi in corrispondenza di ogni intersezione. Attraverso una media dei valori calcolati per ogni pixel dello $z-$buffer di ciascuna piramide si trova l'intensità della luce riflessa sull'elemento di partenza.
Questo metodo riesce a combinare la radiosità col ray tracing ricorsivo, ma porta ad una proliferazione dei fattori di forma, il cui calcolo è lento. Abbiamo già osservato che, se tutti gli elementi riflettono luce anche specularmente, contando anche gli elementi virtuali il loro numero si raddoppia ed il numero dei fattori di forma si quadruplica. Una variante più efficiente, introdotta in [**43**], calcola fattori di forma estesi, i quali sono in grado di tener conto di un numero arbitrario di riflessioni speculari o rifrazioni. Questi fattori di forma estesi si calcolano come sopra ma mediante Ray Tracing ricorsivo fin dal primo passo, invece che mediante semicubi e $z-$buffer come nel meccanismo usuale della radiosità, oppure mediante Ray Tracing non ricorsivo come nel caso della mappa di occlusione vista prima.

6.15. Esempi di calcolo della radiosità mediante simmetrie

6.15.1. Esempio: distribuzione della luce in una tenda piramidale. La scena consiste dell'interno di una tenda, cioè una stanza a forma di piramide equilatera, quindi con quattro facce identiche a forma di triangolo equilatero. La faccia di base emette luce con intensità 1 (possiamo pensare che ci sia un fuoco acceso al centro della base: in tal caso la luce viene emessa al centro, ma nella presente approssimazione, peraltro alquanto rudimentale, l'intero triangolo di base consiste di un unico elemento che emette luce, che indichiamo col numero 1). Le altre facce non emettono luce propria, solo riflessa. La base, cioè il pavimento, è di terra scura, e l'approssimiamo come una superficie non riflettente, cioè $\rho_1 = 0$. Le pareti sono grigio chiaro, e poniamo $\rho_i = \frac{1}{2}$ per $i = 2, 3, 4$. Quindi il vettore dell'energia propria emessa è $\boldsymbol{e} = (1, 0, 0, 0)$, mentre il vettore della riflettività è $\boldsymbol{\rho} = (0, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. Però in questo caso gli elementi sono piani, e quindi i fattori di forma diagonali, F_{ii}, sono tutti nulli. D'altra parte, la tenda è completamente simmetrica: ogni faccia copre, vista da ogni altra faccia, lo stesso angolo solido. Perciò, per $i \neq j$, i fattori di forma F_{ii} sono tutti uguali, cioè c'è un unico fattore di forma non nullo, che per semplicità chiamiamo F. (Si osservi che, poiché le aree di tutti gli elementi sono uguali, questo risultato conferma anche l'equazione di reciprocità (6.2.2). Ora, dall'equazione (6.2.1) della normalizzazione dell'angolo solido totale (o, se si preferisce, di conservazione dell'energia) $\sum_j F_{ij} = 1$, si ottiene $F = 1/3$. Pertanto il sistema lineare della radiosità diventa:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -\rho F & 1 & -\rho F & -\rho F \\ -\rho F & -\rho F & 1 & -\rho F \\ -\rho F & -\rho F & -\rho F & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Procediamo a risolvere il sistema con metodi di rilassamento. Tralasciamo il metodo di Jacobi (introdotto nella Sottosezione 6.7.3), le cui iterazioni si calcolano in maniera assolutamente elementare in quanto la matrice A da invertire (notazione come nella Nota 6.7.2) è diagonale (si rivedano le equazioni (6.7.7) e (6.7.10)). In effetti, abbiamo visto in (6.7.9) che la matrice di iterazione del metodo di Jacobi è $Q = -D^{-1}(L + U)$, e quindi il procedimento iterativo di Jacobi (6.7.10) è $\boldsymbol{b}^{(k+1)} = -D^{-1}(L + U)\boldsymbol{b}^{(k)} + D^{-1}\boldsymbol{e}$. Poiché la matrice D^{-1} si ottiene direttamente prendendo i reciproci dei suoi termini, che sono non nulli solo sulla diagonale, l'iterazione si calcola banalmente con una pedissequa applicazione delle operazioni matriciali riga per colonna. Non svolgeremo mai in questo libro questi calcoli tediosi, tranne che un paio di volte, negli Esercizi 6.16.4 e 6.16.9, nel

primo dei quali le matrici non sono assegnate numericamente bensì simbolicamente in termini dei coefficienti di riflessione e dei fattori di forma (ed in entrambi la dimensione delle matrici è molto bassa).

Il calcolo delle iterazioni di Jacobi è banale. si noti che qui $D = \mathbb{I} = D^{-1}$, e la matrice di iterazione di Jacobi è $-D^{-1}(L+U) = -(L+U)$. Abbiamo

$$L+U = \begin{pmatrix} 0 & 0 & 0 & 0 \\ -\frac{1}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} & 0 \end{pmatrix}.$$

Perciò, partendo con $\boldsymbol{b}^{(0)} = \mathbf{0}$, si ottiene

$$\boldsymbol{b}^{(1)} = D^{-1}\boldsymbol{e} = \boldsymbol{e} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix},$$

ed analogamente

$$\boldsymbol{b}^{(2)} = -(L+U)\boldsymbol{b}^{(1)} + \boldsymbol{e} = \begin{pmatrix} 0 \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \end{pmatrix},$$

$$\boldsymbol{b}^{(3} = -(L+U)\boldsymbol{b}^{(2)} + \boldsymbol{e} = \begin{pmatrix} 0 \\ \frac{2}{9} \\ \frac{2}{9} \\ \frac{2}{9} \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{2}{9} \\ \frac{2}{9} \\ \frac{2}{9} \end{pmatrix},$$

$$\boldsymbol{b}^{(4} = -(L+U)\boldsymbol{b}^{(3)} + \boldsymbol{e} = \begin{pmatrix} 0 \\ \frac{13}{54} \\ \frac{13}{54} \\ \frac{13}{54} \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{13}{54} \\ \frac{13}{54} \\ \frac{13}{54} \end{pmatrix},$$

$$\boldsymbol{b}^{(5} = -(L+U)\boldsymbol{b}^{(4)} + \boldsymbol{e} = \begin{pmatrix} 0 \\ \frac{20}{81} \\ \frac{20}{81} \\ \frac{20}{81} \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{20}{81} \\ \frac{20}{81} \\ \frac{20}{81} \end{pmatrix} = \begin{pmatrix} 1 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}.$$

L'iterazione successiva, per le prime due cifre decimali, dà lo stesso risultato: il risultato esatto è

$$\boldsymbol{b}^{(6)} == \begin{pmatrix} 1 \\ \frac{121}{336} \\ \frac{121}{336} \\ \frac{121}{336} \end{pmatrix} = \begin{pmatrix} 1 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}.$$

Quindi questo è ol punto fisso, nell'approssimazione di 1/100, ossia con le prime due cifre decimali esatte.

6.15.1.1. *Metodo di Gauss–Seidel.* Per poter applicare il metodo iterativo di Gauss–Seidel, formulato in (6.8.2), dobbiamo spezzare M come $M = N - P$, con N matrice triangolare inferiore (o meglio non superiore: la diagonale di N coincide con quella di M) e P triangolare (strettamente) superiore. Quindi:

$$N = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -\rho F & 1 & 0 & 0 \\ -\rho F & -\rho F & 1 & 0 \\ -\rho F & -\rho F & -\rho F & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \rho F & \rho F \\ 0 & 0 & 0 & \rho F \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

In questo metodo, il ciclo di iterazione (6.8.7) viene eseguito su tutte le componenti dei vettori, quindi su tutte le n equazioni del sistema. Il vettore $\boldsymbol{e}$ ora vale $(1, 0, 0, 0)$. Per semplicità di calcolo conviene partire con l'approssimante iniziale $\boldsymbol{b}^{(0)} = (0, 0, 0, 0)$. Si ottiene, da (6.8.7):

$$\boldsymbol{b}^{(k+1)} = N^{-1} P \boldsymbol{b}^{(k)} + N^{-1} \boldsymbol{e}. \tag{6.15.1}$$

Il lettore è invitato a verificare che una matrice triangolare inferiore a dimensione 4, con 1 sulla diagonale, cioè del tipo

$$J = \begin{pmatrix} 1 & 0 & 0 & 0 \\ x & 1 & 0 & 0 \\ y & z & 1 & 0 \\ u & v & w & 1 \end{pmatrix}$$

ha per inversa (esercizio!):

$$J^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -x & 1 & 0 & 0 \\ xz - y & -z & 1 & 0 \\ (y - xz)w + xv - u & zw - v & -w & 1 \end{pmatrix}$$

Perciò

$$N^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \rho F & 1 & 0 & 0 \\ \rho F + (\rho F)^2 & \rho F & 1 & 0 \\ \rho F + 2(\rho F)^2 + (\rho F)^3 & \rho F + (\rho F)^2 & \rho F & 1 \end{pmatrix}$$

Ora possiamo trovare gli approssimanti dati dalle prime iterazioni del metodo di Gauss–Seidel, cioè del rilassamento eseguito ricalcolando ad ogni ciclo iterativo tutte le n equazioni del sistema, ciascuna applicata ai dati provenienti dal ciclo precedente. Si trova, da (6.15.1):

$$\boldsymbol{b}^{(1)} = N^{-1} \boldsymbol{e}$$

$$\boldsymbol{b}^{(2)} = N^{-1} P \boldsymbol{b}^{(1)} + N^{-1} \boldsymbol{e} = N^{-1} P \boldsymbol{b}^{(1)} + \boldsymbol{b}^{(1)} = (N^{-1} P + \mathbb{I}) \boldsymbol{b}^{(1)}$$

$$\boldsymbol{b}^{(3)} = N^{-1} P \boldsymbol{b}^{(2)} + N^{-1} \boldsymbol{e} = N^{-1} P \boldsymbol{b}^{(2)} + \boldsymbol{b}^{(1)}$$

Poiché il vettore $\boldsymbol{e}$ è il primo vettore della base canonica, il risultato dell'applicare a $\boldsymbol{e}$ una matrice è il vettore dato dalla prima colonna della matrice. Pertanto:

$$\boldsymbol{b}^{(1)} = \begin{pmatrix} 1 \\ \rho F \\ \rho F + (\rho F)^2 \\ \rho F + 2(\rho F)^2 + (\rho F)^3 \end{pmatrix}$$

$$\boldsymbol{b}^{(2)} = (N^{-1}P + \mathbb{I}) \begin{pmatrix} 1 \\ \rho F \\ \rho F + (\rho F)^2 \\ \rho F + 2(\rho F)^2 + (\rho F)^3 \end{pmatrix}$$

$$= \begin{pmatrix} 1 \\ \rho F + 2(\rho F)^2 + 3(\rho F)^3 + (\rho F)^4 \\ \rho F + 2(\rho F)^2 + 4(\rho F)^3 + 4(\rho F)^4 + (\rho F)^5 \\ \rho F + 2(\rho F)^2 + 4(\rho F)^3 + 7(\rho F)^4 + 5(\rho F)^5 + (\rho F)^6 \end{pmatrix}$$

Nel nostro caso, $\rho = \frac{1}{2}$ e $F = 1/3$, quindi $\rho F = 1/6$. Se, per semplicità di calcolo, limitiamo la precisione alla prima cifra decimale, possiamo trascurare, in prima approssimazione, le potenze $(\rho F)^n$ con n maggiore di 2, perché $(\rho F)^2 = 0.02778$ e $(\rho F)^3 = 0.00077$ (si potrebbe trascurare già $(\rho F)^2$ se non fosse per il fatto che questo fattore compare moltiplicato per 2 nella terza e nella quarta componente del vettore $\boldsymbol{b}^{(2)}$). Quindi, l'approssimazione con la prima cifra decimale esatta di $\boldsymbol{b}^{(1)}$ è $\boldsymbol{b}^{(1)} = (1, 0.17, 0.19, 0.23)$, mentre $\boldsymbol{b}^{(2)} = (1, 0.24, 0,24, 0.25)$. Per $\boldsymbol{b}^{(3)}$ si ottiene, ponendo per semplicità $t = \rho F$:

$$\boldsymbol{b}^{(2)} = N^{-1}P \begin{pmatrix} 1 \\ t + 2t^2 + 3t^3 + t^4 \\ t + 2t^2 + 4t^3 + 4t^4 + t^5 \\ t + 2t^2 + 4t^3 + 7t^4 + 5t^5 + t^6 \end{pmatrix} + \begin{pmatrix} 1 \\ t \\ t + t^2 \\ t + t^2 + t^3 \end{pmatrix}$$

$$= \begin{pmatrix} 1 \\ t + 2t^2 + 4t^3 + 8t^4 + 11t^5 + 6t^6 + t^7 \\ t + 2t^2 + 4t^3 + 8t^4 + 15t^5 + 16t^6 + 7t^7 + t^8 \\ t + 2t^2 + 4t^3 + 8t^4 + 16t^5 + 26t^6 + 22t^7 + 8t^8 + t^9 \end{pmatrix} \approx \begin{pmatrix} 1 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}$$

Si osservi che le componenti dei vettori $\boldsymbol{b}^{(2)}$ e $\boldsymbol{b}^{(3)}$ sono termini polinomiali in t nei quali i monomi di grado più basso si stabilizzano, cioè mantengono gli stessi coefficienti (lasciamo al lettore, come esercizio, dimostrare per induzione che il coefficiente del monomio t^n in tutte le componenti delle approssimazioni successive si stabilizza sul valore $2n - 1$). Poiché alla precisione richiesta solo questi termini contano, per n maggiore di 2 le approssimazioni iterative $\boldsymbol{b}^{(n)}$ sono costanti: abbiamo raggiunto l'equilibrio, cioè la soluzione $\boldsymbol{b} = (1, 0.25, 0.25, 0.25)$. Si osservi che la prima componente, che corrisponde al pavimento della tenda piramidale, vale sempre 1 perché, pur non riflettendo luce, emette luce propria con potenza 1 per via della fiamma al suo centro. Osserviamo anche che all'equilibrio le tre pareti della tenda ricevono la stessa quantità di illuminazione: anche questo era previsto, per via della simmetria della scena fra le tre pareti. È evidente da questi calcoli che le iterazioni del metodo di Jacobi, in ciascuna delle quali si debbono ricalcolare tutte le equazioni del

sistema, sono laboriose. Presentiamo qui un notebook di Mathematica (software di manipolazione simbolica e numerica) che esegue i calcoli precedenti in maniera automatica:

```
P = {{0, 0, 0, 0}, {0, 0, t, t}, {0, 0, 0, t}, {0, 0, 0, 0}}
NN = {{1,0, 0, 0}, {-t, 1, 0, 0}, {-t, -t, 1, 0},
  {-t, -t, -t, 1}}
e = {1, 0, 0, 0}
Ninv = Inverse[NN]; TraditionalForm[Ninv]
b = {0, 0, 0, 0}
For[i=1, i<5, i++,
Expand[b = Ninv . P . b  + Ninv . e]; N[b] /. t -> 1/6]
```

Il lettore che non disponga di Mathematica può facilmente convertire al linguaggio C il precedente notebook e quelli che presenteremo in seguito; però poi, per gli integrali numerici necessari per i fattori di forma, deve usare le sofisticate librerie matematiche disponibili in C.

Ecco i risultati:

$$\boldsymbol{b}^{(1)} = (1, 0.167, 0.194, 0.233)$$
$$\boldsymbol{b}^{(2)} = (1, 0.238, 0.245, 0.249)$$
$$\boldsymbol{b}^{(3)} = (1, 0.249, 0.25, 0.25)$$
$$\boldsymbol{b}^{(4)} = (1, 0.25, 0.25, 0.25)$$

6.15.1.2. *Metodo di Southwell.* Trattiamo infine lo stesso esempio con il metodo di Southwell, introdotto nella Sezione 6.11. Il ciclo di iterazione dovrebbe ora riguardare la matrice dell'energia K, introdotta nella Sottosezione 6.11.1, invece di quella della radiosità M, ma poiché le aree degli elementi sono tutte uguali le due matrici coincidono. Quindi il ciclo iterativo rimane lo stesso (equazione (3) nella sezione Soluzione con metodi iterativi: rilassamento di Jacobi), ma viene eseguito ora su una sola equazione del sistema, quella con il resto maggiore. Il vettore dei resti alla $k-$sima iterazione è proporzionale a $\sigma^{(k)} = \boldsymbol{e} - M\boldsymbol{b}^{(k)}$, grazie a (6.11.10). Quando il vettore dei resti diventa nullo abbiamo raggiunto l'equilibrio. Per ciascuna iterazione calcoliamo il vettore dei resti, determiniamo l'indice (diciamo j) della sua componente più grande (in valore assoluto) e calcoliamo l'iterazione successiva svolgendo solo la $j-$sima equazione del sistema lineare della radiosità. Con il metodo di Gausss–Seidel ci volevano da tre a quattro iterazioni, risolvendo 4 equazioni per ogni iterazione. Quindi ora potremmo aspettarci di dover svolgere da 12 a 16 iterazioni. Ma poiché ogni volta trattiamo proprio la componente con l'errore maggiore, il metodo è più efficiente, e bastano solo sette iterazioni. Forniamo i risultati con un numero maggiore di cifre decimali rispetto a prima per evidenziare la progressione dell'iterazione. Si noti, per verifica di quanto appena detto, che in ogni iterazione cambia una sola componente (mai la prima, perché quella ha errore zero, in quanto la corrispondente elemento ha coefficiente di riflettività zero).

$$\boldsymbol{b}^{(1)} = (1, 0, 0, 0.226852)$$
$$\boldsymbol{b}^{(2)} = (1, 0, 0.238555, 0.226852)$$
$$\boldsymbol{b}^{(3)} = (1, 0.244234, 0.238555, 0.226852)$$
$$\boldsymbol{b}^{(4)} = (1, 0.244234, 0.238555, 0.248236)$$
$$\boldsymbol{b}^{(5)} = (1, 0.249596, 0.238555, 0.248236)$$
$$\boldsymbol{b}^{(6)} = (1, 0.249596, 0.238555, 0.249872)$$
$$\boldsymbol{b}^{(7)} = (1, 0.249596, 0.249957, 0.249872)$$

cioè, a due cifre decimali esatte (in realtà tre!), $\boldsymbol{b}^{(7)} = (1, 0.25, 0.25, 0.25)$. Ecco il corrispondente notebook di Mathematica. Si noti che tutte gli elementi hanno la stessa area A, quindi, a meno della moltiplicazione per A, il vettore resto alla $k-$sima iterazione coincide con $\boldsymbol{e} - M\boldsymbol{b}^{(k)}$.

```
t=1/6.
\noindent
\begin{verbatim}
P = {{0, 0, 0, 0}, {0, 0, t, t}, {0, 0, 0, t}, {0, 0, 0, 0}}
NN = {{1,0, 0, 0}, {-t, 1, 0, 0}, {-t, -t, 1, 0},
  {-t, -t, -t, 1}} .
M=NN-P.       * La matrice M e' la matrice della radiosita' *
D= {{1,0,0,0},{0,1,0,0,},{0,0,1,0},
{0,0,0,1}}.  *In questo esempio, la matrice D e' l'identita' *
Dinv=Inverse[D].    * In questo esempio, Dinv = D = identita' *
Q=Dinv.(M-D).        * La matrice Q e' la matrice di iterazione di Jacobi *
e = {1, 0, 0, 0}.
%Ninv=Inverse[NN]
b={0, 0, 0, 0}
maxr=1.
While[(maxr>0.001),
  r=N[e-M.b];
  maxr=Max[r[[1]], r[[2]], r[[3]], r[[4]], r[[5]],r[[6]]];
  For [i=1, i<7, i++,
    If [(r[[i]]==maxr),i0=i,Continue]
    ];
  b[[i0]]=N[Q[[i0]].b + Dinv[[i0]].e];
Print[N[b]] ]
```

6.15.2. Esempio: distribuzione della luce in una stanza cubica. La scena ora consiste dell'interno di una stanza a forma di cubo, quindi con sei facce quadrate identiche. La faccia alta ha una lampada che emette luce con intensità 1. Le altre facce non emettono luce propria, solo riflessa. Non ci sono finestre. La base, cioè il pavimento, è più scura degli altri muri: fissiamo $\rho_1 = \frac{1}{4}$. Le pareti ed il soffitto sono grigio chiaro, e poniamo $\rho_i = \frac{1}{2}$ per $i = 2, 3, 4, 5, 6$. Anche in questo caso ci limitiamo all'approssimazione rudimentale nella quale ciascuna parete forma un unico elemento (quindi non noteremo gradazioni di luce sulle pareti). Per maggiore generalità scriviamo ρ_+ invece di $\frac{1}{2}$ e ρ_- invece di $\frac{1}{4}$. Quindi il vettore dell'energia propria emessa è $\boldsymbol{e} = (0, 0, 0, 0, 0, 1)$, mentre il vettore della riflettività è $\boldsymbol{\rho} = (\frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$.
Anche in questo caso gli elementi sono piani, e quindi i fattori di forma diagonali, F_{ii}, sono tutti nulli. D'altra parte, la stanza è completamente simmetrica. Quindi ogni faccia copre, vista da ogni altra faccia ad essa laterale, lo stesso angolo solido. Inoltre ogni faccia copre lo stesso angolo solido (ma diverso da quello di prima) quando è vista dalla faccia diametralmente opposta. Perciò ci sono solo due fattori di forma non nulli: quello laterale, che per semplicità chiamiamo F_l, e quello frontale, F_f. In particolare, $F_{ij} = F_f$ per $(i, j) = (1, 6), (2, 4), (3, 5)$ ed i loro trasposti, e $F_{ij} = F_f$ per $(i, j) = (1, 2)$, $(1, 3), (1, 4)$, $(1, 5)$, $(6, 2)$, $(6, 3)$, $(6, 4)$, $(6, 5)$, $(2, 3)$, $(2, 5)$, $(3, 4)$, $(4, 5)$ e tutti i loro trasposti. Si noti che ciascuna faccia ha quattro facce laterali ed una frontale. Pertanto, dall'identità (6.2.1) della normalizzazione dell'angolo solido totale (o conservazione dell'energia) si ha $\sum_j F_{ij} = 1$, e quindi si ottiene $F_f + 4F_l = 1$. Il valore di $F_l f$ è stato calcolato nella Sezione 6.5, e vale $\frac{1}{5}$. Quindi anche $F_l = \frac{1}{5}$: ma sarebbe interessante calcolarlo direttamente. Il calcolo si basa sulla formula (6.5.1).
Qui non ci sono ostruzioni dentro la stanza, quindi H_{ij} vale costantemente 1, e le facce antistanti

sono parallele, quindi i due angoli θ_i e θ_j sono uguali. Chiameremo quest'angolo θ. L'integrale si può approssimare numericamente con il metodo illustrato nella Sottosezione 6.3.2, legato al metodo di $z-$buffer emicubico. Ma in questo caso l'integrale si può anche calcolare analiticamente come segue. Per simmetria, possiamo supporre $i = 1$ e $j = 6$, cioè che la faccia i sia quella inferiore e la faccia j quella superiore. Per l'invarianza sotto cambiamento di scala degli angoli solidi sottesi da facce antistanti, possiamo supporre che il cubo abbia lato 1, quindi che tutte le facce abbiano area 1. Possiamo anche supporre che il cubo abbia le facce parallele ai piani coordinati e giaccia nell'ottante positivo, con un vertice nell'origine. Allora la sua faccia inferiore ha coordinate $(x, y, 0)$ per x e y che variano fra 0 e 1. Analogamente, la faccia superiore ha coordinate $(x', y', 1)$ per x' e y' che variano fra 0 e 1.
Perciò, a meno del fattore π al denominatore, l'integrando ha la forma seguente. Chiamiamo $\boldsymbol{p}$ il vettore che congiunge un punto generico della faccia inferiore ad uno della faccia superiore: $\boldsymbol{p} = (x', y', 1) - (x, y, 0) = (x' - x, y' - y, 1)$. Osserviamo che la terza coordinata di $\boldsymbol{p}$ vale costantemente 1. Ora normalizziamo $\boldsymbol{p}$, ottenendo $\boldsymbol{q} = \boldsymbol{p}/\|\boldsymbol{p}\|$. Il valore di $\cos\theta$ è esattamente il prodotto scalare fra il versore $\boldsymbol{q}$ ed il versore perpendicolare alla due facce, cioè il versore dell'asse z, ovvero $(0, 0, 1)$. Pertanto $\cos\theta$ è nient'altro che la terza componente del versore $\boldsymbol{q}$: cioè $\cos\theta = 1/\|p\|$. D'altra parte, la distanza r fra $(x, y, 0)$ e $(x', y', 1)$ è proprio $\|p\|$. Quindi l'integrando è $1/(\pi\|p\|^4)$. (Questi risultati sono stati già osservati nella Sezione 6.3). Questo integrando è il reciproco di un polinomio di secondo grado nelle quattro variabili x, y, x', y', e quindi si può calcolare analiticamente in termini di funzioni arcoseno e logaritmo. Il calcolo è laborioso, ed analogo a quello svolto nell'Esempio della Sezione 6.5. Il risultato è $F_f = \frac{1}{5}$.
Chi vuole può riverificare il risultato analitico tramite approssimazione numerica. Ecco il listato di un notebook di Mathematica che svolge il calcolo:

```
pointdown[x, y] = {x, y, 0};
pointup[x', y'] = {x', y', 1};
displacement[x, y, x', y'] = pointup[x', y'] - pointdown[x, y];
NormSquare[x, y, x', y'] = 0;
For[i = 1, i < 4, i++,
    NormSquare[x, y, x', y'] = NormSquare[x, y, x', y']
     += displacement[x, y, x', y'][[i]]^2];
Norm[x, y, x', y'] = NormSquare[x, y, x', y']^(1/2);
Integrand[x, y, x', y'] = 1/ NormSquare[x, y, x', y']^2;
N[F = (1/Pi) Integrate[Integrand[x, y, x', y'], {x, 0, 1},
{y, 0, 1}, {x', 0, 1}, {y', 0, 1}] ]
```

Il fatto che $F_l = F_f$ è vero solo perché la stanza ha forma cubica. In seguito (Esempio nella Sottosezione 6.15.3) vedremo il caso di stanze di forma rettangolare, dove questo non accade. Nel seguito indicheremo con F il valore comune $F = F_l = F_f$. Quindi il sistema lineare della radiosità è il seguente:

$$\begin{pmatrix} 1 & -\rho_- F & -\rho_- F & -\rho_- F & -\rho_- F & -\rho_- F \\ -\rho_+ F & 1 & -\rho_+ F & -\rho_+ F & -\rho_+ F & -\rho_+ F \\ -\rho_+ F & -\rho_+ F & 1 & -\rho_+ F & -\rho_+ F & -\rho_+ F \\ -\rho_+ F & -\rho_+ F & -\rho_+ F & 1 & -\rho_+ F & -\rho_+ F \\ -\rho_+ F & -\rho_+ F & -\rho_+ F & -\rho_+ F & 1 & -\rho_+ F \\ -\rho_+ F & -\rho_+ F & -\rho_+ F & \rho_+ F & \rho_+ F & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \end{pmatrix}$$

cioè, con i valori che abbiamo scelto ($\rho_- = \frac{1}{4}$ e $\rho_+ = \frac{1}{2}$),

$$\begin{pmatrix} 1 & -0.05 & -0.05 & -0.05 & -0.05 & -0.05 \\ -0.1 & 1 & -0.1 & -0.1 & -0.1 & -0.1 \\ -0.1 & -0.1 & 1 & -0.1 & -0.1 & -0.1 \\ -0.1 & -0.1 & -0.1 & 1 & -0.1 & -0.1 \\ -0.1 & -0.1 & -0.1 & -0.1 & 1 & -0.1 \\ -0.1 & -0.1 & -0.1 & -0.1 & -0.1 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

La matrice a primo membro è la matrice della radiosità, che come sempre denotiamo con M. Svolgiamo il calcolo banale ma tedioso delle iterazioni di Jacobi; si noti che qui $D = \mathbb{I} = D^{-1}$, e la matrice di iterauine di Jacobi è

$$-(L+U) = -\begin{pmatrix} 0 & -\frac{1}{20} & -\frac{1}{20} & -\frac{1}{20} & -\frac{1}{20} & -\frac{1}{20} \\ -\frac{1}{10} & 0 & -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} \\ -\frac{1}{10} & -\frac{1}{10} & 0 & -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} \\ -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} & 0 & -\frac{1}{10} & -\frac{1}{10} \\ -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} & 0 & -\frac{1}{10} \\ -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{10} & 0 \end{pmatrix}$$

Perciò, partendo con $\boldsymbol{b}^{(0)} = \mathbf{0}$, si ottiene

$$\boldsymbol{b}^{(1)} = D^{-1}\boldsymbol{e} = \boldsymbol{e} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

ed analogamente

$$\boldsymbol{b}^{(2)} = (L+U)\boldsymbol{b}^{(1)} + \boldsymbol{e} = \begin{pmatrix} \frac{1}{20} \\ \frac{1}{10} \\ \frac{1}{10} \\ \frac{1}{10} \\ \frac{1}{10} \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{20} \\ \frac{1}{10} \\ \frac{1}{10} \\ \frac{1}{10} \\ \frac{1}{10} \\ 1 \end{pmatrix},$$

$$\boldsymbol{b}^{(3} = (L+U)\boldsymbol{b}^{(2)} + \boldsymbol{e} = \begin{pmatrix} \frac{14}{200} \\ \frac{27}{200} \\ \frac{27}{200} \\ \frac{27}{200} \\ \frac{27}{200} \\ \frac{209}{200} \end{pmatrix},$$

$$\boldsymbol{b}^{(4} = (L+U)\boldsymbol{b}^{(3)} + \boldsymbol{e} = \begin{pmatrix} \frac{317}{4000} \\ \frac{304}{2000} \\ \frac{304}{2000} \\ \frac{304}{2000} \\ \frac{304}{2000} \\ \frac{2122}{2000} \end{pmatrix},$$

$$\boldsymbol{b}^{(5} = (L+U)\boldsymbol{b}^{(4)} + \boldsymbol{e} = \begin{pmatrix} 0.083 \\ 0.164 \\ 0.164 \\ 0.164 \\ 0.164 \\ 1.069 \end{pmatrix},$$

$$\boldsymbol{b}^{(6} = (L+U)\boldsymbol{b}^{(5)} + \boldsymbol{e} = \begin{pmatrix} 0.086 \\ 0.164 \\ 0.164 \\ 0.164 \\ 0.164 \\ 1.074 \end{pmatrix},$$

$$\boldsymbol{b}^{(7)} == \begin{pmatrix} 0.087 \\ 0.165 \\ 0.165 \\ 0.165 \\ 0.166 \\ 1.075 \end{pmatrix},$$

$$\boldsymbol{b}^{(8)} == \begin{pmatrix} 0.087 \\ 0.166 \\ 0.166 \\ 0.166 \\ 0.166 \\ 1.075 \end{pmatrix},$$

L'iterazione successiva, per le prime due cifre decimali, dà lo stesso risultato: il risultato esatto per le prime tre cig=fre decimali è quindi $\boldsymbol{b}^{(8)} = (0.087, 0.165, 0.165, 0.165, 0.165, 1.075)$.

Applichiamo ora il metodo di Gauss–Seidel (Sezione 6.8), spezzando M come $M = N - P$, con N matrice triangolare inferiore (o meglio non superiore) e P triangolare (strettamente) superiore. Abbiamo:

$$N = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -0.1 & 1 & 0 & 0 & 0 & 0 \\ -0.1 & -0.1 & 1 & 0 & 0 & 0 \\ -0.1 & -0.1 & -0.1 & 1 & 0 & 0 \\ -0.1 & -0.1 & -0.1 & -0.1 & 1 & 0 \\ -0.1 & -0.1 & -0.1 & -0.1 & -0.1 & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} 0 & 0.05 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0 & 0 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0 & 0 & 0 & 0.1 & 0.1 & 0.1 \\ 0 & 0 & 0 & 0 & 0.1 & 0.1 \\ 0 & 0 & 0 & 0 & 0 & 0.1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Calcoliamo l'inversa di N. Possiamo usare il metodo di Gauss, o anche il metodo diretto usato prima in (6.15.2). Per semplicità ricaviamo il risultato dalla stesso notebook di Mathematica usato in quella circostanza per verifica. Si ottiene:

$$N^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0.1 & 1 & 0 & 0 & 0 & 0 \\ 0.11 & 0.1 & 1 & 0 & 0 & 0 \\ 0.121 & 0.11 & 0.1 & 1 & 0 & 0 \\ 0.1331 & 0.121 & 0.11 & 0.1 & 1 & 0 \\ 0.14641 & 0.1331 & 0.121 & 0.11 & 0.1 & 1 \end{pmatrix}$$

Lasciamo al lettore l'interessante esercizio di ricavare per induzione dall'andamento dei termini della matrice N^{-1} (espressi per le prime righe in (6.15.2)) il fatto che, quando tutti i coefficienti non nulli di N valgono 0.1, i coefficienti di N^{-1} sono del tipo $0.x$ con dato dalla successione dei coefficienti binomiali.

6.15.2.1. *Metodo di Gauss–Seidel.* Ora gli approssimanti alle varie iterazioni successive si trovano come nel caso della tenda piramidale dell'Esempio nella Sottosezione 6.15.1. Ecco i risultati con il metodo di Gauss–Seidel:

$$\boldsymbol{b}^{(1)} = (0.05, 0.105, 0.116, 0.127, 0.140, 1.053)$$
$$\boldsymbol{b}^{(2)} = (0.077, 0.151, 0.155, 0.158, 0.159, 1.070)$$
$$\boldsymbol{b}^{(3)} = (0.085, 0.163, 0.164, 0.164, 0.164, 1.074)$$
$$\boldsymbol{b}^{(4)} = (0.086, 0.165, 0.165, 0.166, 0.166, 1.075)$$
$$\boldsymbol{b}^{(5)} = (0.087, 0.166, 0.166, 0.166, 0.166, 1.075)$$
$$\boldsymbol{b}^{(6)} = (0.087, 0.166, 0.166, 0.166, 0.166, 1.075)$$

Alla quinta iterazione l'approssimante ha raggiunto il punto di equilibrio (limitatamente alle prime tre cifre decimali: se si richiede una precisione maggiore allora sono necessarie più iterazioni). Per completezza, riportiamo qui il notebook di Mathematica che abbiamo utilizzato:

```
t=0.1
P = {{0, t/2, t/2, t/2, t/2, t/2}, {0, 0, t, t, t, t},
{0, 0, 0, t, t, t}, {0, 0, 0, 0, t, t}, {0, 0, 0, 0, 0, t},
{0, 0, 0, 0, 0, 0}}
NN = {{1, 0, 0, 0, 0, 0}, {-t, 1, 0, 0, 0, 0},
{-t, -t, 1, 0, 0, 0}, {-t, -t, -t, 1,0, 0},
{-t, -t, -t, -t, 1, 0}, {-t,-t, -t, -t, -t, 1}}
e = {0, 0, 0, 0, 0, 1}
Ninv = Inverse[NN]
TraditionalForm[Ninv]
b = {0, 0, 0, 0, 0, 0}
For[i = 0, i < 7, i++,
N[Expand[b = Ninv. P . b  + Ninv . e]]; Print[b] ]
```

6.15.2.2. *Metodo di Southwell.* Infine, presentiamo i risultati ottenuti con il metodo di Southwell, secondo le linee esposte nella Sezione 6.11. La matrice dell'energia K nella Sottosezione 6.11.1, coincide anche in questo caso con M perché le aree di tutte gli elementi sono uguali; inoltre le aree valgono 1, quindi i vettori $\boldsymbol{\beta}$ della energia e $\boldsymbol{b}$ della radiosità coincidono. I risultati si ottengono rilassando una solo elemento (invece che tutti e sei) per ogni ciclo di iterazione: come visto nella Sottosezione 6.11.1, si tratta dell'elemento col valore massimo del vettore resto $\sigma^{(k)}$, il quale, grazie a (6.11.8) (o equivalentemente a (6.11.8)), vale $\boldsymbol{\sigma}^{(k)} = \boldsymbol{e} - M\boldsymbol{b}^{(k)}$.
Quindi mediamente potremmo aspettarci sei volte più iterazioni, ma, poiché ogni volta rilassiamo l'elemento con l'errore maggiore, questo metodo è più veloce: bastano 17 iterazioni per raggiungere l'equilibrio a meno di tre cifre decimali esatte. Non riportiamo i risultati per brevità: essi si possono trovare grazie al seguente notebook di Mathematica, del tutto analogo a quello già presentato nella Sottosezione 6.15.1.2:

```
t=0.1
P={{0,t/2,t/2,t/2,t/2,t/2},{0,0,t,t,t,t},{0,0,0,t,t,t},
{0,0,0,0,t,t},{0,0,0,0,0,t},{0,0,0,0,0,0}}.
NN={{1,0,0,0,0,0},{-t,1,0,0,0,0},{-t,-t,1,0,0,0},
{-t,-t,-t,1,0,0},{-t,-t,-t,-t,1,0},{-t,-t,-t,-t,-t,1}}.
M=NN-P.       * La matrice M e' la matrice della radiosita' *
D= {{1,0,0,0,0,0},{0,1,0,0,0,0},{0,0,1,0,0,0},
{0,0,0,1,0,0},{0,0,0,0,10t},{0,0,0,0,0,1}}.
*In questo esempio, la matrice D e' l'identita' *
Dinv=Inverse[D].
*In questo esempio,Dinv = D = identita' *
Q=Dinv.(M-D).
* La matrice Q e' la matrice di iterazione di Jacobi *
e={0,0,0,0,0,1}
%Ninv=Inverse[NN]
b={0,0,0,0,0,0}
maxr=1.
While[(maxr>0.001),
  r=N[e-M.b];
```

```
  maxr=Max[r[[1]], r[[2]], r[[3]], r[[4]], r[[5]],r[[6]]];
  For [i=1, i<7, i++,
    If [(r[[i]]==maxr),i0=i,Continue]
    ];
  b[[i0]]=N[Q[[i0]].b + Dinv[[i0]].e];
Print[N[b]] ]
```

Ecco le approssimazioni successive secondo Southwell ottenute tramite questo notebook di Mathematica:

```
{0,         0,        0,        0,        0,        1      }
{0,         0,        0,        0,        0.139755, 1      }
{0,         0,        0,        0.144806, 0.139755, 1      }
{0,         0,        0.148367, 0.144806, 0.139755, 1      }
{0,         0.150457, 0.148367, 0.144806, 0.139755, 1      }
{0.0791693, 0.150457, 0.148367, 0.144806, 0.139755, 1      }
{0.0791693, 0.150457, 0.148367, 0.144806, 0.139755, 1.06874}
{0.0791693, 0.150457, 0.148367, 0.144806, 0.163011, 1.06874}
{0.0791693, 0.150457, 0.148367, 0.163851, 0.163011, 1.06874}
{0.0791693, 0.150457, 0.16432,  0.163851, 0.163011, 1.06874}
{0.0791693, 0.164545, 0.16432,  0.163851, 0.163011, 1.06874}
{0.0862235, 0.164545, 0.16432,  0.163851, 0.163011, 1.06874}
{0.0862235, 0.164545, 0.16432,  0.163851, 0.163011, 1.07451}
{0.0862235, 0.164545, 0.16432,  0.163851, 0.165701, 1.07451}
{0.0862235, 0.164545, 0.16432,  0.165798, 0.165701, 1.07451}
{0.0862235, 0.164545, 0.165846, 0.165798, 0.165701, 1.07451}
{0.0862235, 0.165868, 0.165846, 0.165798, 0.165701, 1.07451}
{0.0868861, 0.165868, 0.165846, 0.165798, 0.165701, 1.07451}
```

6.15.3. Esempio: distribuzione della luce in una stanza rettangolare. Ora, invece di una stanza a forma di cubo, consideriamo una stanza rettangolare, di altezza 1, lunghezza L e profondità W. Le condizioni di illuminazione sono le stesse della stanza cubica vista precedentemente nell'Esempio della precedente Sottosezione 6.15.2: le trascriviamo qui per completezza. La faccia alta ha una lampada che emette luce con intensità 1. Le altre facce non emettono luce propria, solo riflessa. Non ci sono finestre. La base, cioè il pavimento, numerato con l'indice 1, è più scura degli altri muri: fissiamo $\rho_1 = \frac{1}{4}$. Le pareti ed il soffitto sono grigio chiaro, e quindi poniamo $\rho_i = \frac{1}{2}$ per $i = 2, \ldots, 6$. Anche in questo caso ci limitiamo all'approssimazione rudimentale nella quale ciascuna parete forma un unico elemento (quindi non avremo gradazioni di luce sulle pareti). Per comodità e maggiore generalità scriviamo ρ_+ invece di $\frac{1}{2}$ e ρ_- invece di $\frac{1}{4}$. Quindi il vettore dell'energia propria emessa è $\boldsymbol{e} = (0, 0, 0, 0, 0, 1)$, mentre il vettore della riflettività è $\boldsymbol{\rho} = (\frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$.
Anche in questo caso gli elementi sono piani, e quindi i fattori di forma diagonali, F_{ii}, sono tutti nulli. Ora peró la stanza non è completamente simmetrica. Ogni faccia copre lo stesso angolo solido solo quando è vista dalle due facce laterali opposte, cioè antistanti fra loro. Quindi ogni faccia ha un fattore di forma per la faccia antistante ed altri due per le due coppie di facce laterali. In tutto quindi ci sono 9 fattori di forma diversi. Ma la relazione di reciprocità della reversibilità dei percorsi ottici (6.2.2) ci permette di ricavare F_{ji} da F_{ij} : $F_{ij} = \frac{A_j}{A_i} F_{ji}$. Perciò dobbiamo solo calcolare, tramite integrazione analitica o numerica, tre fattori di forma per facce in disposizione antistante e tre per disposizione laterale. Numeriamo le facce cosi': 1 il pavimento, 6 il soffitto, 2 e 4 le parete di lunghezza L, 3 e 5 le pareti di lunghezza W. In realtà, l'equazione (6.2.1) della normalizzazione dell'angolo solido totale $\sum_j F_{ij} = 1$ ci fornisce il valore dell'ultimo fattore di forma F_{im} di ogni faccia

i, una volta noto F_{ij} per tutti i j diversi da m. Quindi dobbiamo calcolare in tutto sei fattori di forma. Come sempre, il calcolo analitico si basa sulla formula (6.5.1).

Cominciamo con le facce opposte. Di nuovo, non ci sono ostruzioni dentro la stanza, quindi H_{ij} vale costantemente 1, e le facce antistanti sono parallele, quindi $\theta_i = \theta_j$. Il calcolo è analogo a quello nella Sottosezione 6.15.2. Approssimiamo l'integrale numericamente, usando un notebook di Mathematica analogo a quello impiegato per la stanza cubica. Ad esempio, se L=2 e $W = 2.5$, troviamo i seguenti fattori di forma per le facce antistanti, a meno di due cifre decimali:

$$F_{16} = F_{61} = 0.45,$$
$$F_{24} = F_{42} = 0.13,$$
$$F_{35} = F_{53} = 0.08.$$

Invece, se $L = W = 2$ (stanza quadrata (ma non cubica!)), si trova:

$$F_{16} = F_{61} = 0.42$$
$$F_{24} = F_{42} = F_{35} = F_{53} = 0.12.$$

Ecco il notebook che abbiamo usato per il caso $i = 1$, $j = 6$ (gli altri si ottengono ruotando in ordine ciclico i valori di altezza, lunghezza e larghezza):

```
height=1
length=2
width =2

pointdown[x,y]={x,y,0};
pointup[x',y']={x',y',height};
displacement[x,y,x',y']=pointup[x',y']-pointdown[x,y];
NormSquare[x,y,x',y']=0;
For[i=1,i<4,i++,
    NormSquare[x,y,x',y']+=displacement[x,y,x',y'][[i]]^2];
Integrand[x,y,x',y']=height^2/ NormSquare[x,y,x',y']^2;
F=(1/(Pi * length * width)) NIntegrate[Integrand[x,y,x',y'],
{x,0,length},{y,0,width},{x',0,length},{y',0,width}]; Print[F]
```

Ora consideriamo il problema del calcolo dei fattori di forma per le facce laterali. Possiamo ancora supporre che la stanza abbia le facce parallele ai piani coordinati e giaccia nell'ottante positivo, con un vertice nell'origine. Allora la sua faccia 1 (il pavimento) ha coordinate $(x, y, 0)$ per x e y che variano fra 0 e L e fra 0 e W, rispettivamente. La faccia laterale per $x = 0$ ha coordinate $(0, y', z')$ per y' che varia fra 0 e L e z' fra 0 e 1. Se $\boldsymbol{p}$ è il vettore applicato (terminologia della geometria affine) da $(x, y, 0)$ a $(0, y', z')$ e $\boldsymbol{q}$ il versore $p/\|p\|$, allora nell'integrando $\cos\theta_1$ è il prodotto scalare fra $\boldsymbol{q}$ ed il versore $(0, 0, 1)$ normale al piano di base, mentre $\cos\theta_2$ è il prodotto scalare fra $\boldsymbol{q}$ ed il versore $(1, 0, 0)$ normale al piano $x = 0$ che contiene l'elemento 2. Quindi $\cos\theta_1 = z'/\|p\|$ e $\cos\theta_2 = x/\|p\|$. Inoltre $r = \|p\|$. (Questi risultati sono stati già osservati nella Sottosezione 6.3.1: si veda in particolare la Figura 6.3.5). L'integrale si può calcolare analiticamente, ma il calcolo è laborioso. Oppure lo si può approssimare numericamente con Mathematica; per $L = 2$ e $W = 2.5$ si trova, ad esempio: $F_{12} = 0.23 = \frac{A_2}{A_1} F_{21}$.
Però l'approssimazione numerica è affetta da un forte errore, perché l'integrando diverge quando r si avvicina a 0, cioè quando i due punti nell'integrando si avvicinano entrambi allo spigolo in cui il pavimento tocca la parete. Per evitare rischi di imprecisione, supponiamo che la stanza sia quadrata, cioè che sia $L = W$ (ma il valore comune non sia 1, perché allora la stanza è cubica ed abbiamo già trattato la stanza cubica). Allora $F_{12} = F_{13} = F_{14} = F_{15} = F_{62} = F_{63} = F_{64} = F_{65}$, e dall'equazione

(6.2.1) della conservazione dell'energia (o dell'angolo solido totale) $\sum_j F_{ij} = 1$ si ricava il loro valore comune, che chiameremo $F_{\text{pav-lato}}$:

$$F_{\text{pav-lato}} = \frac{1 - F_{16}}{4} = 0.145.$$

Analogamente, chiamiamo $F_{\text{lato-pav}}$ il valore comune $F_{21} = F_{26} = F_{41} = F_{46} = F_{31} = F_{36} = F_{51} = F_{56}$ e $F_{\text{lato-lato}}$ il valore comune $F_{32} = F_{52} = F_{34} = F_{54} = F_{23} = F_{25} = F_{43} = F_{45}$. Sappiamo che $F2_{21} = A1/A2F_{12} = 2F_{12}2$, quindi $F_{\text{lato-pav}}v = 2F_{\text{pav-lato}} = 0.29$. D'altra parte, sappiamo che $\sum_j F_{2j} = 1$, ma $F_{23} = F_{25} = F_{\text{lato-lato}}$, $F_{24} = 0.12$ e $F_{26} = F_{26} = F_{\text{lato-pav}} = 0.29$. Da qui segue che $F_{\text{lato-lato}} = (1 - 0.12 - 2 \cdot 0.29)/2 = 0.15$. Abbiamo determinato tutti i fattori di forma per la stanza quadrata. Riassumendo: ci sono 2 fattori di forma frontali:

$$\begin{aligned} &\text{da pavimento a soffitto (o viceversa)} = 0.42; \\ &\text{da parete a parete antistante} \qquad\quad = 0.12; \end{aligned}$$

Poi:

$$\begin{aligned} &\text{c'è un fattore di forma da parete a parete adiacente} && = 0.15; \\ &\text{c'è un fattore di forma da parete a pavimento (o soffitto)} && = 0.29; \\ &\text{c'è un fattore di forma da pavimento (o soffitto) a parete} && = 0.145. \end{aligned}$$

Con questi dati possiamo scrivere la matrice M del sistema della radiosità, e le matrici triangolare inferiore e superiore N e P della decomposizione $M = N - P$, come nelle equazioni (6.7.4), (6.7.5), (6.8.1) e (6.8.2). Ci limitiamo a scrivere N e P, ricordando che la riflettività del pavimento vale $\frac{1}{4}$ e quella degli altri elementi $\frac{1}{2}$.

$$N = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{2}\,0.29 & 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2}\,0.29 & -\frac{1}{2}\,0.15 & 1 & 0 & 0 & 0 \\ -\frac{1}{2}\,0.29 & -\frac{1}{2}\,0.12 & -\frac{1}{2}\,0.15 & 1 & 0 & 0 \\ -\frac{1}{2}\,0.29 & -\frac{1}{2}\,0.15 & -\frac{1}{2}\,0.12 & -\frac{1}{2}\,0.15 & 1 & 0 \\ -\frac{1}{2}\,0.42 & -\frac{1}{2}\,0.145 & -\frac{1}{2}\,0.145 & -\frac{1}{2}\,0.145 & -\frac{1}{2}\,0.145 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -0.145 & 1 & 0 & 0 & 0 & 0 \\ -0.145 & -0.075 & 1 & 0 & 0 & 0 \\ -0.145 & -0.06 & -0.07 & 1 & 0 & 0 \\ -0.145 & -0.075 & -0.06 & -0.07 & 1 & 0 \\ -0.21 & -0.0725 & -0.0725 & -0.0725 & -0.0725 & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} 0 & \frac{1}{2}\,0.145 & \frac{1}{2}\,0.145 & \frac{1}{2}\,0.145 & \frac{1}{2}\,0.145 & \frac{1}{4}\,0.42 \\ 0 & 0 & \frac{1}{2}\,0.15 & \frac{1}{2}\,0.12 & \frac{1}{2}\,0.15 & \frac{1}{4}\,0.29 \\ 0 & 0 & 0 & \frac{1}{2}\,0.15 & \frac{1}{2}\,0.12 & \frac{1}{4}\,0.29 \\ 0 & 0 & 0 & 0 & \frac{1}{2}\,0.15 & \frac{1}{4}\,0.29 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4}\,0.29 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0.0725 & 0.0725 & 0.0725 & 0.0725 & 0.1005 \\ 0 & 0 & 0.075 & 0.06 & 0.075 & 0.0725 \\ 0 & 0 & 0 & 0.075 & 0.065 & 0.0725 \\ 0 & 0 & 0 & 0 & 0.075 & 0.0725 \\ 0 & 0 & 0 & 0 & 0 & 0.0725 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

6.15.3.1. *Metodo di Jacobi.* Abbiamo, come al solito, $D = \mathbb{I} = D^{-1}$. I calcoli sono come sempre elementari, ci limitiamo ad elencare i risultati. La matrice di iterazione di Jacobi è

$$-D^{-1}(L+U) = -(L+U) = \begin{pmatrix} 0 & 0.0363 & 0.0363 & 0.0363 & 0.0363 & 0.105 \\ 0.145 & 0 & 0.075 & 0.06 & 0.075 & 0.145 \\ 0.145 & 0.075 & 0 & 0.075 & 0.06 & 0.145 \\ 0.145 & 0.06 & 0.075 & 0 & 0.075 & 0.145 \\ 0.145 & 0.075 & 0.06 & 0.0725 & 0 & 0.145 \\ 0.21 & 0.0725 & 0.0725 & 0.0725 & 0.0725 & 0 \end{pmatrix}$$

Perciò, partendo con $\boldsymbol{b}^{(0)} = \mathbf{0}$, si ottiene

$$\boldsymbol{b}^{(1)} = D^{-1}\boldsymbol{e} = \boldsymbol{e} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

ed analogamente

$$\boldsymbol{b}^{(2)} = (L+U)\boldsymbol{b}^{(1)} + \boldsymbol{e} = \begin{pmatrix} 0.105 \\ 0.145 \\ 0.145 \\ 0.145 \\ 0.145 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.105 \\ 0.145 \\ 0.145 \\ 0.145 \\ 0.145 \\ 1 \end{pmatrix},$$

$$\boldsymbol{b}^{(3} = (L+U)\boldsymbol{b}^{(2)} + \boldsymbol{e} = \begin{pmatrix} \frac{14}{200} \\ \frac{27}{200} \\ \frac{27}{200} \\ \frac{27}{200} \\ \frac{27}{200} \\ \frac{209}{200} \end{pmatrix},$$

$$\boldsymbol{b}^{(4} = (L+U)\boldsymbol{b}^{(3)} + \boldsymbol{e} = \begin{pmatrix} 0.1261 \\ 0.1306 \\ 0.1306 \\ 0.1306 \\ 0.1306 \\ 0.0642 \end{pmatrix}.$$

6.15.3.2. *Metodo di Gauss–Seidel.* Ora possiamo eseguire le iterazioni come nei precedenti esempi della tenda piramidale e della stanza cubica. Per non diventare ripetitivi, ci limitiamo a riportare i risultati del metodo di Gauss–Seidel (questa volta alle prime quattro cifre decimali). Il metodo di Southwell verrà implicitamente trattato fra qualche pagina insieme al raffinamento progressivo.

$$\boldsymbol{b}^{(1)} = (0.1005, 0.0871, 0.0936, 0.0993, 0.1067, 1.0491)$$
$$\boldsymbol{b}^{(2)} = (0.1335, 0.1164, 0.118, 0.1192, 0.1202, 1.0624)$$
$$\boldsymbol{b}^{(3)} = (0.1411, 0.1225, 0.1228, 0.1231, 0.1233, 1.0653)$$
$$\boldsymbol{b}^{(4)} = (0.1427, 0.1238, 0.1238, 0.1239, 0.1239, 1.0659)$$
$$\boldsymbol{b}^{(5)} = (0.1430, 0.124, 0.124, 0.1241, 0.1241, 1.066)$$
$$\boldsymbol{b}^{(6)} = (0.1431, 0.1241, 0.1241, 0.1241, 0.1241, 1.066)$$
$$\boldsymbol{b}^{(7)} = (0.1431, 0.1241, 0.1241, 0.1241, 0.1241, 1.066)$$

Ecco il notebook che abbiamo usato per il calcolo:

```
t1=0.0725; t2=0.075; t3=0.06; t4=0.145
P={{0,t1,t1,t1,t1,0.1005},{0,0,t2,t3,t2,t1},{0,0,0,t2,t3,t1},
{0,0,0,0,t2,t1},{0,0,0,0,0,t1},{0,0,0,0,0,0}}
NN={{1,0,0,0,0,0},{-t4,1,0,0,0,0},{-t4,-t2,1,0,0,0},
{-t4,-t3,-t2,1,0,0},{-t4,-t2,-t3,-t2,1,0},
{-0.21,-t1,-t1,-t1,-t1,1}}; TraditionalForm[NN]
e={0,0,0, 0,0,1}
Ninv=Inverse[NN]; TraditionalForm[Ninv]
b={0,0,0,0,0,0}
For[i=0, i<10, i++, N[Expand[b=Ninv .P. b + Ninv .e]]; Print[b]  ]
```

Si noti quanto rapido sia l'aumento delle componenti del vettore di radiosità nelle prime iterazioni: infatti all'inizio l'iterazione cambia la prima cifra decimale. Poi la prima cifra si stabilizza e comincia a cambiare la seconda, e via via le altre. Quindi le differenze successive diventano via via ordini di grandezza più piccoli. In effetti, osserviamo che le prime due cifre decimali sono stabili già dopo tre iterazioni, le prime tre dopo quattro iterazioni, le prime quattro dopo sei iterazioni. Proprio per questo rapido incremento iniziale e lento incremento successivo il metodo del raffinamento progressivo, introdotto nelle sezioni Verso un raffinamento progressivo e seguenti, permette di estrapolare in pochissime iterazioni il punto fisso finale. Tra qualche pagina svolgeremo i calcoli con questo metodo.

6.15.3.3. *Metodo di Southwell.* In questo metodo, utilizziamo il vettore dell'energia creata $\boldsymbol{\varepsilon}$, quello dell'energia irradiata $\boldsymbol{\beta}$, e la matrice dell'energia K. Le pareti laterali hanno area 2, mentre pavimento (di indice $i = 1$) e soffitto ($i = 6$) hanno area 4. Poiché $K_{ij} = \frac{A_i}{A_j} M_{ij}$, la matrice K si decompone come $K = N' - P'$, dove le matrici N' e P' differiscono dalle matrici N e P utilizzate precedentemente in questo modo:

- N' differisce da N solo nella ultima riga, colonne 2,3,4,5 (dove i coefficienti vengono moltiplicati per $\frac{1}{2}$), e nella prima colonna, righe 2,3,4,5 (i cui coefficienti vengono moltiplicati per 2);
- P' differisce da P solo nella prima riga, colonne 2,3,4,5 (dove i coefficienti vengono moltiplicati per 2), e nella ultima colonna, righe 2,3,4,5 (i cui coefficienti vengono moltiplicati per $\frac{1}{2}$).

Ora scriviamo per esteso le nuove matrici. Con queste matrici ed il notebook di Mathematica usato prima si possono trovare gli approssimanti iterativi $\boldsymbol{\beta}^{(k)}$, e da essi ricavare $\boldsymbol{b}^{(k)}$. Per il momento omettiamo i calcoli, perché li svolgeremo nella prossima Sottosezione nell'illustrare il raffinamento progressivo.

$$N' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -0.29 & 1 & 0 & 0 & 0 & 0 \\ -0.29 & -\frac{1}{2}\,0.15 & 1 & 0 & 0 & 0 \\ -0.29 & -\frac{1}{2}\,0.12 & -\frac{1}{2}\,0.15 & 1 & 0 & 0 \\ -0.29 & -\frac{1}{2}\,0.15 & -\frac{1}{2}\,0.12 & -\frac{1}{2}\,0.15 & 1 & 0 \\ -\frac{1}{2}\,0.42 & -\frac{1}{4}\,0.145 & -\frac{1}{4}\,0.145 & -\frac{1}{4}\,0.145 & -\frac{1}{4}\,0.145 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -0.29 & 1 & 0 & 0 & 0 & 0 \\ -0.29 & -0.075 & 1 & 0 & 0 & 0 \\ -0.29 & -0.06 & -0.07 & 1 & 0 & 0 \\ -0.29 & -0.075 & -0.06 & -0.075 & 1 & 0 \\ -0.21 & -0.03625 & -0.03625 & -0.03625 & -0.03625 & 1 \end{pmatrix}$$

$$P' = \begin{pmatrix} 0 & 0.145 & 0.145 & 0.145 & 0.145 & \frac{1}{4}\,0.42 \\ 0 & 0 & \frac{1}{2}\,0.15 & \frac{1}{2}\,0.12 & \frac{1}{2}\,0.15 & \frac{1}{8}\,0.29 \\ 0 & 0 & 0 & \frac{1}{2}\,0.15 & \frac{1}{2}\,0.12 & \frac{1}{8}\,0.29 \\ 0 & 0 & 0 & 0 & \frac{1}{2}\,0.15 & \frac{1}{8}\,0.29 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{8}\,0.29 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0.145 & 0.145 & 0.145 & 0.145 & 0.1005 \\ 0 & 0 & 0.075 & 0.06 & 0.075 & 0.03625 \\ 0 & 0 & 0 & 0.075 & 0.06 & 0.03625 \\ 0 & 0 & 0 & 0 & 0.075 & 0.03625 \\ 0 & 0 & 0 & 0 & 0 & 0.03625 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

6.15.3.4. *Raffinamento progressivo.* Come già spiegato nella Sottosezione 6.11.3, un vantaggio del metodo di che stiamo presentando (Southwell con raffinamento, Sezione 6.12, in particolare la Sottosezione 6.12.2) è che si presta a riconsiderare il sistema lineare della radiosità dal punto di vista nel quale si considera l'energia irradiata da ogni elemento nell'ambiente invece che ricevuta dall'ambiente. In questo modo si ottengono cospicui guadagni di tempo e di allocazione di memoria nel calcolo dei fattori di forma, come visto alla fine della Sottosezione 6.12.1. Poiché però i fattori di forma li abbiamo già calcolati, tralasciamo questo aspetto e ci rivolgiamo alla estrapolazione del punto di equilibrio tramite raffinamento progressivo degli approssimanti. Questo richiede una stima della energia residua da irradiare, pesata con le aree degli elementi, presentata nella Sottosezione 6.12.3, alla quale facciamo riferimento per le formule che stiamo per citare.
Rammentiamo gli ingredienti di questa stima. Gli approssimanti della radiosità vengono incrementati del termine in (6.12.11):

$$b_i'^{(k)} = b_i^{(k)} + \rho_i\,\Phi,$$

dove Φ è una stima del flusso per unità di area dell'energia totale non irradiata, dato da (6.12.10): $\Phi = \frac{RH}{A}$.
In quest'ultima equazione, A è la somma totale delle aree degli elementi (nel nostro caso $A = 16$, perché pavimento e soffitto hanno area 4 e le pareti hanno area 2), R è la riflettività media ambientale che tiene conto di un numero arbitrario di interriflessioni della luce, e che si ottiene a partire dalla media pesata della riflettività degli elementi,

$$\rho_{\text{media}} = \frac{\sum_{i=1}^{n} \rho_i A_i}{\sum_{i=1}^{n} A_i}\ ,$$

grazie a (6.12.8):

$$R = \frac{1}{1 - \rho_{\text{media}}}\ .$$

Infine, $H = H^{(k)}$ è una stima dell'energia totale ancora non irradiata nell'ambiente dopo k iterazioni (ossia interiflessioni):

$$H^{(k)} = \sum_{i=1}^{n} \sigma_i^{(k)}$$

con il termine residuo dell'energia dato da (6.12.11):

$$\sigma_i^{(k)} = \beta_i^{(k+1))} - \beta_i^{(k)},$$

che è equivalente a $\boldsymbol{r}^{(k)} = \boldsymbol{\varepsilon} - K\boldsymbol{\beta}^{(k)}$.

Nel nostro esempio, il pavimento ha riflettività $\frac{1}{4}$ ed area 4, le altre pareti ed il soffitto hanno riflettività $\frac{1}{2}$ ed area rispettivamente 2 e 4, quindi

$$\rho_{\text{media}} = \frac{(4 \cdot \frac{1}{4} + (4 \cdot 2 + 4) \cdot \frac{1}{2}}{16} = \frac{7}{16} = 0.4375.$$

Pertanto $R = \frac{16}{9} = 1.778$, e diventa facile trovare $bi'^{(k)}$ da (6.12.11): basta generare le iterazioni $\beta_i^{(k)}$ col metodo di Southwell, calcolare $\sigma_i^{(k)}$ da (6.12.4), quindi $H^{(k)}$ da (6.12.9) ed infine $\Phi = \Phi^{(k)}$ da (6.12.10). Tutto questo si ottiene quindi con questa semplice modifica del notebook con cui abbiamo generato le iterazioni del metodo di Southwell nella del tutto analogo a quello già presentato nelle Sottosezioni 6.15.1.2 e 6.15.2.2:

```
A={4,2,2,2,2,4};
rho={1/4,1/2,1/2,1/2,1/2,1/2};t=0.03625;t1=0.075;t2=0.06;
t3=0.21;t4=0.29;
t5=0.1005;
P={{0,t4/2,t4/2,t4/2,t4/2,t5},{0,0,t1,t2,t1,t },{0,0,0,t1,t2,t},
{0,0,0,0,t1,t},{0,0,0,0,0,t},{0,0,0,0,0,0}};
NN={{1,0,0,0,0,0},{-t4,1,0,0,0,0},{-t4,-t1,1,0,0,0},
{-t4,-t2,-t1,1,0,0},{-t4,-t1,-t2,-t1,1,0},
{-t3,-t,-t,-t,-t,1}};
K=NN-P; * In questo esempio, K e' la la matrice della potenza *
D= {{1,0,0,0,0,0},{0,1,0,0,0,0},{0,0,1,0,0,0},
{0,0,0,1,0,0},{0,0,0,0,10t},{0,0,0,0,0,1}}.
* In questo esempio, la matrice D e' l'identita' *
Dinv=Inverse[D].
* In questo esempio, Dinv = D = identita' *
Q=Dinv.(K-D).
e={0,0,0, 0,0,1};
b={0,0,0,0,0,0};
eps=e; beta=b; bprime=b;
AreaTot=0;
For [i=1,i<7,i++,AreaTot=AreaTot+A[[i]]];
rhoMedia=0;
For [i=1,i<7,i++,rhoMedia=rhoMedia+rho[[i]]  A[[i]]  ];
rhoMedia=rhoMedia/AreaTot;
R=1/(1-rhoMedia);
kk=0;maxr=1;
While[(maxr>0.001),
  r=N[eps-K.beta];
  maxr=Max[r[[1]], r[[2]], r[[3]], r[[4]], r[[5]],r[[6]]];
  For [i=1, i<7, i++,
    If [(r[[i]]==maxr),i0=i,Continue]
    ];
  beta[[i0]]=N[Q[[i0]]. beta+ Dinv[[i0]] .eps];
  For [i=1,i<7,i++,b[[i]]=beta[[i]]/A[[i]]];
  H=0;
  For [i=1,i<7,i++,H=H+r[[i]]];
  Phi=R H/AreaTot;
```

```
  For [i=1,i<7,i++,bprime[[i]]=b[[i]]+ rho[[i]] Phi];
  Print[kk]; Print[N[b]];Print[N[bprime]];
kk++]
```

Alla trentatreesima iterazione i due metodi danno lo stesso risultato, e da quel momento in poi le prime quattro cifre decimali si stabilizzano. Il metodo numerico basato sul raffinamento è ad ogni iterazione considerevolmente più vicino al risultato finale di quanto non sia il metodo di Southwell. Nonostante questo, entrambi i metodi impiegano lo stesso numero di iterazioni a stabilizzarsi, ma già alla sesta iterazione il metodo del raffinamento fornisce una buona approssimazione del risultato finale. A differenza del metodo di Southwell, il raffinamento non produce una approssimazione monotòna crescente, anzi, dopo una crescita notevolissima al primo passo, gli approssimanti del raffinamento convergono in maniera decrescente, ma ad ogni passo essi sono più grandi di quelli di Southwell, come deve essere a causa di (6.12.11).

Ecco le iterazioni successive del metodo di Southwell ricavate da Mathematica. Per ogni iterazione, il primo vettore è l'approssimante vero, il secondo l'approssimante aumentato nel senso del raffinamento progressivo:

```
1
{0.,0.,0.,0.,0.,0.25}
{0.0277778,0.0555556,0.0555556,0.0555556,0.0555556,0.305556}
2
{0.025125,0.,0.,0.,0.,0.25}
{0.0319444,0.0136389,0.0136389,0.0136389,0.0136389,0.263639}
3
{0.025125,0.,0.,0.,0.040056,0.25}
{0.0329774,0.0157047,0.0157047,0.0157047,0.0557607,0.265705}
4
{0.025125,0.,0.,0.0425984,0.040056,0.25}
{0.0316227,0.0129954,0.0129954,0.0555938,0.0530514,0.262995}
5
{0.025125,0.,0.0449013,0.0425984,0.040056,0.25}
{0.030182,0.0101141,0.0550154,0.0527125,0.05017,0.260114}
6
{0.025125,0.0469889,0.0449013,0.0425984,0.040056,0.25}
{0.0286635,0.0540659,0.0519783,0.0496754,0.047133,0.257077}
7
{0.0377795,0.0469889,0.0449013,0.0425984,0.040056,0.25}
{0.0397288,0.0508876,0.0488001,0.0464971,0.0439547,0.253899}
8
{0.0377795,0.0469889,0.0449013,0.0425984,0.040056,0.261529}
{0.0402491,0.0519281,0.0498405,0.0475376,0.0449952,0.266468}
9
{0.0377795,0.0469889,0.0449013,0.0425984,0.0522409,0.261529}
{0.0392826,0.0499951,0.0479076,0.0456046,0.0552471,0.264535}
10
{0.0377795,0.0469889,0.0449013,0.0530102,0.0522409,0.261529}
{0.0388705,0.049171,0.0470834,0.0551923,0.0544229,0.263711}
11
{0.0377795,0.0469889,0.0535775,0.0530102,0.0522409,0.261529}
{0.0385184,0.0484667,0.0550553,0.054488,0.0537187,0.263006}
12
```

```
{0.0412054,0.0469889,0.0535775,0.0530102,0.0522409,0.261529}
{0.0416509,0.0478799,0.0544684,0.0539012,0.0531318,0.26242}
13
{0.0412054,0.053977,0.0535775,0.0530102,0.0522409,0.261529}
{0.0417917,0.0551496,0.0547501,0.0541829,0.0534135,0.262701}
14
{0.0412054,0.053977,0.0535775,0.0530102,0.0522409,0.262703}
{0.0415554,0.0546769,0.0542774,0.0537102,0.0529409,0.263403}
15
{0.0412054,0.053977,0.0535775,0.0530102,0.0547526,0.262703}
{0.0414569,0.05448,0.0540805,0.0535133,0.0552557,0.263206}
16
{0.0420122,0.053977,0.0535775,0.0530102,0.0547526,0.262703}
{0.0421788,0.0543101,0.0539107,0.0533434,0.0550858,0.263036}
17
{0.0420122,0.053977,0.0535775,0.0549109,0.0547526,0.262703}
{0.0422119,0.0543765,0.053977,0.0553104,0.0551521,0.263103}
18
{0.0420122,0.053977,0.0550148,0.0549109,0.0547526,0.262703}
{0.0421476,0.0542479,0.0552858,0.0551819,0.0550235,0.262974}
19
{0.0420122,0.0550806,0.0550148,0.0549109,0.0547526,0.262703}
{0.042099,0.0552543,0.0551886,0.0550846,0.0549263,0.262877}
20
{0.0423342,0.0550806,0.0550148,0.0549109,0.0547526,0.262703}
{0.0423837,0.0551797,0.0551139,0.05501,0.0548517,0.262802}
21
{0.0423342,0.0550806,0.0550148,0.0549109,0.0551788,0.262703}
{0.042397,0.0552062,0.0551404,0.0550365,0.0553043,0.262829}
22
{0.0423342,0.0550806,0.0550148,0.0549109,0.0551788,0.262898}
{0.0423825,0.0551773,0.0551116,0.0550076,0.0552755,0.262995}
23
{0.0423342,0.0550806,0.0550148,0.0552347,0.0551788,0.262898}
{0.0423662,0.0551446,0.0550789,0.0552988,0.0552428,0.262962}
24
{0.0423342,0.0550806,0.0552528,0.0552347,0.0551788,0.262898}
{0.0423552,0.0551227,0.055295,0.0552768,0.0552209,0.26294}
25
{0.0424254,0.0550806,0.0552528,0.0552347,0.0551788,0.262898}
{0.0424384,0.0551066,0.0552789,0.0552607,0.0552048,0.262924}
26
{0.0424254,0.0552633,0.0552528,0.0552347,0.0551788,0.262898}
{0.0424422,0.0552968,0.0552864,0.0552683,0.0552123,0.262932}
27
{0.0424254,0.0552633,0.0552528,0.0552347,0.0552819,0.262898}
{0.042436,0.0552845,0.055274,0.0552559,0.055303,0.262919}
28
{0.0424461,0.0552633,0.0552528,0.0552347,0.0552819,0.262898}
```

```
{0.0424532,0.0552775,0.055267,0.0552489,0.0552961,0.262912}
29
{0.0424461,0.0552633,0.0552528,0.0552884,0.0552819,0.262898}
{0.0424541,0.0552792,0.0552687,0.0553043,0.0552978,0.262914}
30
{0.0424461,0.0552633,0.0552528,0.0552884,0.0552819,0.262923}
{0.0424523,0.0552756,0.0552651,0.0553006,0.0552941,0.262935}
31
{0.0424461,0.0552633,0.0552948,0.0552884,0.0552819,0.262923}
{0.0424502,0.0552714,0.0553029,0.0552965,0.05529,0.262931}
32
{0.0424461,0.0552967,0.0552948,0.0552884,0.0552819,0.262923}
{0.0424488,0.055302,0.0553001,0.0552936,0.0552871,0.262928}
33
{0.042458,0.0552967,0.0552948,0.0552884,0.0552819,0.262923}
{0.0424595,0.0552997,0.0552978,0.0552914,0.0552849,0.262926}
```

Queste approssimazioni nelle quali ciascuna faccia e' un elemento sono terribilmente rudimentali. Per una ragionevole gradazione delle luci, una faccia dovrebbe scomporsi in molte centinaia di elementi. Ma la mole di calcoli crescerebbe notevolmente. Supponiamo ad esempio di considerare la modellazione, ancora troppo rudimentale, in cui ciascuna faccia si scompone in 25 elementi (quindi l'elemento centrale del soffitto è la lampada: finalmente il soffitto diventa piùrealistico, e non emette più luce uniformemente da tutta la propria superficie!) Ci sono in tutto $25 \cdot 6 = 150$ elementi. Quindi le matrici sono di dimensione 150. Quanti fattori di forma devono essere calcolati? A prima vista sembrerebbe di doverne calcolare $150^2 = 22500$, un numero già elevatissimo. Ma invece sono di meno, perché molti coincidono. Ad esempio, data un elemento, tutti i fattori di forma verso elementi di un'altra parete dipendono solo dallo spostamento relativo fra le due, quindi sono solo 25; inoltre, l'insieme dei fattori di forma da elementi di una parete a elementi di una parete adiacente rimane identico indipendentemente da quale delle quattro pareti adiacenti si sceglie. Quindi basta calcolare 25 fattori di forma per elementi su pareti adiacenti ed altri 25 per elementi su pareti antistanti: in tutto 50. Però poi bisogna scrivere il codice in maniera intelligentemente appropriata per fare in modo che tenga conto in modo automatico di quali fattori di forma si ripetono.
Lasciamo al lettore il calcolo di quale mole di lavoro sarebbe richiesta se anziché suddividere ogni faccia in 25 elementi la considerassimo un unico elemento ma provvedessimo a sottostrutturarla in 25 sottoelementi.

6.16. Esercizi finali sulla radiosità

Esercizio 6.16.1. Una stanza di forma cubica ha una luce di intensità 1 sul soffitto ed una finestra che lascia entrare luce con intensità 1 sulla parete sud. Come già visto nell'Esempio 6.5, il fattore di forma fra pavimento e soffitto vale $\frac{1}{5}$. Le pareti (senza finestre) ed il pavimento sono neri (coefficiente di riflettività zero); il soffitto e la parete sud hanno coefficiente di riflettività $\frac{1}{2}$. Si calcolino le prime due iterazioni del metodo di Jacobi e del metodo di Gauss–Seidel per il calcolo delle radiosità, prendendo le pareti, il pavimento ed il soffitto come elementi.

Esercizio 6.16.2. Una scena consiste dell'interno di una semisfera di raggio 2, al centro della quale c'è una cabina semisferica di raggio 1 (che protrude dal pavimento nello stesso verso dell'emisfera esterna). La scena è l'intercapedine fra le due semisfere. La luce viene da un'apertura sulla semisfera esterna, con potenza 1. La semisfera esterna ha fattore di riflettività $\frac{2}{3}$. Quella interna ha fattore di riflettività $\frac{1}{3}$. Il pavimento ha fattore di riflettività 0. Il fattore di forma dal pavimento alla semisfera esterna è $\frac{7}{8}$.

(*i*) Si trovino tutti gli altri fattori di forma.
(*ii*) Si trovino le prime tre iterazioni della radiosità ($\boldsymbol{b}^{(1)}$, $\boldsymbol{b}^{(2)}$ e $\boldsymbol{b}^{(3)}$) con il metodo di Jacobi.
(*iii*) Si trovi $\boldsymbol{b}^{(1)}$ secondo il metodo di Gauss–Seidel.
(*iv*) Si trovi la seconda iterazione del metodo di Southwell.

Esercizio 6.16.3. La scena consiste dell'interno dell'astronave Enterprise, fatta a ciambella, di metallo con fattore di riflettività $\frac{1}{2}$. Tutte le luci sono spente a causa di un attacco dei Klingon, tranne che nel pannello di comando, che emette luce con potenza 1. I membri dell'equipaggio, impavidi nonostante l'attacco, rimasti senza computers funzionanti, alla fioca luce emessa dal pannello di comando sono impegnati a calcolare a mano l'illuminazione che si distribuisce dal pannello di comando all'astronave (un modo come un altro per scoprire se nella corazza si è formato qualche buco, che ovviamente avrebbe riflettività 0 invece che $\frac{1}{2}$). Essi immaginano l'interno dell'astronave separato in quattro settori di 90 gradi tramite piani perpendicolari che si incontrano sull'asse centrale della ciambella (quello rispetto al quale la ciambella è invariante per rotazione). Così l'astronave si scompone in quattro elementi, al centro di una delle quali è il pannello di controllo. La ciambella è sufficientemente sottile perché da ciascun elemento si riescano a vedere solo quelli adiacenti. Il fattore di forma di ogni elemento su sé stesso dipende dal valore del raggio interno ed esterno della ciambella, perché se il raggio esterno è grande e quello interno è piccolo allora ognuno dei quattro elementi vede non solo quelli adiacenti ma anche il quarto, e questo cambia il valore dell'autofattore di forma; ma supponiamo che i raggi siano tali che l'autofattore di forma valga $\frac{1}{2}$ (al termine dei calcoli dovremo anche verificare che questa scelta sia compatibile con l'equazione (6.2.1) di normalizzazione dell'angolo solido totale (o di conservazione dell'energia)), e con il fatto che i fattori di forma devono essere compresi fra 0 e 1.

Purtroppo, quando comincia il calcolo l'intero elemento opposto al pannello di comando è stata demolito dai Klingon, e quindi è diventato nero (il colore dello spazio esterno). I membri dell'equipaggio, prima di essere uccisi dai Klingon, sono in grado di:

(*i*) trovare tutti i fattori di forma;
(*ii*) trovare le prime tre iterazioni della radiosità ($\boldsymbol{b}^{(1)}$, $\boldsymbol{b}^{(2)}$ e $\boldsymbol{b}^{(3)}$) con il metodo di Jacobi;
(*iii*) trovare $\boldsymbol{b}^{(1)}$ e $\boldsymbol{b}^{(2)}$ con il metodo di Gauss–Seidel;
(*iv*) trovare $\boldsymbol{b}^{(1)}$ e $\boldsymbol{b}^{(2)}$ con il metodo di Southwell.

Lo siete anche voi, prima di essere uccisi da questo esercizio?
Si assuma di sapere che l'elemento opposto al pannello di comando sia ormai nero.

Esercizio 6.16.4. Una scena consiste dell'interno di una torre cilindrica di raggio r ed altezza m. Sia α il fattore di forma dal pavimento alla parete cilindrica. È possibile determinare tutti i fattori di forma in funzione di α? Se non lo è, si mostri perché; se lo è, per quali r e m ogni scelta di α in $[0, 1]$ è compatibile con il problema? Nel caso $r = m$, per ogni scelta accettabile di α, si scriva il sistema lineare della radiosità assumendo che il pavimento sia nero, la parete ed il soffitto bianchi (ossia con fattore di riflettività 1) ed il soffitto emetta luce a potenza 1, e si calcolino le prime tre iterazioni per la radiosità con il metodo di Jacobi e con il metodo di Gauss–Seidel, e la soluzione esatta della radiosità all'equilibrio.

Svolgimento. Siano β il fattore di forma da pavimento a soffitto (e viceversa), γ quello da parete a pavimento (o soffitto), e σ quello dalla parete e sé stessa: non ci sono altri fattori di forma. L'area della parete è $2\pi rm$, quella del pavimento è πr^2, quindi, per la regola di reciprocità,

$$\gamma = \frac{r}{2m}\,\alpha. \tag{6.16.1}$$

Le identità relative alla conservazione dell'angolo solido totale sono $2\gamma + \sigma = 1$ e $\alpha + \beta = 1$. Grazie a (6.16.1), la prima di queste identità diventa

$$\sigma = 1 - \frac{r}{m}\,\alpha \tag{6.16.2}$$

e la seconda è

$$\beta = 1 - \alpha. \tag{6.16.3}$$

Quindi tutti i fattori di forma sono funzioni di α. Affinché il risultato sia accettabile occorre che tutti i fattori di forma siano positivi e non superiori a 1. Quindi, per (6.16.2), se $r \leqslant m$, σ appartiene a $\left[0, 1 - \frac{r}{m}\right] \subset [0,1]$ per ogni valore di $\alpha \in [0,1]$, e, grazie a (6.16.3), $\beta \in [0,1]$ per ogni $\alpha \in [0,1]$. Se invece $r > m$ la condizione $\sigma \geqslant 0$ implica che deve essere $\alpha \geqslant \frac{m}{r}$.

In particolare, se $r = m$ ogni scelta di $\alpha \in [0,1]$ è accettabile. In tal caso, numerando con 1 il pavimento, 2 la parete e 3 il soffitto, il vettore della potenza creata diventa $\boldsymbol{e} = (0,\, 0,\, 1)$ e la matrice M della radiosità diventa

$$M = \begin{pmatrix} 1 & 0 & 0 \\ -\rho\gamma & 1-\rho\sigma & -\rho\gamma \\ -\rho\beta & -\rho\alpha & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{\rho\alpha}{2} & 1-\rho\left(1-\frac{\alpha}{2}\right) & -\frac{\rho\alpha}{2} \\ -\rho(1-\alpha) & -\rho\alpha & 1 \end{pmatrix}$$

La matrice di iterazione di Jacobi è $Q = -D^{-1}(L+U)$, ed il procedimento iterativo di Jacobi (6.7.10) è $\boldsymbol{b}^{(k+1)} = -D^{-1}(L+U)\boldsymbol{b}^k + D^{-1}\boldsymbol{e}$. Abbiamo già osservato che l'applicazione di questo metodo non presenta alcuna difficoltà di calcolo: qui la svolgiamo solo perché non abbiamo fissato numericamente i coefficienti di matrice, ma li abbiamo lasciati in forma simbolica. La matrice D^{-1} si ottiene direttamente prendendo i reciproci dei suoi termini:

$$D^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{1-\rho(1-\alpha)} & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{perché} \quad D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1-\rho(1-\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

e la matrice $L+U$ consiste dei restanti termini di M:

$$L+U = \begin{pmatrix} 0 & 0 & 0 \\ -\frac{\rho\alpha}{2} & 0 & -\frac{\rho\alpha}{2} \\ -\rho(1-\alpha) & -\rho\alpha & 0 \end{pmatrix}$$

Pertanto,

$$Q = -D^{-1}(L+U) = -\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{1-\rho(1-\alpha)} & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 & 0 & 0 \\ -\frac{\rho\alpha}{2} & 0 & -\frac{\rho\alpha}{2} \\ -\rho(1-\alpha) & -\rho\alpha & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 \\ \frac{\rho\alpha}{2(1-\rho(1-\alpha))} & 0 & \frac{\rho\alpha}{2(1-\rho(1-\alpha))} \\ \rho(1-\alpha) & \rho\alpha & 0 \end{pmatrix}$$

Ora, da $\boldsymbol{b}^{(0)} = 0$ si ottiene

$$\boldsymbol{b}^{(1)} = D^{-1}\boldsymbol{e} = D^{-1}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

e poi

$$\boldsymbol{b}^{(2)} = Q\boldsymbol{b}^{(1)} + D^{-1}\boldsymbol{e} = (Q + \mathbb{I}) \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{\rho\alpha}{2(1-\rho(1-\alpha))} & 1 & \frac{\rho\alpha}{2(1-\rho(1-\alpha))} \\ \rho(1-\alpha) & \rho\alpha & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ \frac{\rho\alpha}{2(1-\rho(1-\alpha))} \\ 1 \end{pmatrix}$$

e

$$\boldsymbol{b}^{(3)} = Q\boldsymbol{b}^{(2)} + D^{-1}\boldsymbol{e} = Q \begin{pmatrix} 0 \\ \frac{\rho\alpha}{2(1-\rho(1-\alpha))} \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad = \begin{pmatrix} 0 \\ \frac{\rho\alpha}{2(1-\rho(1-\alpha))} \\ 1 + \frac{(\rho\alpha)^2}{2(1-\rho(1-\alpha))} \end{pmatrix}$$

Adesso applichiamo invece il metodo di Gauss–Seidel, i cui calcoli sono lievemente più impegnativi. Spezziamo $M = N - P$, dove

$$N = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{\rho\alpha}{2} & 1-\rho(1-\alpha) & 0 \\ -\rho(1-\alpha) & -\rho\alpha & 1 \end{pmatrix}$$

e

$$P = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \frac{\rho\alpha}{2} \\ 0 & 0 & 0 \end{pmatrix}.$$

Qui N è del tipo

$$N = \begin{pmatrix} 1 & 0 & 0 \\ x & a & 0 \\ z & y & 1 \end{pmatrix},$$

e si verifica subito che la sua inversa è

$$N^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{x}{a} & \frac{1}{a} & 0 \\ \frac{xy-az}{a} & -\frac{y}{a} & 1 \end{pmatrix}.$$

Quindi nel nostro caso

$$N^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2}\ \frac{\rho\alpha}{1-\rho(1-\alpha)} & \frac{1}{1-\rho(1-\alpha)} & 0 \\ \frac{\rho\alpha(1-\alpha)+\frac{1}{2}\rho^2\alpha^2}{1-\rho(1-\alpha)}\ \frac{\rho\alpha(1-\alpha)+\rho^2(1-2\alpha)}{1-\rho(1-\alpha)} & \frac{\rho\alpha}{1-\rho(1-\alpha)} & 1 \end{pmatrix}.$$

Pertanto si ha

$$N^{-1}P = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2}\,\frac{\rho\alpha}{1-\rho(1-\alpha)} \\ 0 & 0 & \frac{1}{2}\,\frac{\rho^2\alpha^2}{1-\rho(1-\alpha)} \end{pmatrix}.$$

La potenza creata per unità di area è il vettore $\boldsymbol{e} = (0, 0, 1)$. Partendo dall'approssimazione iniziale $\boldsymbol{b}^{(0)} = (0, 0, 0)$ si ottiene $\boldsymbol{b}^{(1)} = N^{-1}\boldsymbol{e} = (0, 0, 1)$ (la terza colonna della matrice N^{-1}). Quindi $N^{-1}P\boldsymbol{b}^{(1)}$ è la terza colonna della matrice $N^{-1}P$, e pertanto

$$\boldsymbol{b}^{(2)} = N^{-1}P\boldsymbol{b}^{(1)} + N^{-1}\boldsymbol{e} = (N^{-1}P + \mathbb{I})\boldsymbol{b}^{(1)} = \frac{1}{2}\begin{pmatrix} 0 \\ \frac{\rho\alpha}{1-\rho(1-\alpha)} \\ \frac{\rho^2\alpha^2}{1-\rho(1-\alpha)} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ \frac{1}{2}\,\frac{\rho\alpha}{1-\rho(1-\alpha)} \\ 1 + \frac{1}{2}\,\frac{\rho^2\alpha^2}{1-\rho(1-\alpha)} \end{pmatrix}.$$

Quindi

$$\boldsymbol{b}^{(3)} = N^{-1}P\boldsymbol{b}^{(2)} + N^{-1}\boldsymbol{e} = N^{-1}P\begin{pmatrix} 0 \\ \frac{\rho\alpha}{1-\rho(1-\alpha)} \\ \frac{\rho^2\alpha^2}{1-\rho(1-\alpha)} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ \frac{\rho\alpha}{2(1-\rho(1-\alpha)} + \frac{\rho^3\alpha^3}{4(1-\rho(1-\alpha)^2} \\ \frac{\rho^2\alpha^2}{2(1-\rho(1-\alpha)} + \frac{\rho^4\alpha^4}{4(1-\rho(1-\alpha)^2} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ \frac{\rho\alpha}{2(1-\rho(1-\alpha)} + \frac{\rho^3\alpha^3}{4(1-\rho(1-\alpha)^2} \\ 1 + \frac{\rho^2\alpha^2}{2(1-\rho(1-\alpha)} + \frac{\rho^4\alpha^4}{4(1-\rho(1-\alpha)^2} \end{pmatrix}.$$

Nel nostro caso, $\rho = \alpha = \frac{1}{2}$, e quindi

$$\boldsymbol{b}^{(1)} = (0,\, 0,\, 1),$$
$$\boldsymbol{b}^{(2)} = (0,\, \frac{1}{6},\, \frac{25}{24}) = (0, 0.166667, 1.041667),$$
$$\boldsymbol{b}^{(3)} = (0,\, \frac{1}{6} + \frac{1}{144},\, 1 + \frac{1}{24} + \frac{1}{576}) = (0,\, 0.173611, 1.0434035).$$

Esercizio 6.16.5. Consideriamo una stanza cubica di lato 10 metri nella quale sono uguali a $\frac{1}{5}$ tutti i fattori di forma non nulli (Sezione 6.5). Nella stanza stendiamo un telo verticale, dal lato che congiunge due pareti a quello opposto (ossia quello che congiunge le due pareti opposte). Il telo divide la stanza in due parti uguali: consideriamo solo una delle due. Il telo e le pareti di questa metà della stanza sono elementi. Numeriamo il soffitto (triangolare) con 1, le due pareti (quadrate)

con 2 e 3, il pavimento (triangolare) con 4, il telo con 5. Il telo è nero, le pareti ed il soffitto sono bianchi. Naturalmnete, non essendoci ostruzioni fra una parete quadrata e l'altra, il fattore di forma d parete a parete rimane quello della stanza cubica, ovvero $\frac{1}{5}$.

(*i*) Assumiamo una ipotesi non necessariamente vera: che i fattori di forma da soffitto a pavimento e da soffitto a parete valgano $\frac{1}{5}$ (sappiamo che lo stesso vale per il fattore di forma da parete a parete). Questi dati sono sufficienti per calcolare tutti i fattori di forma? Sono compatibili fra loro? Se sì li si calcoli, se no si spieghi perché.

(*ii*) Come cambia la risposta se si chiede solo che il fattore di forma da soffitto a pavimento e quello da parete a parete siano $\frac{1}{5}$?

(*iii*) Rammentando che il fattore di form da parete a parete è noto e vale $\frac{1}{5}$, si determini il range di valori ammissibili per il fattore di forma da soffitto a pavimento.

Svolgimento. Poiché gli elementi sono piani, i fattori di forma da un elemento a sé stesso sono nulli. Per simmetria, ci sono solo cinque fattori di forma indipendenti: da triangolo (soffitto) a quadrato (parete intera), da triangolo a triangolo (soffitto a pavimento), da rettangolo (telo) a triangolo, da quadrato a rettangolo e da quadrato a quadrato: i fattori di forma in senso inverso si ricavano da questi tramite l'equazione di reciprocità $A_i F_{ij} = A_j F_{ji}$. Indichiamo questi cinque fattori di forma con F_{TQ}, F_{TT}, F_{RT}, F_{QR} e F_{QQ}, rispettivamente. Osservando che le aree di questi elementi verificano

$$\begin{aligned} A_Q &= 2A_T \\ A_R &= \sqrt{2}A_Q\,, \end{aligned}$$

dalla relazione di reciprocità (6.2.2) otteniamo

$$\begin{aligned} F_{QT} &= \frac{1}{2}F_{TQ}\,, \\ F_{TR} &= 2\sqrt{2}F_{RT}\,, \\ F_{QR} &= \sqrt{2}F_{RQ}\,. \end{aligned}$$

Ora applichiamo la relazione (6.2.1) di conservazione dell'energia (ovvero dell'angolo solido), $\sum_j F_{ij} = 1$. Si ottiene il sistema lineare:

$$\begin{aligned} F_{QQ} + F_{QR} + 2F_{QT} &= 1, \\ F_{TT} + F_{TR} + 2F_{TQ} &= 1, \\ 2F_{RT} + 2F_{RQ} &= 1 \end{aligned}$$

Sostituendo in queste equazioni le precedenti identità ora abbiamo

$$\begin{aligned} F_{QQ} + \sqrt{2}F_{RQ} + F_{TQ} &= 1, \\ F_{TT} + 2\sqrt{2}F_{RT} + 2F_{TQ} &= 1, \\ 2F_{RT} + 2F_{RQ} &= 1. \end{aligned}$$

Scriviamo $a = F_{QQ}$: sappiamo che $a = \frac{1}{5}$, ma lasciamo il simbolo a nel caso il lettore voglia cimentarsi con lo stesso problema per una stanza rettangolare, dove il fattore di forma varia con il rapporto fra larghezza, lunghezza ed altezza (Sottosezione 6.15.3). Per semplicità, scriviamo $b = F_{TT}$, $c = F_{RQ}$, $d = F_{TQ}$, $e = F_{RT}$. Abbiamo tre equazioni in cinque incognite a, b, c, d, e in cui in realtà a non è un'incognita, e tutti questi valori sono vincolati ad appartenere all'intervallo $[0, 1]$. Le equazioni sono:

$$\begin{aligned} a + \sqrt{2}c + d &= 1, \\ b + 2\sqrt{2}e + 2d &= 1, \\ 2e + 2c &= 1. \end{aligned}$$

Dall'ultima equazione ricaviamo

$$c = \frac{1}{2} - e.$$

Sostituendo nella prima otteniamo

$$a - \sqrt{2}e + d = 1 - \frac{1}{\sqrt{2}} \ .$$

In tal modo restano due equazioni in quattro incognite (ma in realtà a è già nota). L'ipotesi della parte (i) del problema è $a = b = d = \frac{1}{5}$. Allora dalla prima equazione si ricava

$$\sqrt{2}e = \frac{1}{\sqrt{2}} - \frac{3}{5} \ ,$$

e dalla seconda si ricava

$$e = \frac{1}{5\sqrt{2}} \ .$$

Chiaramente, le due ultime equazioni per e sono inconsistenti, e quindi la scelta $b = d = \frac{1}{5}$ è incompatibile con le leggi di conservazione e la geometria del problema. Quindi i dati assegnati sono sufficienti ma incompatibili fra loro.

Nella parte (ii) si richiede solo $a = d = \frac{1}{5}$. Pertanto restano due equazioni in due incognite, le seguenti: la prima è, come prima,

$$\sqrt{2}e = \frac{1}{\sqrt{2}} - \frac{3}{5} \ ,$$

mentre la seconda diventa $b + \sqrt{2} - \frac{6}{5} + \frac{2}{5} = 1$, ma allora $b \approx 1.8 - 1.414$ e quindi $0 < b < 1$ ed i dati sono compatibili.

Per la domanda (iii), dalle equazioni viste sopra si deduce:

$$c + e = \frac{1}{2} \ ,$$

$$\sqrt{e} - d = \frac{1}{\sqrt{2}} - \frac{4}{5} \ ,$$

$$b + 4d = \frac{13}{5} - \sqrt{2}.$$

Quindi il range delle soluzioni consiste dell'intersezione dei tre iperpiani di codimensione 1 determinati da queste tre equazioni intersecata a sua volta con il cubo quadridimensionale $\{0 \leqslant b \leqslant 1,\ 0 \leqslant c \leqslant 1,\ 0 \leqslant d \leqslant 1,\ 0 \leqslant e \leqslant 1\}$. Poichè la mappa di riflessione $e \mapsto \frac{1}{2} - e = c$ lascia questo cubo invariante, una volta trovato il range di e non ci sono ulteriori restrizioni per il range di c.
Ad esempio, la seconda equazione limita il range di e all'intervallo $\left[0,\ \frac{1}{2} + \frac{1}{5\sqrt{2}}\right]$: per valori di e maggiori dell'estremo destro di questo intervallo, d diventa maggiore di 1. Si lascia al lettore verificare le limitazioni del range per le altre variabili b, c e d.

Esercizio 6.16.6. La scena consiste di una stanza cubica di lato 1 il cui pavimento è stato sostituito da una piramide il cui vertice alto tocca il centro del soffitto. Le quattro facce della piramide sono numerate da 1 a 4, le pareti da 5 a 8 (5 antistante a 1, 6 antistante a 2 e così via), ed il soffitto ha il numero 9. La piramide non riflette la luce, però la emette con potenza per unità di area uguale a 1. Le pareti 5 e 7 sono nere. Tutti gli altri coefficienti di riflettività sono $\frac{1}{2}$ tranne quello del soffitto che è $\frac{1}{4}$. Supponiamo (ma nella realtà non è così) che i fattori di forma da ciascuna faccia della piramide alla parete antistante ed al soffitto siano uguali, e valgono entrambi un numero a fissato e dato per noto, ed il fattore di forma da ciascuna parete al soffitto valga b, dove a e b sono due numeri fra 0 e 1.

(i) Qual'è l'area di ciascuna faccia della piramide?

(*ii*) Se $a = b$, i dati sono sufficienti a trovare tutti i fattori di in termini di a? Se sì, determinare i valori di a per cui i dati sono compatibili (ossia tali che tutti i fattori di forma siano compresi fra 0 e 1).

(*iii*) Se la risposta al punto precedente è negativa, assumiamo $a \neq b$. Che condizioni dobbiamo imporre su a e b affinché i dati siano compatibili? Se si sono trovate ai punti precedenti valori di a e b per i quali i dati sono compatibili, si scelga a piacere una tale coppia a e b (uguali fra loro, se la domanda al punto (*ii*) aveva risposta positiva), e si risponda alle seguenti domande:

(*iv*) Si trovino le prime due iterazioni della radiosità $(\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)})$ con il metodo di Jacobi.

(*v*) Si trovino $\boldsymbol{b}^{(1)}$ e $\boldsymbol{b}^{(2)}$ con il metodo di Gauss–Seidel.

(*vi*) Si trovi la seconda iterazione del metodo di Southwell.

ESERCIZIO 6.16.7. Una stanza si compone di un cubo la cui faccia superiore è sostituita da un tetto fatto a forma di calotta a volta (estruso esternamente al cubo), con area doppia di quella delle facce del cubo. Il soffitto emette luce con potenza 1, il pavimento è nero, le pareti ed il soffitto hanno fattore di riflettività $\frac{1}{2}$. Il fattore di forma da una parete a quella opposta frontalmente vale α. Sappiamo che, se il soffitto fuoriesce dal cubo e non intrude, deve essere $\alpha = \frac{1}{5}$, come nel caso della stanza cubica della Sottosezione **??**, calcolato nella Sottosezione 6.3.2, perché non ci sono occlusioni.

(*i*) Si calcolino tutti i fattori di forma, supponendo che il soffitto sia un unico elemento. Sarebbe possibile rispondere a questa domanda se il tetto, invece di essere estruso, intrudesse nel cubo?

(*ii*) Si calcolino le prime due iterazioni della radiosità con il metodo di rilassamento di Jacobi.

(*iii*) Si calcolino le prime cinque iterazioni del metodo di Gauss–Seidel.

Svolgimento. Chiamiamo F_{lat} il fattore di forma da una faccia quadrata ad una faccia quadrata adiacente, e F_{front} quello da una faccia quadrata a quella frontale. Abbiamo già visto che, se il tetto estrude dal cubo, si ha $F_{\text{lat}} = F_{\text{front}} = \alpha = frac15$, ma manteniamo questa notazione simbolica per favorire il lettore che volesse affrontare lo stesso problema per una stanza rettangolare, nella quale $F_{\text{lat}} \neq F_{\text{front}}$.

La volta, vista dalle altre facce, copre lo stesso angolo solido che coprirebbe la faccia quadrata superiore del cubo, e quindi il fattore di forma da faccia laterale a volta vale ancora F_{lat} e quello da pavimento a volta vale F_{front}. Se invece il tetto intrudesse ma fosse convesso, allora $F_V = 0$, ma i fattori di forma da pavimento a soffitto e da parete a soffitto e da parete a parete sarebbero diversi, e quindi il numero delle incognite aumenterebbe, rendendo insolubile il problema del calcolo dei fattori di forma da principi gnerali di conservazione e di simmetria.

Poiché la volta ha area doppia delle altre facce, dalla relazione di reciprocità (6.2.2) $A_i F_{ij} = A_j F_{ji}$ otteniamo che il fattore di forma da volta a faccia laterale vale $\frac{1}{2} F_{\text{lat}}$ e quello da volta a pavimento vale $\frac{1}{2} F_{\text{front}}$. Infine, la volta è convessa invece che piana, e quindi ha un fattore di forma non nullo F_{v} verso sé stessa. Allora la conservazione dell'angolo solido totale, ossia il fatto che l'angolo solido totale sia il 100% (ovvero 1), porta alle seguenti identità:

- $4F_{\text{lat}} + F_{\text{front}} = 1$ (angolo solido visto dal pavimento o da ciascuna delle pareti);
- $F_{\text{v}} + 2F_{\text{lat}} + \frac{1}{2} F_{\text{front}} = 1$ (angolo solido visto dalla volta).

Questo sistema lineare ha due equazioni e tre incognite, di cui poniamo una, F_{front}, uguale ad un parametro fisso α compreso fra 0 e 1. Allora si possono ricavare F_{lat} dalla prima equazione e F_{v} dalla seconda in termini di α, ma occorre verificare la compatibilità: per quali valori di α questi due risultati rimangono compresi fra 0 e 1. Si trova:

- $F_{\text{lat}} = \frac{1-\alpha}{4}$, sempre compreso fra 0 e 1 se lo è α;
- $F_{\text{v}} = 1 - \frac{1-\alpha}{2} - \frac{\alpha}{2} = \alpha$, anch'esso compreso fra 0 e 1.

Quindi per ogni valore ammissibile del parametro α si trova una soluzione compatibile. Nel caso di tetto che estrude, $\alpha = \frac{1}{5}$ e tutti i fattori di forma sono calcolati. Se il tetto intrudesse, si avrebbe
Numeriamo con 1 il soffitto, con 2,3,4,5 le quattro pareti laterali e con 6 il pavimento. Poiché tutti i coefficienti di riflettività valgono $\frac{1}{2}$ tranne quello del pavimento che vale 0, la matrice M della radiosità è la seguente:

$$\begin{pmatrix} 1-\frac{\alpha}{2} & \frac{\alpha-1}{16} & \frac{\alpha-1}{16} & \frac{\alpha-1}{16} & \frac{\alpha-1}{16} & -\frac{\alpha}{4} \\ \frac{\alpha-1}{8} & 1 & \frac{\alpha-1}{8} & -\frac{\alpha}{2}\frac{\alpha-1}{8} & \frac{\alpha-1}{8} & \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & 1 & \frac{\alpha-1}{8} & -\frac{\alpha}{2}\frac{\alpha-1}{8} & \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 1 & \frac{\alpha-1}{8} & \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 1 & \frac{\alpha-1}{8} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Il vettore della potenza luminosa creata è $\boldsymbol{e} = (1, 0, 0, 0, 0, 0)$, perché solo il soffitto emette luce. Il sistema lineare della radiosità, $M\boldsymbol{b} = \boldsymbol{e}$, ci dà il vettore delle radiosità dei sei elementi, ma osserviamo che la radiosità del pavimento è 0 perché il pavimento ha coefficiente di riflettività 0 (essendo nero) e non emette luce propria. Quindi possiamo scartare l'ultima incognita e ridurci al seguente problema lineare a dimensione cinque:

$$\begin{pmatrix} 1-\frac{\alpha}{2} & \frac{\alpha-1}{16} & \frac{\alpha-1}{16} & \frac{\alpha-1}{16} & \frac{\alpha-1}{16} \\ \frac{\alpha-1}{8} & 1 & \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & 1 & \frac{\alpha-1}{8} & -\frac{\alpha}{2} \\ \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 1 & \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Con abuso di notazione, continuiamo a denotare questo sistema lineare con $M\boldsymbol{b} = \boldsymbol{e}$. Il rilassamento di Jacobi (6.7.10) approssima la soluzione $\boldsymbol{b}$ con le seguenti iterazioni: $\boldsymbol{b}^{(k+1)} = Q\boldsymbol{b}^{(k)} + D^{-1}\boldsymbol{e}$, dove Q è la matrice di iterazione $D^{-1}(L+U)$. Poichè D^{-1} differisce dalla matrice identità solo perché il suo primo elemento, invece di essere 1, è $1/\left(1-\frac{\alpha}{2}\right)$, si ottiene immediatamente

$$Q = -D^{-1}(L+U) = \begin{pmatrix} 0 & \frac{\alpha-1}{8(2-\alpha)} & \frac{\alpha-1}{8(2-\alpha)} & \frac{\alpha-1}{8(2-\alpha)} & \frac{\alpha-1}{8(2-\alpha)} \\ \frac{\alpha-1}{8} & 0 & \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & 0 & \frac{\alpha-1}{8} & -\frac{\alpha}{2} \\ \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 0 & \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 0 \end{pmatrix}$$

Partendo da $\boldsymbol{b}^{(0)} = 0$, troviamo

$$\boldsymbol{b}^{(1)} = D^{-1}\boldsymbol{e} = D^{-1}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \end{pmatrix},$$

$$\boldsymbol{b}^{(2)} = Q \begin{pmatrix} 0 \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \end{pmatrix} + D^{-1} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{(\alpha-1)^2}{16(2-\alpha)} \\ \frac{(\alpha-1)^2}{32} - \frac{\alpha(\alpha-1)}{16} \\ \frac{(\alpha-1)^2}{32} - \frac{\alpha(\alpha-1)}{16} \\ \frac{(\alpha-1)^2}{32} - \frac{\alpha(\alpha-1)}{16} \\ \frac{(\alpha-1)^2}{32} - \frac{\alpha(\alpha-1)}{16} \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \\ \frac{\alpha-1}{8} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{(\alpha-1)^2}{16(2-\alpha)} \\ \frac{1-\alpha^2}{32} + \frac{\alpha-1}{8} \\ \frac{1-\alpha^2}{32} + \frac{\alpha-1}{8} \\ \frac{1-\alpha^2}{32} + \frac{\alpha-1}{8} \\ \frac{1-\alpha^2}{32} + \frac{\alpha-1}{8} \end{pmatrix}.$$

Veniamo al metodo di Gauss–Seidel. Scriviamo $M = N - P$, con

$$N = \begin{pmatrix} 1-\frac{\alpha}{2} & 0 & 0 & 0 & 0 \\ \frac{\alpha-1}{8} & 1 & 0 & 0 & 0 \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & 1 & 0 & 0 \\ \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 1 & 0 \\ \frac{\alpha-1}{8} & \frac{\alpha-1}{8} & -\frac{\alpha}{2} & \frac{\alpha-1}{8} & 1 \end{pmatrix}$$

e

$$P = \begin{pmatrix} 0 & \frac{1-\alpha}{16} & \frac{1-\alpha}{16} & \frac{1-\alpha}{16} & \frac{1-\alpha}{16} \\ 0 & 0 & \frac{1-\alpha}{8} & \frac{\alpha}{2} & \frac{1-\alpha}{8} \\ 0 & 0 & 0 & \frac{1-\alpha}{8} & \frac{\alpha}{2} \\ 0 & 0 & 0 & 0 & \frac{1-\alpha}{8} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Calcoliamo N^{-1}, ad esempio con il metodo di Gauss. Se si lascia arbitrario il valore di α, l'espressione di N^{-1} è assai complicata e laboriosa: la riportiamo qui di seguito per comodità del lettore. La matrice N^{-1} è triangolare inferiore con sulla diagonale i valori reciproci di quelli della matrice N: quindi

$$N_{11}^{-1} = \frac{1}{1-\frac{\alpha}{2}} = \frac{2}{2-\alpha}$$

e gli altri valori diagonali valgono 1. Per quanto riguarda i valori sotto la diagonale, si trova, riga

per riga:

$$N_{21}^{-1} = \frac{\alpha - 1}{4(\alpha - 2)}$$

$$N_{31}^{-1} = \frac{9 - 10\alpha + \alpha^2}{64 - 32\alpha}\ ; \quad N_{32}^{-1} = \frac{1 - \alpha}{8}$$

$$N_{41}^{-1} = \frac{-73 + 51\alpha + 21\alpha^2 + \alpha^3}{256(\alpha - 2)}\ ; \quad N_{42}^{-1} = \frac{1 + 30\alpha + \alpha^2}{64}\ ; \quad N_{43}^{-1} = \frac{1 - \alpha}{8}$$

$$N_{51}^{-1} = \frac{649 - 476\alpha - 226\alpha^2 + 52\alpha^3 + \alpha^4}{2048(\alpha - 2)}\ ; \quad N_{52}^{-1} = \frac{65 - 3\alpha - 61\alpha^2 - \alpha^3}{512}\ ;$$

$$N_{53}^{-1} = \frac{1 + 30\alpha + \alpha^2}{64}\ ; \qquad N_{54}^{-1} = \frac{1 - \alpha}{8}\ .$$

Vediamo i risultati del metodo di Gauss–Seidel nel caso particolare $\alpha = \frac{1}{5}$, che è molto più agevole. Si trova

$$N^{-1} = \begin{pmatrix} 1.111111 & 0 & 0 & 0 & 0 \\ 0.111111 & 1 & 0 & 0 & 0 \\ 0.122222 & 0.1 & 1 & 0 & 0 \\ 0.134444 & 0.11 & 0.1 & 1 & 0 \\ 0.147889 & 0.121 & 0.11 & 0.1 & 1 \end{pmatrix}$$

e

$$P = \begin{pmatrix} 0 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0 & 0 & 0.1 & 0.1 & 0.1 \\ 0 & 0 & 0 & 0.1 & 0.1 \\ 0 & 0 & 0 & 0 & 0.1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Cominciamo con l'approssimazione iniziale $\boldsymbol{b}^{(0)} = \mathbf{0}$. Si ottiene:

$$\boldsymbol{b}^{(1)} = N^{-1}\boldsymbol{e} = \begin{pmatrix} 1.111111 \\ 0.111111 \\ 0.122222 \\ 0.134444 \\ 0.147889 \end{pmatrix}$$

e l'iterazione successiva è

$$\boldsymbol{b}^{(2)} = N^{-1}P\boldsymbol{b}^{(1)} + N^{-1}\boldsymbol{e} = N^{-1}P\boldsymbol{b}^{(1)} + \boldsymbol{b}^{(1)}$$

$$= N^{-1}\begin{pmatrix} 0.0286481 \\ 0.0433204 \\ 0.0354302 \\ 0.02552884 \\ 0.0132927 \end{pmatrix} + \begin{pmatrix} 1.111111 \\ 0.111111 \\ 0.122222 \\ 0.134444 \\ 0.147889 \end{pmatrix} = \begin{pmatrix} 1.13976 \\ 0.154431 \\ 0.157652 \\ 0.159973 \\ 0.161182 \end{pmatrix}$$

Ovviamente, per simmetria, a parte il primo valore, all'equilibrio gli altri dovrebbero essere uguali, ma due sole iterazioni sono poche. Con tre iterazioni si trova

$$\boldsymbol{b}^{(3)} = \begin{pmatrix} 1.14629 \\ 0.162511 \\ 0.162996 \\ 0.163298 \\ 0.163509 \end{pmatrix}$$

e con quattro

$$\boldsymbol{b}^{(4)} = \begin{pmatrix} 1.14735 \\ 0.163715 \\ 0.163787 \\ 0.1638368 \\ 0.163869 \end{pmatrix}$$

e con cinque

$$\boldsymbol{b}^{(5)} = \begin{pmatrix} 1.14751 \\ 0.1639 \\ 0.16392 \\ 0.163919 \\ 0.163924 \end{pmatrix}$$

Continuano le iterazioni, si vede che esse si stabilizzano al valore

$$\boldsymbol{b}^{(\infty)} = \begin{pmatrix} 1.14754 \\ 0.163934 \\ 0.163934 \\ 0.163934 \\ 0.163934 \end{pmatrix}$$

Esercizio 6.16.8. Una scena consiste dell'interno di un corridoio ad angolo, dato dall'unione di tre cubi di lato 1 , il secondo ed il terzo dei quali si congiungono al primo su due pareti adiacenti di esso (ovviamente non ci sono pareti interne). La parete del terzo cubo opposta a quella dove esso si congiunge al primo cubo e' una finestra dalla quale entra luce a potenza per unita' di area pari a 1: ovviamente il suo coefficiente di riflettività e' zero. Le altre pareti non emettono luce; le pareti

laterali ed il soffitto hanno coefficiente di riflettivita' $\frac{1}{2}$, il pavimento 0. Il pavimento, il soffitto e le sei pareti laterali, una delle quali e' la finestra, formano altrettanti elementi. Numeriamo il pavimento con 0, poi procedendo in verso orario (visto dall'alto) la finestra con 1, le due pareti corte che formano un angolo ottuso con 3 e 4, l'ultima parete corta con 5, le due pareti lunghe con 6 e 7, il soffitto con 8.

(1) Quanti sono i fattori di forma indipendenti?
(2) Supponiamo di assegnare valore α al fattore di forma dal pavimento al soffitto, β dalla parete lunga alla parete corta adiacente, γ dalla parete lunga alla parete corta ad essa parallela, δ da una parete lunga all'altra parete lunga, θ fra due pareti corte adiacenti che si vedono. Questi dati sono sufficienti a trovare tutti i fattori di forma? Si spieghi la risposta.
(3) Se lo sono, si determinino gli altri fattori di forma, se no si fissi il numero minimo indispensabile di altri fattori di forma necessario per poterli trovare tutti, e quindi si trovino i restanti.
(4) Si attribuiscano ai fattori di forma valori numerici a piacere ma compatibili, e si trovino le prime due iterazioni del metodo di Jacobi.

Esercizio 6.16.9. Come nell'Esercizio 4 della sezione Esercizi sul calcolo dei fattori di forma, si consideri una ciotola K di forma emisferica di raggio 1 appoggiata all'origine nel suo punto più basso, ossia tangente all'origine al piano $\{z = 0\}$ e contenuta nel semispazio superiore $\{z \geqslant 0\}$: quindi il bordo della ciotola è un cerchio J sul piano $\{z = 1\}$. Poiché K è convessa, il suo fattore di forma verso sé stessa è positivo.

(*i*) Si assuma che che la ciotola abbia coefficiente di riflettività $\frac{1}{2}$, e che l'esterno della ciotola non rifletta luce e si possa quindi approssimare con un coperchio costituito da un disco nero (quindi la scena diventa l'interno di K). Si mostri che il fattore di forma F_{KK} vale $\frac{1}{2}$. Al polo nord di S poniamo una luce puntiforme con potenza 1 (densità di energia per unità di tempo e area). Si calcolino le prime tre iterazioni del metodo di Jacobi per la radiosità.
(*ii*) Si calcolino le prime tre iterazioni del metodo di Gauss–Seidel.
(*iii*) Si calcolino le prime tre iterazioni del metodo di Southwell.

Svolgimento. Parte (*i*). Sia C il disco superiore nero. Sappiamo che il fattore di forma F_{CK} vale 1, perché il disco vede solo l'emisfero, ma non sé stesso ($F_{CC} = 0$, perché il disco è piano). Inoltre l'area del disco è esattamente metà dell'area dell'emisfero dello stesso raggio, quindi, per la regola di reciprocità (6.2.2), $F_{KC} = \frac{1}{2}$. Per la conservazione dell'angolo solido totale, $F_{KK} + F_{KC} = 1$, e quindi $F_{KK} = \frac{1}{2}$. Numeriamo l'elemento K con il numero 1 e l'elemento C con 2. Allora il vettore $\boldsymbol{e}$ della potenza creata è $\boldsymbol{e} = (0, 1)$. La matrice della radiosità è

$$M = \begin{pmatrix} \frac{3}{4} & -\frac{1}{4} \\ 0 & 1 \end{pmatrix}.$$

Parte (*i*). Scriviamo la matrice di iterazione di Jacobi (6.7.9), ovvero $Q = -D^{-1}(L + U)$. Ora L è la matrice nulla ed i termini diagonali e triangolare superiore di M sono

$$D = \begin{pmatrix} \frac{3}{4} & 0 \\ 0 & 1 \end{pmatrix} \qquad \text{e} \qquad U = \begin{pmatrix} 0 & \frac{1}{4} \\ 0 & 0 \end{pmatrix}.$$

Chiaramente,

$$D^{-1} = \begin{pmatrix} \frac{4}{3} & 0 \\ 0 & 1 \end{pmatrix} \qquad \text{e} \qquad Q = -N^{-1}U \begin{pmatrix} 0 & \frac{1}{3} \\ 0 & 0 \end{pmatrix}.$$

Partiamo come sempre con l'approssimazione $\boldsymbol{b}^{(0)} = \mathbf{0}$ e calcoliamo le iterazioni di Jacobi dalla formula iterativa $\boldsymbol{b}^{(k+1)} = Q\boldsymbol{b}^{(k)} + D^{-1}\boldsymbol{e}$. Si ottiene:

$$\boldsymbol{b}^{(1)} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

$$\boldsymbol{b}^{(2)} = \begin{pmatrix} \frac{1}{3} \\ 1 \end{pmatrix},$$

$$\boldsymbol{b}^{(3)} = \begin{pmatrix} \frac{1}{3} \\ 1 \end{pmatrix}.$$

Quindi alla terza iterazione gli approssimanti si sono stabilizzati, ed abbiamo raggiunto l'equilibrio, ossia la soluzione. Si noti che abbiamo scritto gli approssimanti lasciando indicate le frazioni, invece che le loro prime cifre decimali, e quindi quello trovato è il risultato esatto e non dipende dalla precisione numerica richiesta.

Parte (*ii*). Ora rieseguiamo il calcolo delle prime tre iterazioni con il metodo di Gauss–Seidel. Decomponiamo $M = N - P$, dove come sempre

$$N = \begin{pmatrix} \frac{3}{4} & 0 \\ 0 & 1 \end{pmatrix} \qquad \text{e} \qquad P = \begin{pmatrix} 0 & \frac{1}{4} \\ 0 & 0 \end{pmatrix}.$$

Chiaramente,

$$N^{-1} = \begin{pmatrix} \frac{4}{3} & 0 \\ 0 & 1 \end{pmatrix}$$

e quindi

$$N^{-1}P = \begin{pmatrix} \frac{4}{3} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & \frac{1}{4} \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{3} \\ 0 & 0 \end{pmatrix}$$

Partiamo, come al solito, con l'approssimazione iniziale $\boldsymbol{b}^{(0)} = (0, 0)$. Allora $\boldsymbol{b}^{(1)} = N^{-1}\boldsymbol{e} = (0, 1)$ (la seconda colonna di N^{-1}). Osserviamo che, per ogni t, si ha $N^{-1}P(t, 1) = \big(\frac{1}{5}, 0\big)$, e quindi $\boldsymbol{b}^{(2)} = N^{-1}P\boldsymbol{b}^{(1)} + N^{-1}\boldsymbol{e} = (\frac{1}{3}\,, 0) + (0,\, 1) = (\frac{1}{3}\,, 1)$, $\boldsymbol{b}^{(3)} = N^{-1}P\boldsymbol{b}^{(2)} + N^{-1}\boldsymbol{e} = (\frac{1}{3}\,, 0) + (0,\, 1) = (\frac{1}{3}\,, 1)$. Pertanto, dopo il terzo passo, le iterazioni si stabilizzano: la radiosità della ciotola K è $\frac{1}{3}$ della potenza emessa dalla lampada ed il resto della luce si perde nello sfondo (ma naturalmente lo sfondo continua a creare luce con densità di potenza pari a 1).

Parte (*iii*). Ora rieseguiamo il calcolo col metodo di Southwell. Alla n-sima iterazione, dobbiamo trovare quale è la componente più grande del vettore resto $\boldsymbol{\sigma}^{(n)} = \boldsymbol{e} - M\boldsymbol{b}^{(n)}$ e rilassare solo quella. Per $n = 0$ ovviamente si ha $\boldsymbol{\sigma} = \boldsymbol{e}$, e quindi rilassiamo la seconda riga (perché la prima componente di $\boldsymbol{e}$ è 0). Il risultato è $\boldsymbol{b}^{(1)} = (0, 1)$ (ossia la prima componente non cambia). Questo era anche il risultato degli altri metodi, i quali però richiedevano di risolvere entrambe le righe del sistema lineare e non una sola (è vero che l'altra è banale, ma il software di esecuzione questo non può saperlo e quindi non può tenerne conto). Ora abbiamo $\boldsymbol{\sigma}^{(1)} = \boldsymbol{e} - M\boldsymbol{b}^{(1)} = (0, 1) - (-\frac{1}{4}\,, 1) = (-\frac{1}{4}\,, 0$. La componente più grande è la prima, e quindi rilassiamo la prima riga. Come già visto, si ottiene per la prima componente di $\boldsymbol{b}^{(2)} = N^{-1}P\boldsymbol{b}^{(1)} + N^{-1}\boldsymbol{e}$ il valore $\frac{1}{3}$. La seconda componente non cambia, resta quindi 1, ed il risultato è $\boldsymbol{b}^{(2)}) = (\frac{1}{3}\,, 1)$: lo stesso risultato degli altri due metodi. Allo stesso modo, si trova $\boldsymbol{b}^{(3)}) = (\frac{1}{3}\,, 1)$ ed a questo punto l'iterazione si è stabilizzata: ma abbiamo eseguito solo la metà delle operazioni aritmetiche (in effetti, abbiamo evitato di risolvere quelle equazioni lineari banali i cui coefficienti sono tutti nulli).

Esercizio 6.16.10. Una scena si compone del volume delimitato esternamente dalle seguenti due superficie: l'emisfero di raggio 1 e centro l'origine nel semispazio $z \geqslant 0$, ed un cono diretto verso il

basso (ossia tutto al di sotto del livello del suo vertice) nel semispazio $\{z \geqslant 0\}$, con asse centrale l'asse z e tangente all'emisfero. L'intersezione di questo cono con il piano $\{z = 0\}$ è un disco il cui raggio è pari all'altezza del cono. Le due superficie (sfera e cono) delimitano un volume V che forma la nostra scena (un altro volume è delimitato da sfera, cono ed il piano di base $\{z = 0\}$, ma in questo problema questo secondo volume non ci interessa). D'ora in poi chiamiamo C la porzione del cono dal cerchio di tangenza al vertice del cono, e S la calotta sferica consistente della parte dell'emisfero compresa fra il parallelo di tangenza ed il polo nord. Suddividiamo la scena nei due elementi C (primo elemento, che indicheremo come *soffitto*) e S (secondo elemento, *pavimento*).

Si osservi che il cono è generato, per rotazione intorno all'asse z, da una semiretta con pendenza 1 (ossia angolo $\frac{\pi}{4}$ rispetto alla verticale).

(*i*) A quale altezza z_0 il cono è tangente alla sfera? A quale angolo di latitudine sull'emisfero è situato il parallelo su cui le due superficie sono tangenti?
(*ii*) Si calcoli l'area di C.
(*iii*) Si calcoli l'area di S.
(*iv*) Si calcolino i fattori di forma F_{SS}, F_{CC}, F_{SC}, F_{CS}, usando la reversibilità dei percorsi ottici (**??**) e la conservazione dell'angolo solido totale (o dell'energia) (6.2.1). Scrivere la matrice M della radiosità assumendo che i fattori di riflettività siano $\rho_S = 1/2$, $\rho_C = 1/4$.
(*v*) Si calcoli M^{-1}. Assumendo che una lampada di potenza 1 stia al vertice del cono, si calcolino le radiosità.
(*vi*) Si consideri il fattore di forma *differenziale* dal vertice del cono alla calotta S, già calcolato in un esercizio precedente. Si noti che il pavimento della scena (la calotta sferica S) è convesso. Rimpiazziamolo con un pavimento piatto (il disco di base B del cono, considerato nella domanda (*iii*)), o con uno concavo (ad esempio un'altra calotta sferica S', della stessa area di S e della stessa curvatura, ma concava, ossia che si protende verso l'esterno del volume del cono invece che verso l'interno: si tratta della riflessione speculare di S rispetto al disco di base B). Come cambia il fattore di forma differenziale?
(*vii*) Supponiamo di rimpiazzare S con un nuovo pavimento S' dato dalla calotta sferica riflessa speculare concava introdotta nella domanda (*vi*). In tal modo si ottengono altri fattori di forma, ad esempio $F_{CS'}$ invece di F_{CS}. Qual è più grande fra $F_{CS'}$ e F_{CS}?
(*viii*) Si ripeta il calcolo dei fattori di forma e della radiosità per il solido delimitato dal cono C e dalla superficie sferica concava S' considerata nella parte precedente.

Svolgimento. Parte (i). L'angolo formato dalla superficie del cono con l'asse z è $\frac{\pi}{4}$. Quindi il parallelo a cui il cono è tangente alla sfera deve avere la stessa pendenza. La inclinazione della sfera rispetto alla verticale è $\frac{\pi}{4}$ al parallelo di latitudine $\frac{\pi}{4}$. Questo parallelo è la circonferenza $z = \cos\frac{\pi}{4} = \frac{1}{\sqrt{2}}$, $x^2 + y^2 = \sin\frac{\pi}{4} = \frac{1}{\sqrt{2}}$.
Parti (ii) e (iii). Abbiamo calcolato l'area di coni e calotte sferiche nella Sezione 5.2.

Rammentiamo il calcolo dell'area della calotta sferica: si usa la formula del fattore di ingrandimento delle aree per una superficie parametrica del tipo $p(x, y) = (x, y, f(x, y))$, dove f è una funzione di due variabili. Nel nostro caso, sia B il disco di base del cono C, ovvero $B = \left\{z = \frac{1}{\sqrt{2}},\ x^2 + y^2 \leqslant \frac{1}{\sqrt{2}}\right\}$, e K la sua proiezione sul piano $z = 0$, ossia $K = \left\{x^2 + y^2 \leqslant \frac{1}{\sqrt{2}}\right\}$. Allora f è la funzione $\sqrt{1 - x^2 - y^2}$, il cui grafico sul dominio K è appunto la calotta sferica S. Il fattore di ingrandimento delle aree in tal caso risulta essere

$$\sqrt{1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial x}\right)^2} = \frac{1}{\sqrt{1 - x^2 - y^2}}\ .$$

Quindi l'area è data dal seguente integrale

$$\begin{aligned}\text{Area}(S) &= \int_K \frac{1}{\sqrt{1-x^2-y^2}}\,dx\,dy = \int_0^{2\pi}\int_0^{\frac{1}{\sqrt{2}}} \frac{1}{\sqrt{1-r^2}}\,r\,dr\,d\theta \\ &= \pi\int_0^{\frac{1}{2}} \frac{1}{\sqrt{1-u}}\,du = 2\pi\,\sqrt{1-v}\Big|_{\frac{1}{2}}^{0} = 2\pi(1-\frac{1}{\sqrt{2}}) = \pi(2-\sqrt{2})\,.\end{aligned}$$

Si rammenti anche che, in base alla stessa formula per il fattore di ingrandimento, l'area del cono è πAR, dove R è il raggio della sua circonferenza di base e A è la distanza da un generico punto della circonferenza di base al vertice del cono. Il cono C ha altezza h uguale al raggio di base, che è $\frac{1}{\sqrt{2}}$, quindi $R = \frac{1}{\sqrt{2}}$ e, per il teorema di Pitagora, $A = \sqrt{R^2+h^2} = 1$. Pertanto $\text{Area}(C) = \frac{\pi}{\sqrt{2}}$.

Parte (iv). Poiché la calotta sferica S è convessa, il fattore di forma F_{SS} è nullo, e per la conservazione dell'angolo solido totale abbiamo $F_{SC} = 1$. Allora, per la relazione di reciprocità ed il calcolo delle aree nelle parti precedenti, si ha

$$F_{CS} = \frac{\text{Area}(S)}{\text{Area}(C)}\,F_{SC} = 2(\sqrt{2}-1)\,.$$

Invece il cono C è concavo e quindi $F_{CC} > 0$; ancora per la conservazione dell'angolo solido totale si ha $F_{CC} + F_{CS} = 1$, da cui

$$F_{CC} = 1 - F_{CS} = 3 - 2\sqrt{2}\,.$$

Pertanto, con i valori scelti $\rho_C = 1/4$ e $\rho_S = 1/2$, la matrice M della radiosità risulta essere

$$\begin{aligned}M &= \begin{pmatrix} 1-\rho_C F_{CC} & -\rho_C F_{CS} \\ -\rho_S F_{SC} & 1-\rho_S F_{SS}\end{pmatrix} = \begin{pmatrix} 1-\frac{1}{4}F_{CC} & -\frac{1}{4}F_{CS} \\ -\frac{1}{2}F_{SC} & 1\end{pmatrix} \\ &= \begin{pmatrix} \frac{3}{4}+\frac{1}{4}F_{CS} & -\frac{1}{4}F_{CS} \\ -\frac{1}{2} & 1\end{pmatrix} = \begin{pmatrix} \frac{1}{4}+\frac{1}{\sqrt{2}} & \frac{1}{2}-\frac{1}{\sqrt{2}} \\ -\frac{1}{2} & 1\end{pmatrix} \\ &= \begin{pmatrix} \frac{\sqrt{2}+4}{4\sqrt{2}} & \frac{\sqrt{2}-2}{2\sqrt{2}} \\ -\frac{1}{2} & 1\end{pmatrix}\,.\end{aligned}$$

Parte (v). È ben noto ed immediatamente verificato che l'inversa di una matrice

$$A = \begin{pmatrix} a & b \\ c & d\end{pmatrix}$$

è

$$A^{-1} = \frac{1}{\det A}\begin{pmatrix} d & -b \\ -c & a\end{pmatrix}$$

dove $\det A = ad - bc$. Nel caso presente,

$$\begin{aligned}\det M &= 1 - \rho_C F_{CC} - \rho_C\rho_S F_{CS} = 1 - \rho_C(1-F_{CS}) - \rho_C\rho_S F_{CS} \\ &= 1 - \rho_C + \rho_C(1-\rho_S)F_{CS}\end{aligned}$$

ossia, con i valori $\rho_C = 1/4$ e $\rho_S = 1/2$,

$$\det M = \frac{3}{4} + \frac{1}{8}F_{CS} = \frac{1}{2} + \frac{1}{4}\sqrt{2} = \frac{1}{2}(1+\frac{1}{\sqrt{2}}) = \frac{1+\sqrt{2}}{2\sqrt{2}}\,.$$

Quindi

$$M^{-1} = \frac{1}{1-\rho_C+\rho_C(1-\rho_S)F_{CS}} \begin{pmatrix} 1-\rho_S F_{SS} & \rho_C F_{CS} \\ \rho_S F_{SC} & 1-\rho_C F_{CC} \end{pmatrix}$$

$$= \frac{1}{1-\rho_C+\rho_C(1-\rho_S)F_{CS}} \begin{pmatrix} 1 & \rho_C F_{CS} \\ \rho_S & 1-\rho_C(1-F_{CS}) \end{pmatrix}$$

$$= \frac{2\sqrt{2}}{1+\sqrt{2}} \begin{pmatrix} 1 & \frac{2-\sqrt{2}}{2\sqrt{2}} \\ \frac{1}{2} & \frac{\sqrt{2}+4}{4\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{2\sqrt{2}}{1+\sqrt{2}} & \frac{2-\sqrt{2}}{1+\sqrt{2}} \\ \frac{\sqrt{2}}{1+\sqrt{2}} & \frac{4+\sqrt{2}}{2+2\sqrt{2}} \end{pmatrix}.$$

La sorgente di luce a potenza 1 sta nel primo elemento, quindi il vettore della potenza creata è il primo vettore della base canonica

$$\mathbf{e} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

e pertanto il vettore delle radiosità è la prima colonna di M^{-1} :

$$\mathbf{b} = M^{-1}\mathbf{e} = \begin{pmatrix} \frac{2\sqrt{2}}{1+\sqrt{2}} \\ \frac{\sqrt{2}}{1+\sqrt{2}} \end{pmatrix} \approx \begin{pmatrix} 1.1716 \\ 0.5858 \end{pmatrix}.$$

Parte (vi). Come già osservato precedentemente negli esercizi della Sezione *Esercizi sula calcolo analitico dei fattori di forma* più sopra in questo Capitolo, il fattore di forma differenziale dal vertice del cono non cambia affatto, perché qualunque sia la forma del pavimento esso copre sempre, visto dal vertice, lo stesso angolo solido, che è l'angolo solido di apertura del cono. Questo è conseguenza del fatto che ogni semiretta interna al cono uscente dal vertice incontra un punto del pavimento ma non incontra mai un altro punto del cono: non ci sono quindi fattori di ostruzione. Si osservi però che la stessa cosa non è vera per le semirette uscenti da altri punti del cono: esse possono incontrare punti del cono invece che del pavimento. Se il pavimento è piatto o concavo (ossia incurvato verso l'esterno del volume) allora il volume risultante è convesso, e quindi le semirette uscenti da un vertice del cono ed indirizzate ad altri punti del cono non possono essere ostruite da punti del pavimento: ne risulta che, in questo caso, il fattore di forma differenziale *da un punto fissato del cono a sé stesso* rimane costante al variare della forma del pavimento, purché essa rimanga concava (naturalmente, questo fattore di forma differenziale ad un punto del cono può cambiare con la scelta del punto. Invece se, come nella situazione originale, il pavimento è convesso, allora alcune semirette da un punto del cono *diverso dal vertice* al cono stesso possono essere intercettate dal pavimento, e la percentuale di tali semiretta cambia al cambiare della superficie (ad esempio, se il pavimento è una superficie sferica rientrante dentro il cono, allora sempre più semirette uscenti da punti vicini alla circonferenza di base del cono vengono intercettate da scelte via via più curve del pavimento. Pertanto, in questo caso, per punti sufficientemente vicini alla base del cono il fattore di forma differenziale decresce al crescere della curvatura.

Parte (vii). I fattori di forma globali F_{CS} e $F_{CS'}$ sono definiti come le medie (di area) dei corrispondenti fattori di forma differenziali al variare del punto origine nell'area del cono. Da quanto visto alla parte precedente, i fattori di forma differenziali se il pavimento è concavo S' sono maggiori (se il punto origine è abbastanza vicino alla base del cono) dei fattori di forma differenziali nel caso del pavimento convesso S. Questo è vero anche se i pavimenti S e S' non sono sferici e riflessi speculari l'uno dell'altro. ma nel caso in esame, in cui essi, essendo riflessi speculari, hanno la stessa area, allora è facile raggiungere la stessa conclusione anche senza ricorrere ai fattori di forma differenziali. Infatti, adesso si ha $F_{SS} = 0$ per la convessità di S, ma $F_{S'S'} > 0$ per la concavità di

S'. Dalla conservazione dell'angolo solido totale ora segue $F_{SC} = 1$ ma $F_{S'C} < 1$. Dalla reciprocità ora abbiamo

$$F_{CS} = \frac{\text{Area}(S)}{\text{Area}(C)} F_{SC} = \frac{\text{Area}(S)}{\text{Area}(C)}$$

ma

$$F_{CS'} = \frac{\text{Area}(S')}{\text{Area}(C)} F_{S'C} < \frac{\text{Area}(S')}{\text{Area}(C)} = \frac{\text{Area}(S')}{\text{Area}(C)} = F_{CS}\,.$$

Quindi $F_{CS'} < F_{CS}$.

Parte (viii). Questi calcoli vengono lasciati per esercizio al lettore. □

ESERCIZIO 6.16.11. (*i*) Una scena consiste di una stanza cilindrica avente per base un disco intorno all'origine di raggio 1 ed altezza 2; il pavimento ed il soffitto hanno riflettività $\frac{1}{2}$ e la parete riflettività $\frac{1}{4}$. Sia α il fattore di forma dal soffitto al pavimento. Si possono determinare tutti gli altri fattori di forma in termini di α? Se no, spiegare perchè; se sì, scrivere la matrice della radiosità e cercare di specificare quale sia il range dei valori ammissibili per α.

(*ii*) Nella stessa stanza il pavimento viene sostituito da un cono retto che estrude dalla stanza (la quale quindi è convessa), di altezza pari al raggio. Rispondere alle domande precedenti. *(Primo suggerimento: se non si ricorda l'area del cono, la si può calcolare grazie alla formula del fattore di ingrandimento delle aree per una superficie parametrica del tipo $p(x, y) = (x, y, f(x, y))$, dove f è una funzione di due variabili. Il fattore di ingrandimento delle aree in tal caso risulta essere*

$$\sqrt{1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

Questo permette di calcolare l'area come integrale doppio.
Secondo suggerimento: visto che il cono estrude, la stanza rimane convessa. Quindi il pavimento non crea occlusioni nella visibilità relativa di ciascunao degli altri elementi, e pertanto gli angoli solidi sottesi ed i fattori di forma non cambiano. Neppure il fattore di forma da soffitto al pavimento conico e dalla parete laterale al pavimento conico cambiano: rimangono entrambi uguali a quelli che erano quando il pavimento era piatto. Le uniche cose che cambiano rospetto a quando il pavimento era piatto sono:

- *cambia il fattore di forma dal pavimento alla parete laterale ed al soffitto (perché cambiano i fattori di forma differenziali dai punti del pavimento conico alle altre due pareti, in quanto ora varia la distanza da ciascuno di questi punti alle pareti: in effetti a caus dell'aumento di questa distanza si vede che dal pavimento che estrude a parete laterale e soffitto i fattori di forma sono più piccoli di quando il pavimento era piatto);*
- *diventa non nullo il fattore di forma dal pavimento a sé stesso: quest'ultimo fatto ci dà un fattore di forma in più. Ma d'altra parte i fattori di forma da pavimento a parete e da pavimento a soffitto si calcolano per reciprocità una volta calcolata l'area del pavimento: quindi ciò che viene a diminuire in questi fattori di forma va a finire nel nuovo fattore di forma da pavimento a pavimento*

Segue da quanto appena detto che tutti i fattori di forma si calcolano a partire da uno di essi, diciamo ad esempio il fattore di forma a parete a pavimento, o da soffitto a pavimento, che rimangono esattamente come erano quando il pavimento era piatto.)

(*iii*) Ora nella stessa stanza il pavimento viene sostituito da un cono retto che intrude nella stanza (la quale quindi non è convessa), di altezza pari al raggio. Rispondere alle domande precedenti. *(Suggerimento: ora la stanza non è convessa. Il pavimento occlude la visibilità dalla parete laterale a sé stessa, e quindi cambia il fattore di forma dalla parete a sé stessa;*

però il pavimento non occlude la visibilità da parete a soffitto, ed il fattore di forma da soffitto a pavimento resta lo stesso delle domande precedenti, come quando il pavimento era piatto, perché anche in questo caso la visibilità non viene occlusa.)

□

ESERCIZIO 6.16.12. Nella stessa stanza cilindrica di raggio m, ora il pavimento P è parabolico, di equazione $z = f(x, y) = x^2 + y^2 - m^2$.

(*i*) Calcolare l'area di P.

(*ii*) Quanto vale il fattore di forma differenziale dal centro del soffitto al pavimento?

(*iii*) Quanto vale il fattore di forma differenziale dal centro del pavimento al soffitto?

(Suggerimento: la stanza è convessa, e quindi ogni raggio dal centro del soffitto (o anche ogni altro punto del soffitto) al pavimento interseca il disco di base del cilindro. Quindi il fattore di forma differenziale è lo stesso che se il pavimento fosse piatto.) □

ESERCIZIO 6.16.13. Si consideri la scena data dall'intercapedine fra una sfera esterna S_2 di raggio $r = \sqrt{2}$ ed una sfera interna concentrica S_1 di raggio 1.

(*a*) Calcolare tutti i fattori di forma della scena consistente di questi due elementi.

(*b*) Dimostrare che il fattore di forma differenziale σ da un punto della sfera esterna in (*a*) a tutta la sfera interna non dipende dal punto. Prendendo qualsiasi pezzo (misurabile!) della sfera esterna con area A, come si può quindi calcolare il fattore di forma dalla sfera interna a quel pezzo della sfera esterna, usando il risultato di (*a*)?

(*c*) Si calcoli il fattore di forma differenziale σ del punto precedente. Quanto varrebbe invece tale fattore di forma se la sfera esterna avesse un raggio tale che, quando guardata dai suoi punti, la sfera interna sottendesse un angolo solido di semiampiezza generica φ?

(*d*) Ora la sfera esterna è una softbox diffusiva, e quella interna un oggetto sferico da fotografare: una macchina fotografica penetra nella softbox e punta in direzione radiale (verso il centro), ma è piccola e possiamo trascurare il cambiamento dei fattori di forma che essa induce. La softbox è illuminata dall'esterno da una luce puntiforme ubicata a distanza $R > r$ dal centro, che quindi ne illumina una calotta S_2^+ (quella determinata dal cono luce con vertice nella sorgente e tangente a S_2). Supponiamo che la sorgente abbia potenza tale da far sì che la calotta illuminata di S_2^+ diffonda luce all'interno della scena al punto (*a*) con potenza per unità di area pari a 1 (quindi la potenza della lampada viene scelta come funzione di R). Scindiamo allora S_2 in due parti: la calotta illuminata ed il suo complemento S_2^-. Nel caso $R = 2$. si calcolino tutti i fattori di forma derivanti da questa scissione (la scena ora ha tre elementi) che è possibile ricavare dalle consuete proprietà geometriche di simmetria (conservazione dell'angolo solido totale e relazioni di reciprocità).

(*e*) Una calotta S_1^+ della sfera interna S_1 viene direttamente illuminata dalla calotta S_2^+: si tratta della calotta di S_1 direttamente visibile da S_2^+. Evidentemente, il fattore di forma da S_2^+ a S_1 coincide con il fattore di forma da S_2^+ a S_1^+, dal momento che il resto di S_1 (che indichiamo con S_1^-)non è visibile da S_2^+. Ora che anche la sfera interna è stata scissa come unione di S_1^+ e S_1^-, la scena consiste di 4 elementi.

(*f*) Ora facciamo variare r e R. Poniamo $\rho = R/r$. Rispondere se possibile alle seguenti domande, e se no indicare perché non è possibile.

(a) Il fattore di forma $F_{S_2^+ S_1^+}$ aumenta, diminuisce o rimane costante al crescere di ρ?

(b) Il fattore di forma $F_{S_2^+ S_2^-}$ aumenta, diminuisce o rimane costante al crescere di ρ?

(c) Il fattore di forma $F_{S_2^+ S_2}$ aumenta, diminuisce o rimane costante al crescere di ρ?

(d) Il fattore di forma $F_{S_2^+ S_2^+}$ aumenta, diminuisce o rimane costante al crescere di ρ?

(e) Il fattore di forma $F_{S_2^- S_2^+}$ aumenta, diminuisce o rimane costante al crescere di ρ?
(f) Il fattore di forma $F_{S_2^- S_2}$ aumenta, diminuisce o rimane costante al crescere di ρ?
(g) Il fattore di forma $F_{S_2^- S_2^-}$ aumenta, diminuisce o rimane costante al crescere di ρ?

Svolgimento.

(*a*) $F_{S_1 S_1} = 0$ perché S_1 è un corpo convesso. Pertanto $F_{S_1 S_2} = 1$ per la regola dell'angolo solido totale. D'altra parte, S_1 ha raggio 1, e S_2 ha raggio $r = \sqrt{2}$, quindi il rapporto delle loro aree è $A_2/A_1 = r^2 = 2$. Allora, per la regola di reciprocità, $F_{S_2 S_1} = F_{S_1 S_2} A_1/A_2 = 1/2$. Di nuovo per la regola dell'angolo solido totale,

$$F_{S_2 S_2} = 1 - F_{S_2 S_1} = 1 - \frac{1}{2} = \frac{1}{2}\,.$$

(*b*) Il fattore di forma *differenziale* $\sigma := F_{xS_1}$ da ogni punto x della sfera esterna a tutta la sfera interna non dipende dal punto x a causa dell'invarianza per rotazione dell'angolo solido sotteso dalla sfera interna rispetto ai punti della sfera esterna, dal momento che le due sfere sono concentriche. Pertanto, il fattore di forma $F_{S_A S_1}$ da un sottoinsieme misurabile S_A di area A di S_2 a tutto S_1, che è una media integrale dei fattori di forma differenziali, coincide con $\sigma := F_{xS_1}$. Poiché la sfera interna, di raggio 1, ha area 4π, otteniamo

$$F_{S_1 S_A} = \frac{A}{4\pi}\sigma\,. \tag{6.16.4}$$

(*c*) Calcoliamo l'apertura angolare con cui la sfera interna viene vista dai punti della sfera esterna. Scegliamo l'origine nel centro comune delle due sfere. Consideriamo, ad esempio, il punto $\boldsymbol{p} = (\sqrt{2}, 0, 0)$ della sfera esterna. Chiaramente il problema è invariante rispetto alle rotazioni assiali intorno al segmento dall'origine a $\boldsymbol{p}$, ossia all'asse x: pertanto limitiamo l'attenzione al sottospazio $\{y = 0\}$. Osserviamo che il punto sulla sfera interna che sta sulla bisettrice del primo quadrante è $\boldsymbol{q} = (1/\sqrt{2}, 1/\sqrt{2})$, ed il versore normale a S_1 in questo punto, essendo il versore radiale, coincide proprio con $\boldsymbol{q}$. Allora la retta tangente ha equazione $x + y =$ costante (ossia la retta è parallela alla antibisettrice, ossia alla bisettrice del secondo e quarto quadrante), ed imponendo che passi per $\mathbf{q}$ si trova che l'equazione è $x + y = \sqrt{2}$. Ma allora il punto $\boldsymbol{p} = (\sqrt{2}, 0)$ appartiene a questa retta! In altre parole, la retta tangente dal punto $\boldsymbol{p}$ alla sfera interna S_1 forma con l'asse x un angolo di $\pi/4$: questa è quindi la semiampiezza dell'angolo solido sotteso dalla sfera interna quando viene guardata dai punti di quella esterna.
Quindi il fattore di forma σ è lo stesso che avremmo se la sfera interna, vista da un punto x della sfera esterna, venisse rimpiazzata con un il disco che ha per bordo la circonferenza dei punti dove le rette uscenti da x sono tangenti a S_1 (si vedano gli Esercizi 1 e 2 nella precedente Sezione "Esercizi sul calcolo dei fattori di forma"). Questo disco sottende lo stesso angolo solido della sfera interna, e questo angolo vale $\pi/2$. Ma proprio nell'Esercizio 1 di quella Sezione abbiamo già visto che in tal caso il valore di σ è

$$\sigma = \sin^2\left(\frac{\pi}{4}\right) = \frac{1}{2}\,. \tag{6.16.5}$$

Se invece la sfera esterna avesse un raggio tale che i suoi punti vedessero la sfera interna con apertura angolare 2φ (quindi con deviazione angolare massima rispetto alla normale, ovvero semiampiezza, pari a φ), allora, sempre per il risultato dello stesso Esercizio, il fattore di forma σ varrebbe $\sin^2\varphi$.

(*d*) Ora la luce à a distanza $R = 2$ dal centro, ed il raggio della sfera esterna è $r = \sqrt{2}$, quindi $R/r = \sqrt{2}$. Lo stesso argomento del punto (*c*) mostra che le rette tangenti dalla posizione della luce alla sfera esterna hanno deviazione angolare $\pi/4$ rispetto alla direzione radiale, ed in particolare le rette tangenti nel sottospazio $\{y = 0\}$ hanno equazione $x \pm y = 2$. Pertanto

la semiampiezza di S_2^+ vista dalla posizione della luce è $\pi/4$, e nel sottospazio $\{y = 0\}$ i punti di tangenza, ossia i punti di bordo di S_2^+, hanno coordinate $x = 1$, $z = \pm 1$. Questo significa che vale la seguente **asserzione***: le rette uscenti da questi due punti e tangenti alle sommità superiore ed inferiore della sfera interna S_1, ossia rispettivamente ai punti* $(0, 1)$ *e* $(0, -1)$*, sono orizzontali.*
Ora calcoliamo l'area di S_2^+. Si tratta dell'area della calotta sferica di raggio $\sqrt{2}$ ed ampiezza angolare $\pi/2$, ossia latitudine che varia da $\pi/4$ a $\pi/2$, ovvero angolo ϕ di deviazione polare che varia da $\pi/4$ a 0. Visto che lo jacobiano delle coordinate sferiche è $r^2 \sin\phi$ e $r = \sqrt{2}$, l'area è

$$A_+ = r^2 \int_0^{2\pi} \int_0^{\frac{\pi}{4}} \sin\phi \, d\phi \, d\theta = -4\pi \cos\phi \big|_0^{\frac{\pi}{4}}$$
$$= 4\pi \left(1 - \cos\frac{\pi}{4}\right) = 2\pi(2 - \sqrt{2}) \, .$$

Allora, visto che l'area della sfera S_2 di raggio $r = \sqrt{2}$ è $4\pi r^2 = 8\pi$, vediamo che l'area A_- della calotta S_2^- è

$$A_- = 8\pi - 2\pi(2 - \sqrt{2}) = 2\pi(2 + \sqrt{2}) \, .$$

Ora, in base alle identità (6.16.5) del punto (c) e (6.16.4) del punto (b), abbiamo

$$F_{S_2^+ S_1} = \sigma = \frac{1}{2}$$
$$F_{S_2^- S_1} = \sigma$$
$$F_{S_1 S_2^+} = \frac{A_+}{4\pi}\sigma = \frac{1}{4}(2 - \sqrt{2})$$
$$F_{S_1 S_2^+} = \frac{A_-}{4\pi}\sigma = \frac{1}{4}(2 + \sqrt{2}) \, ,$$

e dalla regola dell'angolo solido totale

$$F_{S_2^+ S_2^+} + F_{S_2^+ S_2^-} + F_{S_2^+ S_1} = 1 \, .$$

Pertanto,

$$F_{S_2^+ S_2^-} = 1 - \sigma - F_{S_2^+ S_2^+} \, .$$

D'altra parte, la calotta S_2^+ ha punti estremi, nel sottospazio $\{y = 0\}$, dati da $(1, \pm 1)$, e quindi ascisse che variano da 1 al raggio $r = \sqrt{2}$, mentre la sfera interna ha ascisse da -1 a 1. Pertanto ogni punto della calotta S_2^+ vede ogni altro punto della stessa calotta, ossia non c'è alcun fattore di occlusione. Ne segue che il fattore di forma $F_{S_2^+ S_2^+}$ è lo stesso che avremmo se la sfera interna non ci fosse, ed in particolare, per la regola dell'angolo solido, vale $1 - F$, dove F è il fattore di forma da S_2^+ al disco D che chiude la calotta S_2^+. Questo fattore di forma non si ricava dalle properietà geometriche,e deve essere calcolato a partire dalla formula integrale: lasciamo il calcolo al lettore interessato.

(e) Per l'asserzione che abbiamo dimostrato al punto (d), le calotte S_1^+ e S_1^- sono i due emisferi di S_1 frontale ed opposto rispetto alla posizione della sorgente di luce, e S_2^+ vede S_1^+ ma non S_1^-, e simmetricamente per S_2^-. Pertanto $F_{S_2^+ S_1^-} = 0 = F_{S_2^- S_1^+}$; inoltre $F_{S_2^+ S_1^+} = F_{S_2^+ S_1} = 1/2$, in base al punto (c). Una volta calcolato tramite l'integrale il fattore di forma $F_{S_2^+ S_2^+}$ che mancava al punto precedente. I fattori di forma $F_{S_2^- S_1^-}$ e $F_{S_2^- S_1^+}$ devono essere calcolati tramite la formula integrale (questa volta S_1^- è una calotta più piccola della massima calotta

di S_1 visibile da S_2^-), ma la loro somma è $F_{S_2^- S_1} = 1/2$, quindi basta calcolarne uno (anche questo calcolo è lasciato al lettore interessato).

□

6.17. Numero dei fattori di forma ricavabili dalle relazioni di reversibilità e di normalizzazione

Una scena con n elementi dà luogo a n^2 fattori di forma, ma solo alcuni debbono essere calcolati analiticamente o numericamente tramite l'integrale che li definisce: gli altri possono essere ricavati da questi tramite le relazioni di reversibilità e di normalizzazione dell'angolo solido totale. Queste ultime sono associate a ciascun elemento i: la somma $\sum_j F_{ij}$ vale 1: così otteniamo n equazioni lineari. Le relazioni di reversibilità, $A_i F_{ij} = A_j F_{ji}$, sono n^2, ma per evitare equazioni banali o ripetizioni restringiamo l'attenzione a $i < j$, ottenendo così altre $n(n-1)/2$ equazioni lineari. Quindi le n^2 incognite F_{ij} soddisfano $n + \frac{n^2}{2} - \frac{n}{2} = \frac{n^2+n}{2}$ equazioni lineari. Il numero di fattori di forma da calcolare direttamente per via analitica (o numerica) è il numero di gradi di libertà delle soluzioni di questo sistema lineare, ossia la dimensione n^2 meno il rango del sistema, che è il numero di righe linearmente indipendenti. Calcolando questo rango sappiamo fin da principio quanti fattori di forma non sono ricavabili dalle suddette relazioni: ma c'è un aspetto ulteriore da considerare, che può ridurre il numero di fattori da calcolare, ed è la simmetria geometrica. infatti, i fattori di forma misurano gli angoli solidi coperti, e quindi sono invarianti per operazioni di rotazione o riflessione: se gli elementi della scena si trasformano l'uno nell'altro tramite operazioni di simmetria geometrica di questo genere, questo ci fornisce uguaglianze per i rispettivi fattori di forma. Ad esempio, se la scena consiste dell'interno di un tetraedro regolare (quattro triangoli equilateri identici), tutti i fattori di forma coincidono, e si ricavano quindi senza calcoli dalla condizione di normalizzazione (si veda l'Esempio 1 qui sotto); se invece consiste dell'interno di un cubo, ci sono solo due fattori di forma indipendenti (da una faccia a quella frontale e da una faccia ad una delle sue facce laterali), e le relazioni di normalizzazione si riducono ad una sola equazione a meno di indipendenza (mentre le relazioni di reversibilità sono già racchiuse nelll'invarianza sotto riflessione): quindi c'è un solo fattore di forma da calcolare (si veda l'Esempio 2 più oltre). Per motivi di spazio non calcoleremo quasi mai il rango di questo sistema di $(n^2+n)/2$ equazioni, ma invitiamo il lettore a farlo in ogni esercizio.

Parte 2

Illuminazione globale

CAPITOLO 7

Il trasporto della luce: flusso, radianza, riflettività bidirezionale ed equazione del rendering

L'obiettivo degli algoritmi di rendering è quello di creare immagini fotorealistiche. Per ogni pixel dell'immagine, questi algoritmi devono trovare l'oggetto della visibile in quel determinato pixel, e poi visualizzarlo con il colore appropriato. Si tratta quindi di algoritmi a precisione di immagine mirati a produrre la maggior possibile verisimiglianza, e quindi basati su modelli fisici del trasporto della luce (anche se, peril fine di ridurre la mole di calcoli, a volte scissi in diverse fasi separate e non sempre legabili insieme in maniera fisicamente corretta). Molti dei contenuti di questa parte sono tratti, o ispirati, dall'eccellente libro di testo **[3]** .

7.1. Quantità radiometriche

7.1.1. Flusso. La potenza radiante, spesso rappresenta con Φ, viene espressa in Watt (W=Joule/sec) e si chiama *flusso*. Il flusso esprime l'energia totale che passa da, a, o attraverso una superficie per unità di tempo. Il flusso non dipende dalla dimensione della sorgente di luce, o da quella dell'oggetto che la riceve, e neanche dalla distanza tra la sorgente e l'oggetto. Questa è considerata la grandezza radiometrica fondamentale, sulla base della quale sono definite tutte le altre.

7.1.2. Irradianza. L'irradianza, solitamente indicata con E, è il flusso radiante incidente su un punto x di una superficie per unità di area, e viene espressa in Watt/m^2:

$$E(x) = \frac{d\Phi}{d\sigma}(x)\,.$$

Qui il simbolo di derivata rispetto all'area significa che stiamo considerando dischi (o intorni aperti) D_r intorno ad un punto x di una superficie, ed il rapporto fra il flusso radiante che entra in D_r e l'area di D_r, e se ne calcola il limite per ottenere $E(x)$, che quindi è in effetti una derivata rispetto al raggio.

7.1.3. Emissione radiante o Radiosità. L'emissione radiante (M), detta anche *radiosità* (B), è il flusso radiante emesso da una sorgente estesa per unità di area, ed è espressa in W/m^2 (si noti che questa definizione non è esattamante uguale a quella del Capitolo precedente, dove il flusso poteva essere uscente o entrante, e quindi non c'era differenza fra irradianza e radiosità). Ossia:

$$M(x) = B(x) = \frac{d\Phi}{d\sigma}(x)\,.$$

Qui la derivata ha lo stesso significato che abbiamo illustrato sopra per la irradianza.

7.1.4. Radianza. La radianza è il flusso radiante emesso da una sorgente estesa per unità di angolo solido e per unità di area proiettata su un piano normale alla direzione considerata: quindi misura la distribuzione angolare della potenza uscente per unità di area. La radianza è una quantità a cinque dimensioni che varia con la posizione e la direzione ed è indicata con come $L = L(x, \mathbf{\Psi})$:

$$L(x,\, \mathbf{\Psi}) = \frac{d^2\Phi}{d\omega\; d\sigma^{\perp}}(x,\, \mathbf{\Psi}) = \frac{d^2\Phi}{d\omega\; d\sigma\; \cos\theta}(x,\, \mathbf{\Psi})\,. \tag{7.1.1}$$

Anche in questo caso la derivata seconda è un limite di quozienti il cui numeratore è il flusso radiante attraverso dischi centrati in un punto x ed angoli solidi con vertice in quel punto, ed il denominatore è l'area di tali dischi moltiplicata per l'ampiezza in steradianti degli angoli solidi.

La radianza può essere entrante o uscente. La notazione: $L(x \to \mathbf{\Theta})$ indica la radianza che parte dal punto x e va in direzione $\mathbf{\Theta}$, mentre $L(x \leftarrow \mathbf{\Theta})$ rappresenta la radianza in arrivo al punto x proveniente dalla direzione $\mathbf{\Theta}$.

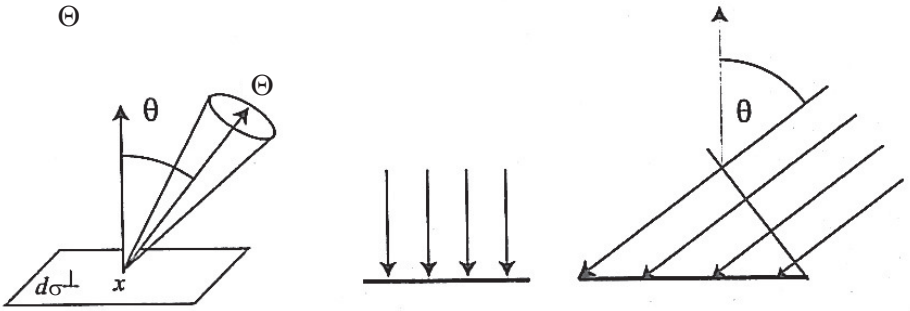


FIGURA 7.1.1. Definizione di radianza: il flusso radiante emesso da una sorgente estesa per unità di angolo solido

La radianza è la quantità più importante negli algoritmi di illuminazione globale, dato che essa esprime l'intensità con cui appaiono illuminati gli oggetti nella scena.

Si definisce anche la irradianza entrante ad un punto x da un angolo solido in direzione $\mathbf{\Psi}$ come il flusso entrante entro quell'angolo solido per unità di area intorno a x (non di area proiettata, in questo caso):

$$E(x \leftarrow \mathbf{\Psi}) = \frac{d^2\Phi}{d\omega\, d\sigma}(x)\,.$$

Quindi, in particolare,

$$dE(x \leftarrow \mathbf{\Psi}) = L(x \leftarrow \mathbf{\Psi})\,\langle \boldsymbol{N}_x, \mathbf{\Psi}\rangle\, d\omega(\mathbf{\Psi})\,.$$

7.2. Relazione tra le quantità radiometriche

In base alla definizione della quantità radiometrica data prima, abbiamo le seguenti relazioni:

$$\begin{aligned}
\Phi(\sigma) &= \int_\sigma \int_\Omega L(x \to \mathbf{\Theta})\,\cos\theta\, d\omega(\mathbf{\Theta})\, d\sigma(x)\\
E(x) &= \int_\Omega L(x \leftarrow \mathbf{\Theta})\,\cos\theta\, d\omega(\mathbf{\Theta})\\
B(x) &= \int_\Omega L(x \to \mathbf{\Theta})\,\cos\theta\, d\omega(\mathbf{\Theta})\,.
\end{aligned}$$

In queste formule, il flusso si intende calcolato su una superficie σ, e gli integrali sono al variare dell'angolo $\mathbf{\Theta}$ sul semispazio $\Omega = \Omega_x$ frontale a x.

7.2.1. Dipendenza dalla lunghezza d'onda. Le misure radiometriche e le quantità descritte precedentemente non dipendono sono solo dalla posizione e dalla direzione, ma anche dipendenti dalla lunghezza d'onda e dall'energia della luce. Se si specifica esplicitamente la dipendenza dalla lunghezza d'onda, ad esempio della radianza, la corrispondente quantità radiometrica si chiama radianza spettrale. La radianza viene allora calcolata integrando la radianza spettrale sull'intervallo di lunghezze d'onda della luce visibile. Ciò che noi vediamo è una qualche media dei dati spettrali, del tipo

$$L(x \to \mathbf{\Theta}) = \int_{\text{spettro}} L(x \to \mathbf{\Theta},\, \lambda)\, d\lambda$$

La dipendenza della lunghezza d'onda dai valori radiometrici è spesso iincorporata implicitamente nell'equazione di illuminazione globale ma non viene menzionata esplicitamente, anche se poi occorre farlo quando si svolgono i calcoli o quando si sviluppa il codice di un applicativo di rendering.

7.3. Proprietà della radianza

La radianza è una quantità radiometrica fondamentale ai fini del rendering. Le altre quantità radiometriche, il flusso, l'irradianza e la radiosità, possono essere derivate dalla radianza. Enunciamo e dimostriamo alcune proprietà della radianza.

7.3.1. Reversibilità. La radianza non varia lungo percorsi rettilinei, purché la luce viaggi nel vuoto, cioè senza diffusione o assorbimento da parte di alcun mezzo. La proprietà dell'invarianza della radianza equivale alla seguente espressione:

$$L(x \to y) = L(y \leftarrow x)\,,$$

che afferma che la radianza che lascia il punto x diretta verso il punto y è uguale alla radianza che arriva al punto y dal punto x. Illustriamo questo in Figura 7.3.1. Dalla definizione di radianza, la potenza totale che parte l'area $d\sigma(x)$ ed arriva all'area $d\sigma(y)$ può essere scritta come segue:

$$L(x \to y) = \frac{d^2\Phi}{(d\sigma(x)\, \cos\theta_x)\, d\omega(x \leftarrow\ d\sigma(y))}\,,$$

$$d^2\Phi = L(x \to y)\, \cos\theta_x\, d\omega(x \leftarrow\ d\sigma(y))\, d\sigma(x)\,,$$

dove $d\boldsymbol{\omega}(x \leftarrow\ d\sigma(y))$ è l'angolo solido sotteso da $d\sigma(y)$, visto da x. La potenza che arriva all'area $d\sigma(y)$ dall'area $d\sigma(x)$, può essere espressa in maniera simile:

$$L(y \leftarrow x) = \frac{d^2\Phi}{(\cos\theta_y\, d\sigma(y))\, d\omega(y \leftarrow\ d\sigma(x))}$$

$$d^2\Phi = L(y \leftarrow x)\, \cos\theta_y\, d\omega(y \leftarrow\ d\sigma(x))\, d\sigma(y)\,.$$

I differenziali degli angoli solidi sono:

$$d\omega_{x \leftarrow \sigma(y)} = \frac{\cos\theta_y\, d\sigma(y)}{r_{xy}^2}\,, \tag{7.3.1}$$

$$d\omega_{y \leftarrow \sigma(x)} = \frac{\cos\theta_x\, d\sigma(x)}{r_{xy}^2}\,.$$

Assumiamo che non ci siano mezzi attraverso i quali si trasmette la luce in arrivo dall'area $d\sigma(y)$, ossia che le due superficie siano nel vuoto: quindi non c'è perdita di energia dovuta alla diffusione

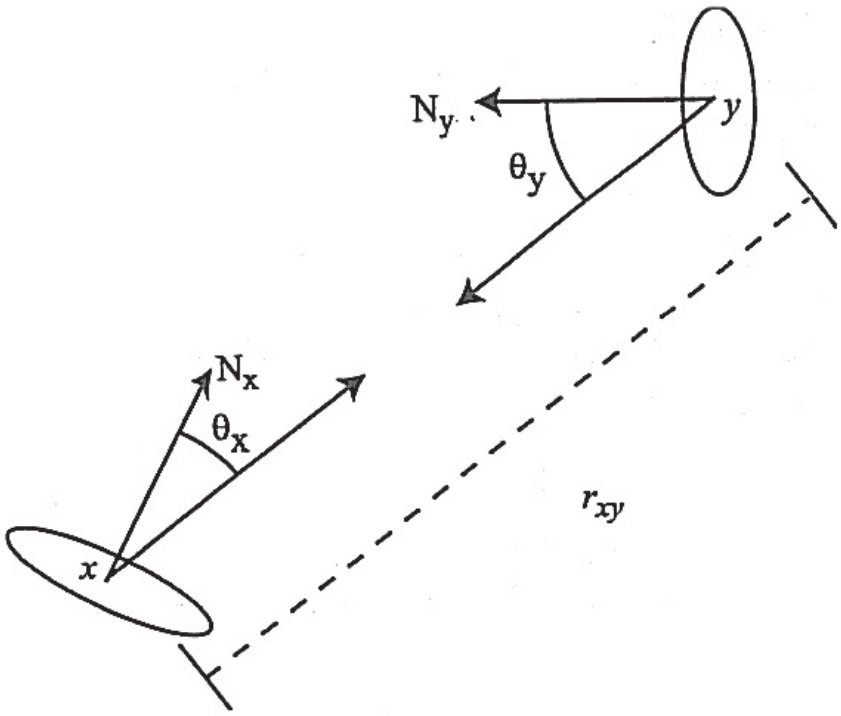


FIGURA 7.3.1. Invarianza della radianza lungo i raggi

ambientale. Allora, per la legge della conservazione dell'energia, tutta l'energia uscente da $d\sigma(x)$ nella direzione della superficie $d\sigma(y)$ deve arrivare a destinazione:

$$L(x \to y)\, \cos\theta_x\, d\omega(x \leftarrow d\sigma(y))\, d\sigma(x) \\ = L(y \leftarrow x)\, \cos\theta_y\, d\omega(y \leftarrow d\sigma(x))\, d\sigma(y)\,,$$

ovvero

$$L(x \to y)\, \cos\theta_x\, \frac{\cos\theta_y\, d\sigma(y)}{r_{xy}^2} = L(y \leftarrow x)\, \cos\theta_y\, \frac{\cos\theta_x\, d\sigma(x)}{r_{xy}^2}\,.$$

Quindi si ottiene

$$L(x \to y) = L(y \leftarrow x)\,.$$

Ripetiamo: la radianza non varia lungo un tragitto rettilineo, e non si attenua con la distanza, grazie all'ipotesi che non ci siano mezzi di propagazione che possono assorbire o deviare l'energia tra le due superficie.
Da questa osservazione segue che, se la radianza incidente o uscente è nota per tutti i punti della superficie, allora la distribuzione della radianza è conosciuta anche per tutti i punti nello spazio tridimensionale. Quasi tutti gli algoritmi usati nella Illuminazione Globale si limitano a calcolare i valori di radianza sui punti alla superficie (assumendo ancora la propagazione nel vuoto).

7.3.2. Rilevanza per il rendering. La radianza è la grandezza a cui rispondono i sensori, come ad esempio le videocamere ed il sistema visivo umano. La risposta dei sensori è proporzionale alla radianza incidente: le costanti di proporzionalità dipendono dalla geometria del sensore.

È chiaro quindi che la radianza è la grandezza che gli algoritmi di Illuminazione Globale devono calcolare.

7.4. Esempi

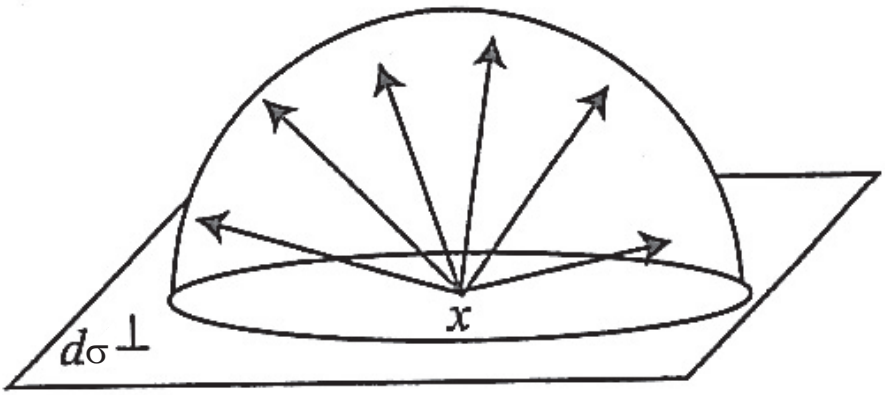


FIGURA 7.4.1. Emettitore diffusivo

7.4.1. Emettitore diffusivo. Consideriamo l'esempio di un emettitore puramente diffusivo σ (con abuso di notazione indichiamo con lo stesso simbolo σ anche l'area di σ). Per definizione, esso emette uguale radianza in tutte le direzioni da ogni punto della sua superficie (come in Figura 7.4.1).

$$L(x \to \mathbf{\Theta}) = L$$

In maniera analoga alla Nota (6.3.1), calcoliamo la potenza emessa dall'emettitore diffusivo:

$$\begin{aligned}
\Phi &= \int_\sigma \int_\Omega L(x \to \mathbf{\Theta}) \cos\theta \, d\omega(\mathbf{\Theta}) \, d\sigma(x) \\
&= \int_\sigma \int_\Omega L \cos\theta \, d\omega(\mathbf{\Theta}) \, d\sigma(x) \\
&= L \left(\int_\sigma d\sigma(x) \right) \left(\int_\Omega \cos\theta \, d\omega(\mathbf{\Theta}) \right) \\
&= L\sigma \int_0^{2\pi} d\phi \int_0^{\frac{\pi}{2}} \cos\theta \, \sin\theta \, d\theta \\
&= 2\pi L\sigma \int_0^{\frac{\pi}{2}} \frac{1}{2} \sin 2\theta \, d\theta \\
&= \frac{1}{2}\pi L\sigma \int_0^{\pi} \sin u \, du \\
&= \pi L\sigma \, ,
\end{aligned}$$

dove σ è l'area dell'emettitore diffusivo. Dall'identità precedente otteniamo la seguente relazione tra potenza, radianza e radiosità di una superficie diffusiva:

$$\Phi = L\sigma\pi = B\sigma\,. \tag{7.4.1}$$

7.4.2. Un emettitore non diffusivo. Consideriamo una sorgente luminosa di area 10x10 cm^2, ogni punto su questa sorgente emette radianza secondo la seguente distribuzione:

$$L(x \to \mathbf{\Theta}) = 6000\cos\theta \qquad (\text{unità:}\quad W/sr\cdot m^2)$$

Ricordiamo che la funzione della radianza è definita per tutte le direzioni nell'emisfero, e tutti i punti sulla superficie. Questa particolare distribuzione è la stessa per tutti i punti nelle sorgenti luminose; comunque, per ogni punto della superficie, c'è un decadimento. La radiosità per ogni punto può essere calcolata come segue:

$$\begin{aligned}
B &= \int_\Omega L(x \to \mathbf{\Theta})\,\cos\theta\, d\omega(\mathbf{\Theta}) \\
&= \int_\Omega 6000\,\cos^2\theta\, d\omega(\mathbf{\Theta}) \\
&= 6000\int_0^{2\pi}\int_0^{\pi/2}\cos^2\theta\sin\theta\, d\theta\, d\phi \\
&= 6000\cdot 2\pi\cdot\left[\tfrac{-\cos^3\theta}{3}\right]_0^{\pi/2} \\
&= 4000\pi\, W/m^2 \\
&= 12566\, W/m^2\,.
\end{aligned}$$

La potenza emessa dall'intera sorgente luminosa può essere calcolata come segue:

$$\begin{aligned}
\Phi &= \int_\sigma\int_\Omega L(x \to \mathbf{\Theta})\cos\theta\, d\omega(\mathbf{\Theta})\, d\sigma(x) \\
&= \int_\sigma (\int_\Omega L\cos\theta\, d\omega(\mathbf{\Theta}))\, d\sigma(x) \\
&= \int_\sigma B(x)\, d\sigma(x) \\
&= 4000\pi\, W/m^2\cdot 0.1\cdot 0.1\, m \\
&= 125.66\, W\,.
\end{aligned}$$

7.5. Emissione della luce

Ci sono vari tipi di sorgenti di luce, per esempio una sorgente come il sole, o altre che emettono in base ad effetti quantistici come nel caso della fluorescenza, dove i materiali assorbono energia a qualche lunghezza d'onda e la riemettono a qualche altra. Come detto prima, per la Computer Graphics non serve una spiegazione dettagliata della luce al livello di meccanica quantistica. Nella maggior parte degli algoritmi di rendering si assume che la luce sia emessa da una sorgente con un dato range di lunghezze d'onda e con una data intensità al variare della lunghezza d'onda (spesso specificata, in maniera approssimata, per le sole componenti primarie rossa, verde e blu). Il calcolo accurato della Illuminazione Globale richiede di specificare le seguenti tre distribuzioni per ogni sorgente di luce: spaziale, direzionale e di intensità spettrale. Le luci puntiformi sono in realtà approssimazioni di sorgenti con estensione piccola, e quindi l'approssimazione puntiforme porta ad una distribuzione spaziale non completamente accurata: le sorgenti di luce estese sono più realistiche. Le distribuzioni

direzionali di tipiche sorgenti di luce sono di solito determinate dalla forma della sorgente stessa (approssimata come un integrale di sorgenti puntiformi isotrope).
Sebbene la distribuzione spettrale della luce possa essere sintetizzata con precisione, motivi di efficienza gli algoritimi di Illuminazione Globale tipicamente la sintetizzano in RGB (o in uno spazio di colore analogo, a tre parametri).

7.6. Interazione della luce con le superfici

L'energia luminosa emessa in una scena interagisce con i diversi oggetti della scena, tramite riflessioni o trasmissioni. Una parte dell'energia luminosa potrebbe essere assorbita dalle superficie e dissipata sotto forma di calore, ma questo fenomeno solitamente non è considerato negli algoritmi di rendering.

7.6.1. BRDF. I materiali interagiscono con la luce in modi differenti, e l'aspetto di materiali diversi appare diverso con le stesse condizioni di luce. Alcuni materiali sono speculari; altri sono diffusivi. L'aspetto dipende anche dalle proprietà rifrattive: gli oggetti semitrasparenti appaiono diversi da quelli opachi. Assumiamo che la luce incidente ad una superficie esca con la stessa lunghezza d'onda con cui era arrivata, ed allo stesso istante: ovvero, trascuriamo effetti come la fluorescenza e la fosforescenza. In molte situazioni fisiche, la luce arriva ad una superficie in un punto x da una direzione di incidenza $\mathbf{\Theta}_i$, e poi riparte dalla superficie da un altro punto y in una direzione $\mathbf{\Theta}_r$. La funzione che definisce la percentuale di luce incidente a x dalla direzione $\mathbf{\Theta}_i$ che viene riflessa o rifratta a y in direzione $\mathbf{\Theta}_r$ si chiama *Bidirectional Scattering Surface Reflectance Distribution Function* (BSSRDF). Facciamo l'ipotesi semplificatrice che la luce incidente in un certo punto esce allo stesso punto: quindi non trattiamo lo scattering interno agli oggetti, che provoca l'uscita di luce da un punto diverso.

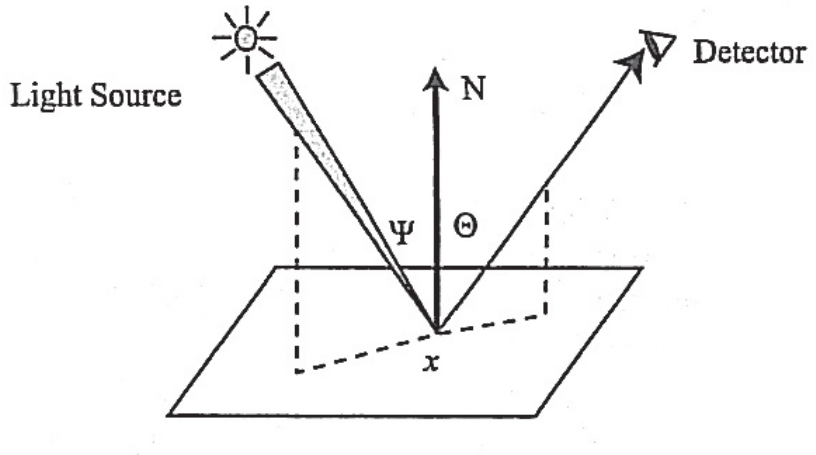


FIGURA 7.6.1. Bidirectional Reflectance Distribution Function

Le proprietà di riflettività di una superficie sono descritte da una funzione chiamata *Bidirectional Reflectance Distribution Function* (BRDF). La BRDF nel punto x è definita come la frazione della radianza differenziale incidente da una direzione $\mathbf{\Psi}$ che viene riflessa in una direzione $\mathbf{\Theta}$ (o più precisamente, la derivata della radianza riflessa rispetto a quella incidente). Essa si indica con $f_r(x, \mathbf{\Psi} \to \mathbf{\Theta})$:

$$f_r(x, \boldsymbol{\Psi} \to \boldsymbol{\Theta}) = \frac{dL(x \to \boldsymbol{\Theta})}{dE(x \leftarrow \boldsymbol{\Psi})}$$

$$= \frac{dL(x \to \boldsymbol{\Theta})}{L(x \leftarrow \boldsymbol{\Psi}) \, \langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle \, d\omega(\boldsymbol{\Psi})} \,, \tag{7.6.1}$$

dove $\langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle = \cos(\boldsymbol{N}_x, \boldsymbol{\Psi})$ è il coseno dell'angolo formato dal vettore normale $\boldsymbol{N}_x$ alla superficie nel punto x e la direzione del vettore incidente $\boldsymbol{\Psi}$.
Si osservi che in generale la BRDF è definita per tutte le direzioni (non solo per quelle nell'emisfero frontale): naturalmente, nel caso di corpi opachi, è nulla nelle direzioni interne (ossia fuori dell'emisfero frontale) ma non per le superficie semitrasparenti. Per abbracciare sia le direzioni trasparenti sia quelle riflettenti a volte si usa il termine *Bidirectional Scattering Distribution Function* (BSDF).

7.6.2. Proprietà della BRDF.

7.6.2.1. *Range.* La BRDF può assumere qualsiasi valore positivo, e può variare con la lunghezza d'onda.

7.6.2.2. *Dimensione.* La BRDF è una funzione a quattro dimensioni definita da ogni punto sulla superficie; due dimensioni corrispondono alle direzioni in arrivo, e due alle direzioni in uscita. Talvolta la BRDF è anisotropa (il valore di f_r cambia se la superficie viene ruotata attorno alla direzione normale), ma molti materiali sono isotropi.

7.6.2.3. *Reciprocità.* Il valore della BRDF rimane invariato se scambiamo fra loro le direzioni di ingresso ed uscita. Questa proprietà si chiama reciprocità di Helmholtz. In formule:

$$f_r(x, \boldsymbol{\Psi} \to \boldsymbol{\Theta}) = f_r(x, \boldsymbol{\Theta} \to \boldsymbol{\Psi})$$

Pertanto, per indicare che le due direzioni possono essere scambiate, usiamo la seguente notazione:

$$f_r(x, \boldsymbol{\Theta} \leftrightarrow \boldsymbol{\Psi})$$

7.6.2.4. *Relazione tra radianza incidente e riflessa.* Il valore della BRDF per una specifica direzione incidente non dipende dalla irradianza lungo altri angoli incidenti: la BRDF è lineare rispetto alle direzioni incidenti. Pertanto, data una figura opaca non emissiva, per ottenere la radianza riflessa totale ad un suo punto x si deve integrare sull'emisfero frontale a x la radianza entrante usando come peso la BRDF, come segue:

$$dL(x \to \boldsymbol{\Theta}) = f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) \, dE(x \leftarrow \boldsymbol{\Psi}) \; d\omega(\boldsymbol{\Psi})$$

$$L(x \to \boldsymbol{\Theta}) = \int_{\Omega_x} f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) \, dE(x \leftarrow \boldsymbol{\Psi})$$

e, da (7.6.1),

$$L(x \to \boldsymbol{\Theta}) = \int_{\Omega_x} f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) \, L(x \leftarrow \boldsymbol{\Psi}) \, \langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle \, d\omega(\boldsymbol{\Psi}) \,. \tag{7.6.2}$$

7.6.2.5. *Conservazione dell'energia.* La legge di conservazione dell'energia richiede che l'ammontare totale dell'energia riflessa su tutte le direzioni deve essere inferiore o uguale alla quantità di energia incidente sulla superficie, l'eccesso di energia viene poi trasformato in calore o altre forme di energia. Per ogni distribuzione di radianza incidente $L(x \leftarrow \boldsymbol{\Psi})$ nell'emisfero, l'energia totale incidente per unità di superficie è l'irradianza totale dell'emisfero stesso:

$$E = \int_{\Omega_x} L(x \leftarrow \boldsymbol{\Psi}) \langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle \, d\omega(\boldsymbol{\Psi})$$

L'energia totale riflessa M è un integrale doppio nell'emisfero. Supponiamo di avere una distribuzione di radianza uscente da una superficie, l'energia totale per unità di superficie che esce da M, è:

$$M = \int_{\Omega_x} L(x \to \boldsymbol{\Theta}) \, \langle \boldsymbol{N}_x, \boldsymbol{\Theta} \rangle \, d\omega(\boldsymbol{\Theta}) \, .$$

Segue dalla definizione di BRDF che

$$dL(x \to \boldsymbol{\Theta}) = f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) L(x \leftarrow \boldsymbol{\Psi}) \langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle \, d\omega(\boldsymbol{\Theta})$$

Integrando questa equazione per ottenere il valore di $L(x \to \boldsymbol{\Theta})$ e combinandola con l'espressione di M abbiamo:

$$M = \int_{\Omega_x} \int_{\Omega_x} f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) L(x \to \boldsymbol{\Theta}) \langle \boldsymbol{N}_x, \boldsymbol{\Theta} \rangle \langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle \, d\omega(\boldsymbol{\Psi}) \, d\omega(\boldsymbol{\Theta})$$

La BRDF soddisfa il principio di conservazione dell'energia per riflessione in un punto della superficie se e solo se, per qualsiasi distribuzione angolare $L(x \leftarrow \boldsymbol{\Psi})$ della radianza incidente, vale la disuguaglianza $M \leqslant E$, ovvero:

$$\frac{\int_{\Omega_x} \int_{\Omega_x} f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) L(x \leftarrow \boldsymbol{\Psi}) \langle \boldsymbol{N}_x, \boldsymbol{\Theta} \rangle \langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle d\omega(\boldsymbol{\Psi}) \, d\omega(\boldsymbol{\Theta})}{\int_{\Omega_x} L(x \leftarrow \boldsymbol{\Psi}) \langle \boldsymbol{N}_x, \boldsymbol{\Psi} \rangle \, d\omega(\boldsymbol{\Psi})} \leqslant 1 \quad (7.6.3)$$

Questa disuguaglianza deve essere vera per qualsiasi distribuzione di radianza incidente. Mettiamoci nel caso in cui questa distribuzione angolare sia una delta di Dirac (ossia una distribuzione con supporto in una unica direzione), cosicché gli integrali scompaiono:

$$L(x \leftarrow \boldsymbol{\Psi}) = L_{in} \, \delta(\boldsymbol{\Psi} - \boldsymbol{\Theta}_0) \, .$$

Allora la disuguaglianza (7.6.3) può essere semplificata come segue: per ogni direzione $\boldsymbol{\Theta}_0$,

$$\int_{\Omega_x} f_r(x, \boldsymbol{\Theta}_0 \leftrightarrow \boldsymbol{\Theta}) \, \langle \boldsymbol{N}_x, \boldsymbol{\Theta} \rangle \, d\omega(\boldsymbol{\Theta}) \leqslant 1 \, .$$

Questa disuguaglianza è una condizione necessaria per la conservazione dell'energia: esprime la conservazione dell'energia solo per una specifica distribuzione di radianza incidente, quella puntiforme. Ma, grazie alla linearità della BRDF rispetto alla intensità ed alla direzione incidente, essa è anche una condizione sufficiente: le radianze incidenti da direzioni differenti non interferiscono, e quindi la radianza uscente per una data distribuzione di radianza incidenteè l'integrale delle radianze incidenti rispetto all'angolo di incidenza. In altre parole, la irradianza si può considerare come l'integrale di irradianze di una distribuzione angolare di sorgenti puntiformi sull'emisfero frontale. Allora la stessa disuguaglianza vale per per ogni integrale di valori di radianza incidente puntiforme: quindi per ogni distribuzione angolare. Però in modelli di riflessione in cui il valore della BRDF dipenda anche dalla luce in arrivo da altre direzioni occorre limitarsi alla disuguaglianza più generale (7.6.3).

Gli algoritmi di Illuminazione Globale spesso scelgono la BRDF in base a qualche modello empirico. Bisogna avere molta cura di accertarsi che questi modelli empirici siano una fisicamente sensati e plausibili: in particolare devono essere soddisfatti la conservazione dell'energia ed il principio di reciprocità di Helmoltz. Il principio di reciprocità di Helmoltz costituisce un vincolo particolarmente importante per gli algoritmi bidirezionali di Illuminazione Globale: questi algoritmi calcolano la distribuzione dell'energia della luce considerando sia percorsi a partire dalle fonti di luce, sia percorsi a partire dall'osservatore: si assume che il percorso della luce sia invertibile, e quindi è essenziale che l'espressione della BRDF soddisfai il principio di reciprocità di Helmoltz.

7.6.3. Esempi di BRDF. A seconda della natura della BRDF, l'aspetto di un materialesu cui rimbalza la luce può essere quello di una superficie diffusiva, uno specchio od una superficie lucida.

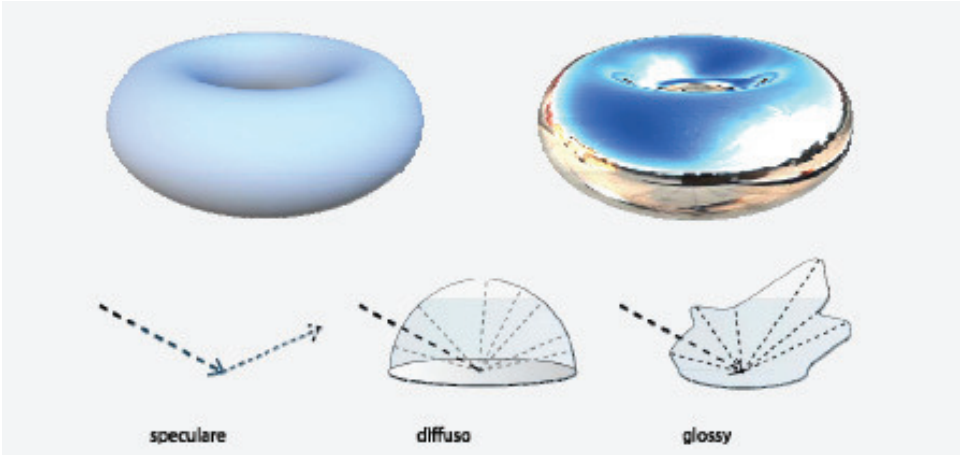


FIGURA 7.6.2. Tre diversi tipi di BRDF: (a) Materiale diffusivo ideale (la luce viene riflessa in tutte le direzioni); (b) Materiale speculare ideale (la luce viene riflessa in una sola direzione); (c) La BDRF *glossy* riflette la luce in base ad una combinazione di diffusione e riflessione speculare

Esaminiamo brevemente i tipi più comuni di BRDF.

7.7. Esempi di BRDF: modelli di riflessione e rifrazione

Mostriamo nelle prossime sezioni i tipici modelli di riflessione.

7.7.1. Superficie diffusive : Modello di Lambert. Il modello euristico più semplice per i materiali diffusivi, ovvero i materiali che possono riflettere la luce in tutte le direzioni dell'emisfero frontale, è attribuito a Lambert. In questo modello la diffusione è ideale in quanto la direzione di riflessione dei fotoni è distribuita uniformemente sull'emisfero:

$$f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) = k_d = \frac{\rho_d}{\pi}\,.$$

Il fattore π al denominatore è il fattore di normalizzazione per l'emisfero utilizzato per ottenere la radianza dalla riflettenza.

7.7.2. Superficie speculari. Superficie perfettamente speculari riflettono o rifrangono la luce solo in una direzione specifica.

7.7.3. Riflessione speculare. Per uno specchio ideale, la direzione della luce riflessa si calcola grazie alla legge della riflessione: la direzione incidente e uscente formano angoli uguali con la normale alla superficie e sono ad essa complanari. Se la direzione di incidenza si indica con $\boldsymbol{\Psi}$ e la normale alla superficie con $\boldsymbol{N}$, ne segue che la direzione della luce riflessa è data da (7.7.1):

$$\boldsymbol{R} = 2\langle \boldsymbol{N}, \boldsymbol{\Psi} \rangle\, \boldsymbol{N} - \boldsymbol{\Psi} \tag{7.7.1}$$

Un riflettore speculare ideale ha solo una direzione uscente per la quale la BRDF è diversa da 0: quindi la sua BRDF è una distribuzione δ di Dirac. I materiali reali non sono mai specchi ideali, ma solo approssimazioni. Nel programma questo calcolo viene seguito dalla funzione `reflect` all'interno

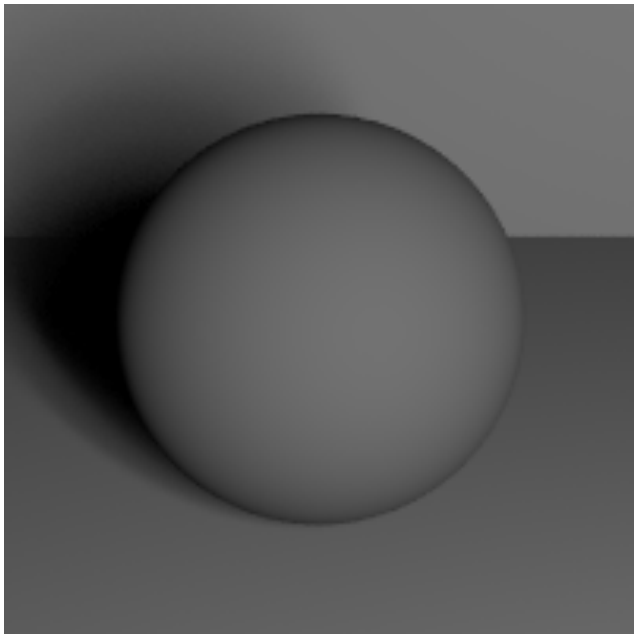


FIGURA 7.7.1. Materiale lambertiano diffusivo con $K_d = (0.3, 0.3, 0.3)$.

del file `utilities.h`.

```
<<utilities.h>>
```

```
// riflessione di un vettore i rispetto alla normale n
inline float3 reflect(float3 i, float3 n){ return ((n*2.0f)*n.dot(i)-i); }
```

7.7.4. Rifrazione. La direzione della rifrazione si calcola in base alla legge di Snell. Chiamiamo T la direzione lungo la quale si trasmette la luce proviente da un mezzo con indice di rifrazione η_1, che si rifrange in un mezzo con indice di rifrazione η_2. La legge di Snell è

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

dove θ_1 e θ_2 sono rispettivamente gli angoli che la normale alla superficie forma con il raggio incidente e il raggio trasmesso. Si noti che nel caso $\theta_1 = 0$, cioè quando il raggio risulta perpendicolare all'interfaccia, la soluzione è $\theta_2 = 0$ per qualunque valore di η_1 e η_2. In altri termini, un raggio che entra in un mezzo in direzione perpendicolare alla sua superficie non viene deviato. Allora, come visto in nella Sezione 4.1, la direzione $\boldsymbol{T}$ del raggio trasmesso è data da:

$$\begin{aligned}\boldsymbol{T} &= -\frac{\eta_1}{\eta_2}\boldsymbol{\Psi} + \boldsymbol{N}\left(\frac{\eta_1}{\eta_2}\cos\theta_1 - \sqrt{1-\left(\frac{\eta_1}{\eta_2}\right)^2(1-\cos^2\theta_1)}\right)\\ &= -\frac{\eta_1}{\eta_2}\boldsymbol{\Psi} + \boldsymbol{N}\left(\frac{\eta_1}{\eta_2}\langle \boldsymbol{N}, \boldsymbol{\Psi}\rangle - \sqrt{1-\left(\frac{\eta_1}{\eta_2}\right)^2(1-\langle \boldsymbol{N}, \boldsymbol{\Psi}\rangle^2)}\right), \qquad (7.7.2)\end{aligned}$$

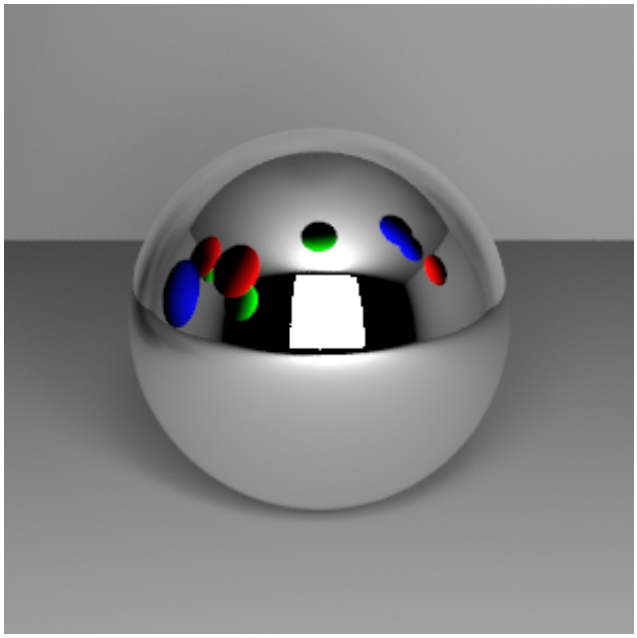


FIGURA 7.7.2. Materiale riflettente con $K_r = (1, 1, 1)$.

dove $\cos\theta_1 = \langle \boldsymbol{N}, \boldsymbol{\Psi} \rangle$ è il prodotto interno fra la normale e la direzione di provenienza. Quando la luce viaggia da un mezzo denso ad uno meno denso può subire una riflessione dentro il mezzo iniziale invece che rifrangersi nel mezzo meno denso. Questo processo è chiamato riflessione interna totale ed accade per tutti gli angoli di incidenza maggiori o uguali dell'angolo critico θ_c, per cui la luce viene rifratta nella direzione radente la superficie di separazione:

$$\eta_1 \sin\theta_c = \eta_2 \sin\frac{\pi}{2},$$

ossia

$$\sin\theta_c = \frac{\eta_2}{\eta_1}.$$

Si può ricavare la stessa condizione dall'identità 7.7.2: la riflessione interna totale si comincia a verificare appena il termine sotto la radice quadrata, $1 - \frac{\eta_1}{\eta_2})^2(1 - \langle \boldsymbol{N}, \boldsymbol{\Psi} \rangle^2)$, diventa minore di 0.

Nel progetto la rifrazione è implementata sia nella versione standard per la computer graphics, ovvero con indice di rifrazione unico per le 3 componenti RGB, sia nella versione con 3 raggi rifratti per ogni componente.

```
<<utilities.h>>
```

```
// rifrazione di un vettore i rispetto ad una normale n
// in questa funzione la rifrazione non varia con la lunghezza d'onda
inline bool refract(float3 &t, float3 i, float3 n, float ior){

    //ci si accerta che il vettore in entrata sia normalizzato
    i.norm();
    //si calcola il coseno tra la normale e il vettore entrante i: <n,i>
   float cos_theta_i=n.dot(i);
```

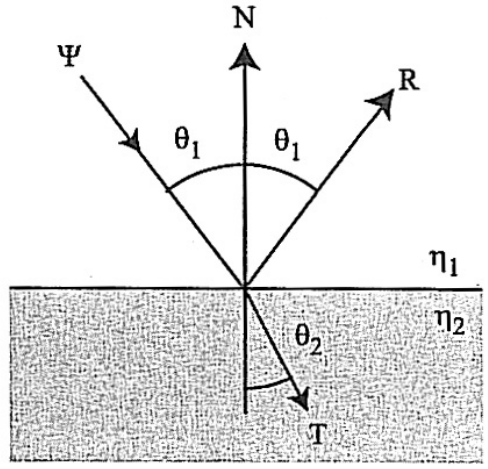


FIGURA 7.7.3. la riflessione e la rifrazione.

```
 //si prende in esame l'indice di rifrazione ior del materiale
//ior = indice incidente/indice trasmesso
//per semplificare il calcolo viene considerato solamente
//il passaggio dal vuoto al mezzo con indice di rifrazione ior
//non Ăš quindi possibile con questa funzione modellare il passaggio di luce
 //tra due materiali con indice di rifrazione differente
float eta=ior;

 //se il coseno dell'angolo tra il vettore i e n Ăš minore di 0 allora
 //si deve invertire la normale e l'indice di rifrazione
 //in questo caso infatti il passaggio di luce dovrĂ  avvenire dal mezzo con
    indice di rifrazione ior al vuoto
 if(cos_theta_i<0){
    cos_theta_i=-cos_theta_i;
    eta=1.0/eta;
    n=n*(-1);
 }

 //calcolo dei fattori necessari:
 float sin2_theta_i=1-(cos_theta_i*cos_theta_i);
 float sin2_theta_t= eta*eta*sin2_theta_i;

 // se questo coefficiente Ăš minore di 0 allora avviene una riflessione totale
 float K= 1-sin2_theta_t;
 if(K<0){

    //riflessione totale
    return false;

 }else{
    //altrimenti si procede con il calcolo del vettore rifratto
```

```
        float cos_theta_t= sqrtf(K);
        t= n*(eta*cos_theta_i-cos_theta_t)-i*eta;
        return true;
    }
}
```

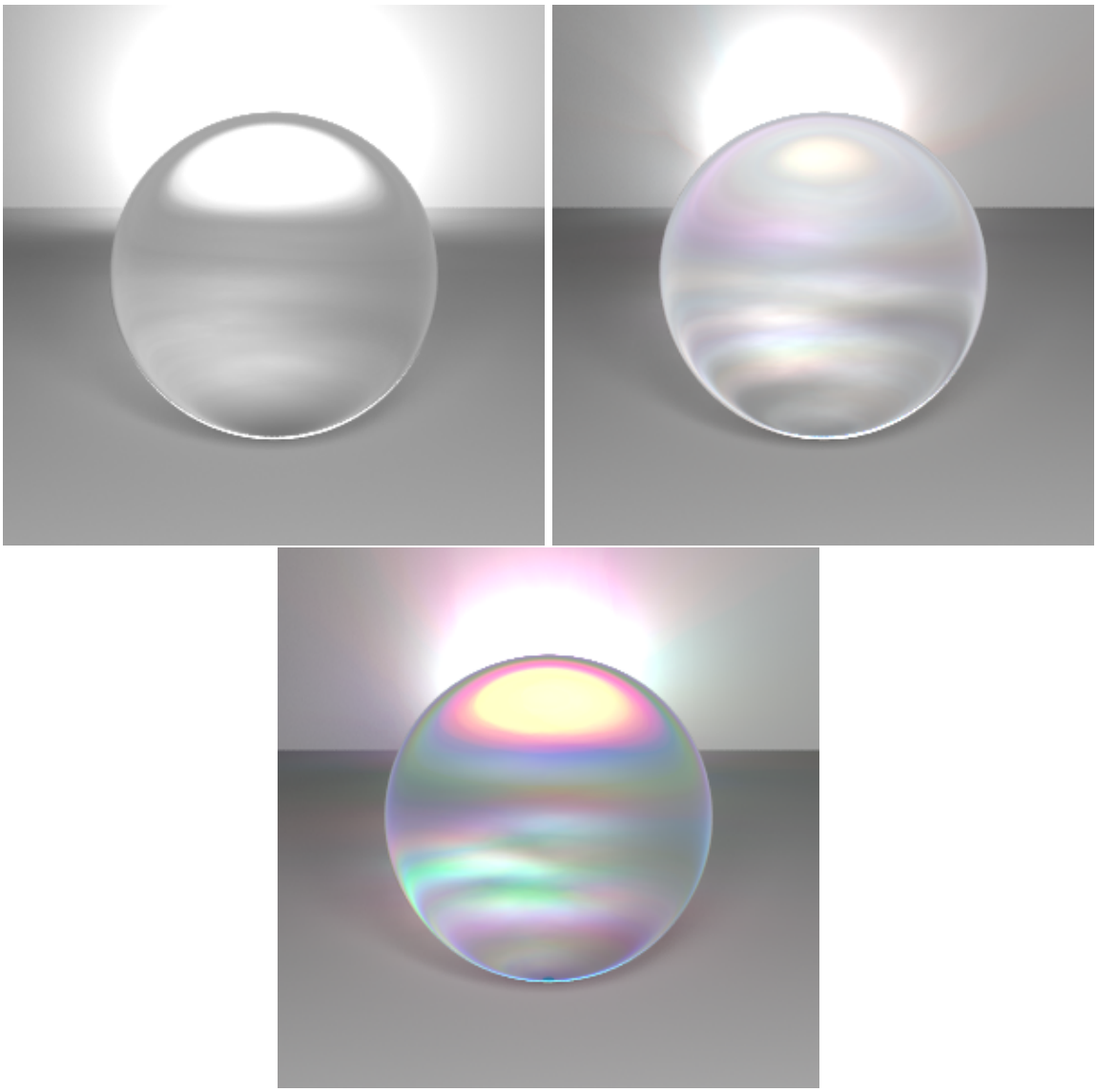


FIGURA 7.7.4. Materiale trasparente: in alto a sinistra con indice di rifrazione (1.05,1.05,1.05), in alto a destra con indice di rifrazione (1.05,1.0515,1.053), in basso con indice di rifrazione (1.03,1.04,1.05), per tutti Ăš stato posto $K_r = (1, 1, 1)$ e $K_g = (1, 1, 1)$.

7.7.4.1. *Reciprocità per superficie trasparenti.* Dobbiamo stare attenti quando consideriamo la BSDF di un oggetto trasparente: per esse non vale la reciprocità. In conseguenza alla legge della rifrazione di Snell, quando un fascio di luce entra in un mezzo denso provenendo da un mezzo meno

denso, viene piegato in modo da avvicinarsi alla direzione normale, e quindi la potenza luminosa per unità di area perpendicolare alla superficie (ossia la radianza) aumenta. Quando un fascio di luce parte da un mezzo denso e viene rifratto in un mezzo meno denso succede il contrario. In conseguenza della legge di Snell, la variazione della radianza è data dal quadrato del rapporto degli indici di rifrazione: $(\eta_2/\eta_1)^2$. Nel caso di superficie trasparenti occorre includere questo fattore.

7.7.5. Modello di Phong. Il modello euristico di illuminazione di Phong richiede solo prodotti interni, ovvero somme e moltiplicazioni, e quindi è implementabile in hardware, e per questo è molto usato. Come visto nella Sezione 2.4, la BRDF per il modello di Phong è

$$f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) = k_s \frac{\langle \boldsymbol{R}, \boldsymbol{\Theta} \rangle^n}{\langle \boldsymbol{N}, \boldsymbol{\Psi} \rangle} + k_d$$

dove R, il vettore riflesso, può essere calcolato dall'equazione 7.7.1.
La semplicità di questo modello è interessante, ma esso ha serie limitazioni: come osservato njella Sezione citata prima, esso non conserva l'energia, non soddisfa la proprietà di reciprocità di Helmoltz nello scambio delle direzioni $\boldsymbol{\Psi}$ e $\boldsymbol{\Theta}$, e non riproduce accuratamente il comportamento dei materiali reali.

7.7.6. Modello di Blinn–Phong. Il modello di Blinn–Phong riscrive l'espressione precedente in termini del vettore medio (*halfway vector*) $\boldsymbol{H}$ fra $\boldsymbol{\Psi}$ e $\boldsymbol{\Theta}$, come segue:

$$f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) = k_s \frac{\langle \boldsymbol{N}, \boldsymbol{H} \rangle^n}{\langle \boldsymbol{N}, \boldsymbol{\Psi} \rangle} + k_d$$

Questo elimina una delle cause della non conservazione dell'energia nel modello di Phong. Infatti, in tale modello compare il coseno dell'angolo fra la direzione dell'osservatore $\boldsymbol{\Theta}$ ed il versore della riflessione speculare della luce $\boldsymbol{R}$. Nel caso in cui luce ed osservatore sono in direzioni vicine e quasi radenti alla superficie (o almeno ad inclinazione maggiore di $\pi/4$), il versore dell'osservatore forma un angolo di più di $\pi/2$ con R, e quindi il coseno è negativo e viene rimpiazzato con zero (perché contributi negativi di illuminazione non hanno senso e non sono ammessi). Quindi in tali casi si perde un contributo di illuminazione che in un modello fisico e non euristico sarebbe positivo e tenuto in conto. Ma nel modello di Blinn–Phong l'angolo fra $\boldsymbol{\Theta}$ e $\boldsymbol{R}$ è rimpiazzato da quello fra $\boldsymbol{N}$ e $\boldsymbol{H}$, che è sempre inferiore a $\pi/2$, e quindi ha coseno positivo e non porta a troncamenti a zero che causano perdite di energia. Nondimeno, l'energia non si conserva ugualmente.

7.7.7. Modello di Blinn–Phong modificato. Il modello di Blinn–Phong modificato risolve il problema della perdita della reciprocità di Helmoltz grazie alla seguente formulazione euristica, che simmetrizza l'espressione rinunciando al denominatore (ma così facendo sottostima i contributi di sorgenti di luce ad alta deviazione, cioè quasi radente alla superficie illuminata):

$$f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) = k_s \langle \boldsymbol{N}, \boldsymbol{H} \rangle^n + k_d$$

7.7.8. Equazione di Fresnel. Le equazioni precedenti stabiliscono le direzioni di riflessione e rifrazione della luce incidente ad una superficie liscia ideale, ma non la quantità di energia luminosa che viene riflessa: a questo scopo sono necessarie le equazioni di Fresnel, che abbiamo presentato nella Sottosezione 2.6.4: ne riassumiamo qui il contenuto per comodità del lettore.

Quando la luce colpisce una superficie liscia, la percentuale di energia luminosa che viene riflessa o rifratta dipende dalla lunghezza d'onda della luce, dalla geometria della superficie e dalla direzione di incidenza. Chiamiamo rispettivamente r_p e r_s le percentuali di energia luminosa che viene riflessa

per le due componenti della luce polarizzata: in base alla legge di Fresnel, esse valgono

$$r_p = \frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}$$

$$r_s = \frac{\eta_1 \cos\theta_1 - \eta_2 \cos\theta_2}{\eta_1 \cos\theta_1 + \eta_2 \cos\theta_2},$$

dove η_1 e η_2 sono gli indici di rifrazione delle due superficie. Se la luce non è polarizzata, allora la percentuale di energia riflessa è

$$F = \frac{|r_p|^2 + |r_s|^2}{2}. \tag{7.7.3}$$

Bisogna sottolineare che queste equazioni si applicano sia per materiali conduttori (metalli) sia per dielettrici (non conduttori, quindi non metalli); per i metalli, l'indice di rifrazione diventa una variabile a valori complessi, $\eta + ik$, mentre per i dielettrici l'indice di rifrazione è un numero reale e $k = 0$. Il coefficiente k è l'esponente che determina il decadimento esponenziale all'aumentare della profondità dell'energia luminosa che entra nel metallo. L'equazione di Fresnel si basa sul principio che la luce è o riflessa o rifratta quando interagisce con una superficie puramente speculare o rifrattiva, ma non assorbita. Poiché non vi è alcun assorbimento di energia luminosa, la somma dei coefficienti di riflessione e rifrazione vale 1. Quindi se la quantità di luce riflessa è F, la quantità di luce rifratta è $1 - F$. Per i materiali conduttori una rapida approssimazione del coefficiente di Fresnel è data dalle equazioni:

$$r_p = \frac{(\eta^2 + k^2)\cos\theta_1^2 - 2\eta\cos\theta_1 + 1}{(\eta^2 + k^2)\cos\theta_1^2 + 2\eta\cos\theta_1 + 1}$$

$$r_s = \frac{(\eta^2 + k^2) - 2\eta\cos\theta_1 + \cos\theta_1^2}{(\eta^2 + k^2) + 2\eta\cos\theta_1 + \cos\theta_1^2}$$

Questa funzione ha bisogno solamente del coseno dell'angolo tra il raggio incidente e la normale alla superficie, che è il prodotto scalare dei due versori corrispondenti, e quindi è implementable con le sole operazioni aritmetiche di somma e moltiplicazione: per questo è rapida.

Nel codice qui di seguito, il fattore di Fresnel viene calcolato dalla funzione `getFresn` all'interno della struttura `Material`. In essa sono gestiti gli indici di rifrazione (`ior`) e i coefficienti di assorbimento (`k`) in 3 dimensioni, corrispondenti alle lunghezze d'onda RGB. Nel caso in cui si abbia riflessione totale interna, per una certa lunghezza d'onda ma non per le altre, si finge che non ci sia e solamente alla fine del calcolo essa viene ripristinata. Ovviamente la riflessione totale interna avviene quando il coefficiente di Fresnel è uguale ad 1.

```
<<geometry.h>>
<<Struct Material>>
```

```
//metodo che calcola il coefficiente di Fresnel in base al coseno dell'angolo di
    incidenza del raggio visuale con la normale
float3 getFresn(float cos_i){

    //viene calcolato solamente per materiali con indice di rifrazione maggiore di 0
    if(ior.max()>0){

    //calcolo del coefficiente di Fresnel
```

```
//si verifica che il materiale sia un conduttore o un dielettrico in base al
    parametro k
if(k.max()<=0){

//indice di rifrazione del vuoto
float3 etai= float3(1);

//ior materiale ( indice di rifrazione del materiale)
float3 etat=ior;

//si verifica che il coseno tra la normale e il raggio entrante nella superfice
    sia maggiore di 0
//se Ăš minore di 0 si considera il raggio come uscente dalla superfice e
    vengono quindi invertiti gli indici di rifrazione
if(cos_i<0){
    etai=etat;
    etat=float3(1);
    cos_i=-cos_i;
}

//calcolo del coefficienti di Fresnel:

float3 et= etai.div(etat);

float3 sint2= et.mult(et)*(1-cos_i*cos_i);

    //se l'espressione per il seno Ăš maggiore di 1 allora la radice Ăš
        negativa, di conseguenza si effettua una riflessione totale, ovvero il
        coefficiente di Fresnel deve essere posto uguale ad 1 (solo in questo
        caso infatti tutta la luce viene completamente riflessa). PoichĂš qui si
        considerano indici di rifrazioni differenti in base alla lunghezza
        d'onda se la riflessione totale non avviene per ogni lunghezza d'onda il
        metodo continua restringendo il seno in tutte le lunghezze d'onda in
        [0,1]. Una volta terminato si pongono i coefficienti di Fresnel, per le
        lunghezze d'onda da ripristinare, uguali ad 1.

    //riflessione totale per tutte le componenti RGB:
    if((sint2.x>1)&&(sint2.y>1)&&(sint2.z>1)){
        return float3(1.) ; //in questo caso il metodo viene interrotto
            immediatamente e si effettua la riflessione totale
    }
    // restrizione in [0,1] dei risultati:
    float3 sint2_=clamp3(sint2);
    //calcolo del coseno:
    float3 cos_t=sqrtf3(float3(1)-sint2_);


    //formula per materiali dielettrici:
    float3 Rparal=((etat*cos_i)-(etai.mult(cos_t)))
```

```
        .div((etat*cos_i)+(etai.mult(cos_t)));
        float3 Rperp=((etai*cos_i)-(etat.mult(cos_t)))
        .div((etai*cos_i)+(etat.mult(cos_t)));
        float3 result= (Rparal.mult(Rparal)+Rperp.mult(Rperp))*0.5;
        if(result.media()<EPS)result=0;

        //controllo della riflessione totale su ogni componente RGB
        if(sint2.x>1){
            result.x=1;
        }
        if(sint2.y>1){
            result.y=1;
        }
        if(sint2.z>1){
            result.z=1;
        }
            return result;
    }else{
            //formula per materiali conduttori:
            float3 tmp= ior.mult(ior)+ k.mult(k);
            float3 tmp2= tmp*cos_i*cos_i;
            float3 Rparal_2= (tmp2- (ior*2.f*cos_i)+1).div(tmp2+ (ior*2.f*cos_i)+1);
            float3 Rperp_2=(tmp-(ior*2.f*cos_i)+(cos_i*cos_i))
            .div(tmp+(ior*2.f*cos_i)+(cos_i*cos_i));
            float3 result= (Rparal_2+Rperp_2) * 0.5f;
            return result;
        }
    }else{
        //se l'indice di rifrazione Ăš nullo di default viene imposta la
            riflessione totale
        return float3(1);
}
```

Al momento del calcolo dell'illuminazione su superficie riflettenti o trasparenti sono richiamati, rispettivamente, i due metodi `S_BRDF` e `T_BRDF`. Essi calcolano quanta energia viene riflessa e quanta trasmessa, in base al coefficiente di Fresnel precedentemente calcolato. Nel programma si lascia, inoltre, la possibilitĂ di creare materiali riflettenti anche senza indice di rifrazione, semplicemente assegnando al materiale un coefficiente di riflettivitĂ K_r maggiore di 0. Infatti la funzione `getFresn`, quando il materiale non possiede un indice di rifrazione, restituisce 1, il che comporta una riflessione totale.

```
//BRDF per materiali riflettenti
float3 S_BRDF(float3 Fresn){

   return Fresn.mult(Kr);

}
```

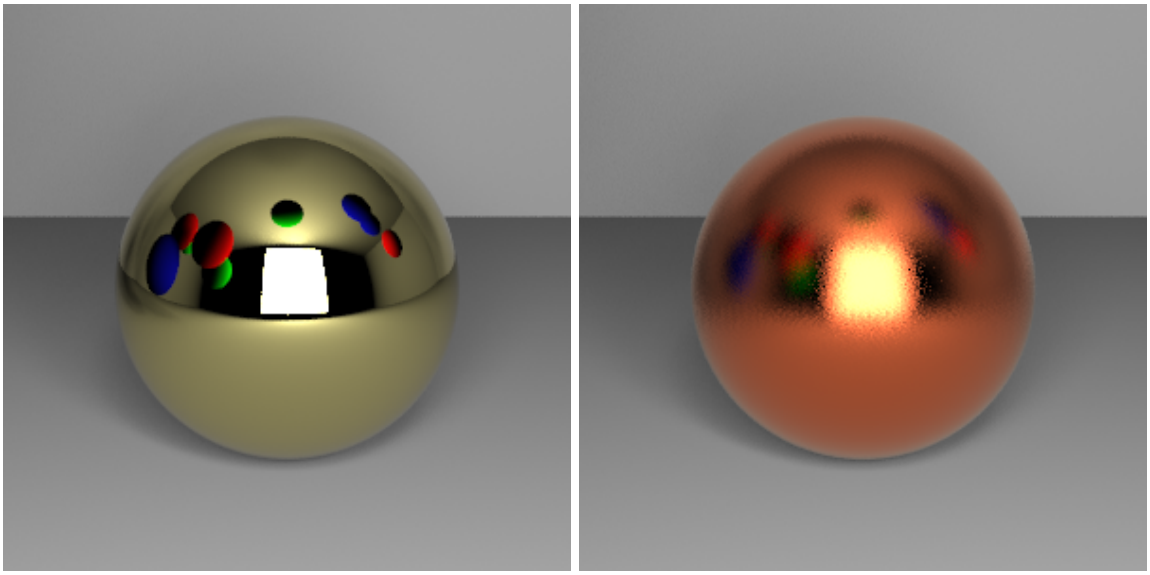


FIGURA 7.7.5. Materiali conduttori: da sinistra a destra, oro e rame. Per il rame ĂŠ stato aggiunto ll fattore `refImperfection`=0.01 utilizzando 100 campioni per la riflessione (vedi capitolo 13).

```
//BRDF per materiali trasparenti
float3 T_BRDF(float3 Fresn){

   return Kg.mult(float3(1)-Fresn);

    }
```

7.7.9. Superficie glossy: modello di Cook–Torrance. Il modello di Cook–Torrance (o meglio, il modello di Torrance-Sparrow) approssima le superficie reali come microsfaccettate, ossia costituite da un insieme di piccole sfaccettature planari speculari, la cui angolazione è casuale con una determinata distribuzione di probabilità, la cui varianza dipende da quanto diffusiva è globalmente la superficie. Ogni raggio incidente $\boldsymbol{\Psi}$ colpisce a caso una di queste sfaccettature piatte. Il modello deve stimare la luce che viene riflessa dalle microfacce in una certa direzione $\boldsymbol{\Theta}$ corrispondente alla posizione dell'osservatore. PoichĂ© le microfacce sono perfettamente speculari esse riflettono la luce proveniente da $\boldsymbol{\Psi}$ nella direzione $\boldsymbol{\Theta}$ solo se la loro normale è orientata come l'*halfway vector* $\boldsymbol{H}$ introdotto nella Sottosezione 7.7.6, ossia il vettore intermedio tra il raggio di entrata e il raggio di uscita:

$$\boldsymbol{H} = \frac{\boldsymbol{\Psi} + \boldsymbol{\Theta}}{||\boldsymbol{\Psi} + \boldsymbol{\Theta}||}$$

Ogni materiale ha la sua appropriata distribuzione $D(\boldsymbol{\Theta}_h)$ dalla quale otteniamo la probabilitĂ che una microfaccia abbia orientazione $\boldsymbol{H}$. Nel modello viene calcolato il risultato medio della riflessione della luce in base ai termini di riflessione e rifrazione di Fresnel sulle microfacce, alla distribuzione dell'inclinazione delle sfaccettature e ad un fattore termine geometrico di ombreggiatura. È più efficace scindere la distribuzione di probabilità come combinazione convessa di una componente che

decade abbastanza rapidamente all'aumentare dell'inclinazione (useremo la distribuzione di Beckmann, con decadimento fortemente esponenziale) e di un'altra componente assolutamente isotropa, ossia puramente diffusiva, con valore medio dato dal coefficiente diffusivo k_d. Il risultato è::

$$f_r(x, \boldsymbol{\Psi} \leftrightarrow \boldsymbol{\Theta}) = \frac{F(\beta)\, D(\boldsymbol{\Theta}_h)\, G}{\pi \langle \boldsymbol{N}, \boldsymbol{\Psi} \rangle \langle \boldsymbol{N}, \boldsymbol{\Theta} \rangle} + k_d \,,$$

dove i tre termini nella componente non diffusa della BRDF sono i coefficienti di riflessione di Fresnel F, la distribuzione delle sfaccettature D e il termine geometrico di ombreggiatura G. Ora spieghiamo ciascuno di questi termini. Il modello assume che la luce non sia polarizzata: pertanto, $F = \frac{|r_p|^2 + |r_s|^2}{2}$. Il termine di riflettenza di Fresnel è calcolato rispetto all'angolo β tra la direzione incidente e lo *halfway vector*: $\cos\beta = \langle \boldsymbol{\Psi}, H \rangle = \langle \boldsymbol{\Theta}, H \rangle$ (in effetti, in base alla definizione di *halfway vector*, questo angolo è lo stesso di quello tra la direzione riflessa uscente e lo *halfway vector*).
La funzione di distribuzione D specifica la distribuzione delle microsfaccettature per il materiale. Ci sono varie scelte per questa distribuzione di probabilità: una delle più comuni è la distribuzione di Beckmann,

$$D(\boldsymbol{\Theta}_h) = \frac{1}{m^2 \cos^4 \boldsymbol{\Theta}_h} e^{-\left(\frac{\tan \boldsymbol{\Theta}_h}{m}\right)^2}$$

dove $\boldsymbol{\Theta}_h$ è l'angolo tra la normale e lo *halfway vector* e $\cos \boldsymbol{\Theta}_h = \langle \boldsymbol{N}, \boldsymbol{H} \rangle$ e m (che chiameremo anche `slope`) ĂŠ lo scarto quadratico medio della pendenza delle microsfaccettature. Se m ĂŠ piccolo allora l'inclinazione delle microsfaccettature varia poco rispetto alla normale della superficie e quindi la riflessione ĂŠ molto a fuoco sulla direzione speculare. Invece, se m ĂŠ grande l'inclinazione ĂŠ elevata e la superficie ĂŠ ruvida: diffonde in maniera uniforme la luce riflessa su tutto l'emisfero frontale. Il termine geometrico G tiene in considerazione il fatto che rispetto alla sorgente o ad un'altra microsfaccettatura, ci siano microsfaccettature che ne coprono altre, occludendo la luce ad esse inviata dalla prima, o dalla sorgente. Questo fatto dĂ luogo ad un fattore di attenuazione geometrica G. Vengono considerate tre diverse situazioni:

- G_1: La luce incidente sulla microsfaccettatura ĂŠ totalmente riflessa e non colpisce più la stessa superficie.
- G_2: La microsfaccettatura ĂŠ totalmente esposta alla luce incidente, ma quella riflessa viene parzialmente o totalmente intercettata da altre microsfaccettature (questa luce intercettata ed a sua volta riflessa contribuisce alla riflessione diffusa).
- G_3: La microsfaccettatura ĂŠ parzialmente schermata dalla luce incidente.

Il fattore geometrico di attenuazione varia da 0 (occlusione totale) ad 1 (nessuna occlusione). Nel primo caso, nel quale tutta la luce incidente viene riflessa, poniamo il fattore di attenuazione geometrica G uguale ad 1. In entrambi gli altri casi la quantità di luce intercettata da altre sfaccettature è pari a $\frac{M}{A}$, dove A è l'area totale della microsfaccettatura e M è l'area la cui luce riflessa è bloccata. Quindi $G_2 = G_3 = 1 - \frac{M}{A}$. La proporzione di luce riflessa in questi casi è stata calcolata in un articolo di Blinn ed il risultato è:

$$G = \min\left\{1,\ \frac{2\langle \boldsymbol{N}, \boldsymbol{H} \rangle \langle \boldsymbol{N}, \boldsymbol{\Theta} \rangle}{\langle \boldsymbol{\Theta}, \boldsymbol{H} \rangle},\ \frac{2\langle \boldsymbol{N}, \boldsymbol{H} \rangle \langle \boldsymbol{N}, \boldsymbol{\Psi} \rangle}{\langle \boldsymbol{\Theta}, \boldsymbol{H} \rangle}\right\}$$

Nel progetto si è scelto di utilizzare una unica BRDF per il modello di Lambert e quello di Cook e Torrance, il secondo viene considerato solamente se il parametro `slope` è diverso da 0.

```
<<geometry.h>>
<<Struct Material>>
```

```
float3 C_T_BRDF(Ray psi,Ray theta,float3 n){
        //parte diffusiva
        float3 fr= Kd*M_1_PI;
         if(slope!=0){

        //dati:

        //halfway vector
        float3 H= (psi.d+theta.d).norm();
        //<psi,H>=<theta,H>
        float c= psi.d.dot(H);
        //<n,H>
        float cNH=n.dot(H);
        //<psi,n>
        float cPsiN=psi.d.dot(n);
        //<teta,n>
        float cThetaN=theta.d.dot(n);

        //calcolo del coefficiente di Fresnel:

        float3 F;
        float3 ior2=float3(ior.x*ior.x,ior.y*ior.y,ior.z*ior.z);
        float3 g=ior2+float3(c*c-1);
        g.abs();
        g=float3(sqrtf(g.x),sqrtf(g.y),sqrtf(g.z));
            //g+c
            float3 gc= g+float3(c);
            //g-c
            float3 g_c=g-float3(c);
            // (g-c)^2/(g+c)^2
            float3 a=((g_c).mult(g_c)).div(gc.mult(gc));
            //(c*(g+c)-1)/(c*(g+c)+1)
            float3 b= ( (gc*c)-float3(1)).div( (gc*c)+float3(1));
            //F= 1/2 * ((g-c)^2/(g+c)^2) * (1+ ((c*(g+c)-1)/(c*(g+c)+1))^2)
            F = a.mult(float3(1)+ b.mult(b))*1/2;

            //Distribuzione di Beckmann:
            double sNH= sqrt(1-pow(cNH,2));

            double tan= sNH/(cNH*slope);
            double e= exp(pow(tan,2));
            double _D = e*(slope*slope*pow(cNH,4));
            double D= 1/_D;

            //Fattore geometrico:
```

```
            double G= 1;
            if( 2*cPsiN*cNH/c <G){ G=2*cPsiN*cNH/c;}
            if(2*cThetaN*cNH/c<G){ G=2*cThetaN*cNH/c;}

            //Modello di Cook-Torrance:
            fr=fr+ F*D*G*M_1_PIf*1/(cPsiN*cThetaN);
        }

    return fr;
    }
```

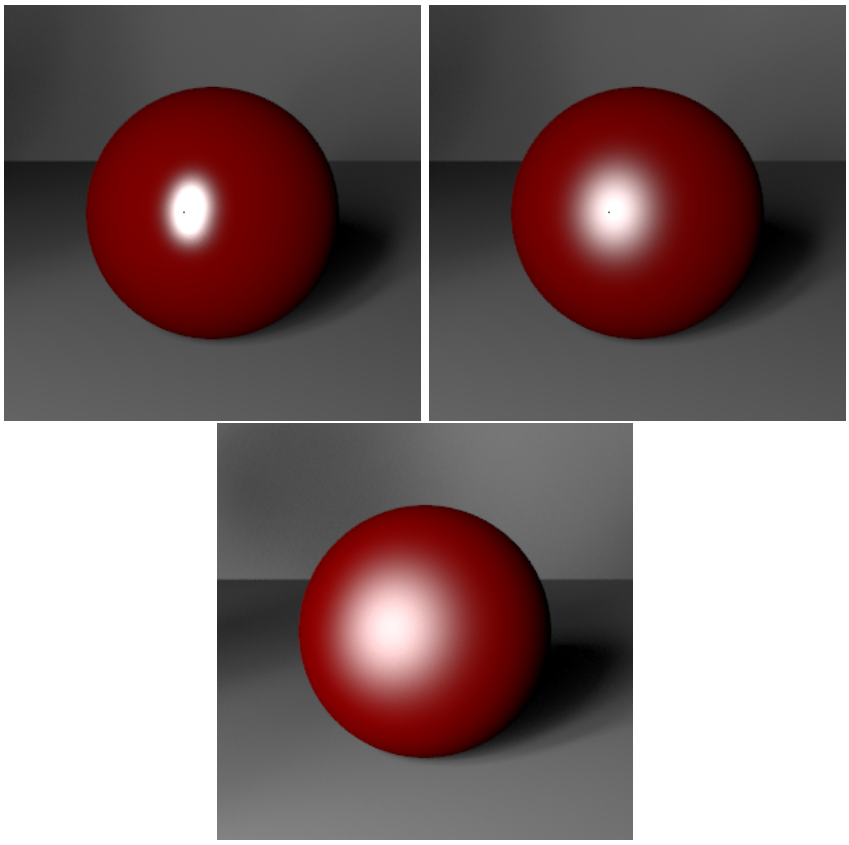


FIGURA 7.7.6. Modello di Cook-Torrance: nell'immagine in alto a sinistra lo `slope` Äš posto a 0.1, a destra a 0.2 e in basso a 0.3

7.8. Equazione del rendering

Ora siamo pronti per formulare la condizione del trasporto della potenza luminosa in una scena tramite una equazione ricorsiva, l'equazione del rendering. L'obiettivo degli algoritmi di Illuminazione Globale è calcolare la distribuzione di equilibrio della potenza luminosa scambiata fra le varie parti della scena. Come precedentemente menzionato, consideriamo il caso in cui la luce si propaga nel vuoto, quindi in assenza di mezzi atmosferici. Su ogni punto della superficie x ed in ogni direzione

$\mathbf{\Theta}$, l'equazione del rendering descrive la radianza uscente $L(x \to \mathbf{\Theta})$ dal punto x in direzione $\mathbf{\Theta}$ come integrale della radianza incidente dalle varie direzioni.

7.8.1. Formulazione emisferica. La formulazione più comune è quella emisferica: la spieghiamo in base alla conservazione dell'energia che passa per un punto x. Supponiamo che $L_e(x \to \mathbf{\Theta})$ sia la radianza creata da una superficie nel punto x ed inviata nella direzione $\mathbf{\Theta}$, e $L_r(x \to \mathbf{\Theta})$ sia la radianza che viene riflessa dalla superficie nel punto x nella direzione $\mathbf{\Theta}$ dopo essere arrivata a x dalle varie direzioni incidenti. Per la conservazione dell'energia, la radianza totale uscente da un punto in una particolare direzione di uscita è la somma della radianza emessa e della radianza riflessa in quella direzione a quel punto della superficie. Quindi:

$$L(x \to \mathbf{\Theta}) = L_e(x \to \mathbf{\Theta}) + L_r(x \to \mathbf{\Theta})\,.$$

Dalla definizione di BRDF abbiamo

$$f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) = \frac{dL_r(x \to \mathbf{\Theta})}{dE(x \leftarrow \mathbf{\Psi})}\,,$$

e, da (7.6.2),

$$L_r(x \to \mathbf{\Theta}) = \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) L(x \leftarrow \mathbf{\Psi}) \langle \boldsymbol{N}_x, \mathbf{\Psi} \rangle \, d\omega(\mathbf{\Psi})$$

Combinando queste equazioni, otteniamo l'equazione del rendering nella forma dell'integrale sull'emisfero:

$$L(x \to \mathbf{\Theta}) = L_e(x \to \mathbf{\Theta}) + \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta})\, L(x \leftarrow \mathbf{\Psi})\, \langle \boldsymbol{N}_x, \mathbf{\Psi} \rangle \, d\omega(\mathbf{\Psi})\,. \tag{7.8.1}$$

Si tratta di una equazione integrale ricorsiva (una equazione di Fredholm di secondo tipo).

7.8.2. Formulazione di area. Una formulazione alternativa frequente si ottiene integrando i contributi della radianza inviata a x non sull'emisfero frontale, bensì su tutti i punti visibili da x nelle superficie degli oggetti della scena.
Prendiamo in considerazione i termini della equazione del rendering (7.8.1) nella precedente formulazione emisferica. Denotiamo con y il punto della scena che x vede nella direzione $\mathbf{\Psi}$. Assumendo che non ci sia nessun mezzo permeabile, ossia di propagazione, sappiamo che la radianza che entra nel punto x in provenienza da $\mathbf{\Psi}$ è la stessa che esce da y in direzione $-\mathbf{\Psi}$:

$$L(x \leftarrow \mathbf{\Psi}) = L(y \to -\mathbf{\Psi})$$

Inoltre, in base a (7.3.1), l'elemento di integrazione rispetto all'angolo solido può essere scritto come segue:

$$d\omega(\mathbf{\Psi}) = d\omega_{x \leftarrow d\sigma(y)} = \langle \boldsymbol{N}_y, -\mathbf{\Psi} \rangle \, \frac{d\sigma(y)}{r_{xy}^2}\,.$$

Sostituendo nell'equazione (7.8.1), trasformiamo l'integrale dell'equazione del rendering in un integrale sull'unione S di tutte le superficie della scena, esattamente come abbiamo fatto nella Sezione 6.3 per il calcolo dei fattori di forma tramite proiezione radiale su un buffer emisferico (metodo di Nusselt):

$$L(x \to \mathbf{\Theta}) = L_e(x \to \mathbf{\Theta}) + \int_S f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta})\, L(y \to -\mathbf{\Psi})\, V(x,y)\, \frac{\langle \boldsymbol{N}_x, \mathbf{\Psi} \rangle \, \langle \boldsymbol{N}_y, -\mathbf{\Psi} \rangle}{r_{xy}^2}\, d\sigma(y)\,.$$

Denotiamo con $G(x,y)$ il termine che dipende solo dalla geometria degli oggetti nei punti x e y:

$$G(x,y) = \frac{\langle \boldsymbol{N}_x, \mathbf{\Psi} \rangle \, \langle \boldsymbol{N}_y, -\mathbf{\Psi} \rangle}{r_{xy}^2}\,.$$

Allora

$$L(x \to \mathbf{\Theta}) = L_e(x \to \mathbf{\Theta}) + \int_S f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta})\, L(y \to -\mathbf{\Psi})\, V(x,y)\, G(x,y)\, d\sigma(y)\,.$$

Questa è la formulazione dell'equazione del rendering in termini di integrazione su tutte le superficie della scena.

7.8.3. Radianza diretta e indiretta. Possiamo riformulare l'equazione del rendering separando i termini di illuminazione diretti da quelli indiretti. L'illuminazione diretta è quella che arriva direttamente su una superficie, proveniente da una sorgente luminosa nella scena; l'illuminazione indiretta è la luce che arriva dopo essere rimbalzata su altri oggetti della scena.
Se separiamo nell'integrale i termini di illuminazione diretta da quelli di illuminazione indiretta, ed indichiamo con $\overrightarrow{yx}$ il versore $-\mathbf{\Psi}$ se $y = r(x, \mathbf{\Psi})$ è il punto della scena visibile da x nella direzione $\mathbf{\Psi}$, scomponiamo l'equazione del rendering come segue:

$$L(x \to \mathbf{\Theta}) = L_e(x \to \mathbf{\Theta} + L_r(x \to \mathbf{\Theta})\,, \tag{7.8.2}$$

$$\begin{aligned} L_r(x \to \mathbf{\Theta}) &= \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) L(x \leftarrow \mathbf{\Psi})\, \langle \boldsymbol{N}_x, \mathbf{\Psi} \rangle\, d\omega(\mathbf{\Psi}) \\ &= \int_S f_r(x, \overrightarrow{xy} \leftrightarrow \mathbf{\Theta}) L(y \to \overrightarrow{yx})\, V(x,y)\, G(x,y)\, d\sigma(y)\,, \\ &= L_{\text{diretta}} + L_{\text{indiretta}}\,, \end{aligned}$$

$$L_{\text{diretta}} = \int_S f_r(x, \overrightarrow{xy} \leftrightarrow \mathbf{\Theta}) L_e(y \to \overrightarrow{yx})\, V(x,y)\, G(x,y)\, d\sigma(y)\,, \tag{7.8.3}$$

$$\begin{aligned} L_{\text{indiretta}} &= \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) L_i(x \leftarrow \mathbf{\Psi})\, \langle \boldsymbol{N}_x, \mathbf{\Psi} \rangle\, d\omega(\mathbf{\Psi}) \\ &= \int_S f_r(x, \overrightarrow{xy} \leftrightarrow \mathbf{\Theta}) L_r(y \to \overrightarrow{yx})\, V(x,y)\, G(x,y)\, d\sigma(y)\,, \end{aligned} \tag{7.8.4}$$

dove il termine L_i è la radianza che arriva a x perché riflessa verso x dal punto $y = r(x, \mathbf{\Psi})$ dirimpettaio di x nella direzione $\mathbf{\Psi}$:

$$L_i(x \leftarrow \mathbf{\Psi}) = L_r(r(x, \mathbf{\Psi}) \to -\mathbf{\Psi})\,. \tag{7.8.5}$$

Così, il termine diretto è il contributo della radianza creata in punti della scena $y = r(x, \overrightarrow{xy})$ visibili da x ed inviata a x (ossia lungo la direzione $\overrightarrow{xy}$); invece l'illuminazione indiretta è la radianza inviata a x da punti $y = r(x, \mathbf{\Psi})$ nei quali non viene creata ma solo riflessa.

7.8.4. Relazione fra radianza e flusso radiativo. La soluzione teorica per il problema dell'illuminazione globale consiste nel trovare tutti i valori della funzione della radianza per tutte le possibili superficie e tutte le direzioni relative ai punti di queste superficie. È chiaro che questo non è possibile in pratica, in quanto richiederebbe di calcolare un integrale per ogni coppia punto-direzione, e queste coppie sono infinite.

Gli algoritmi di illuminazione globale hanno l'obiettivo di calcolare la radianza *media* su insiemi di punti e direzioni in un'area. Un modo per calcolare il valore medio della radianza è quello di calcolare su quell'area il flusso della varianza. Assumiamo che la radianza cambi lentamente nell'area: quindi il valore medio può essere approssimato dividendo il flusso per l'angolo solido totale e la superficie totale dell'area.

Possiamo esprimere il flusso radiativo in termini di radianza integrando la distribuzione della radianza su tutti i possibili punti della superficie e le direzioni attorno a questi punti. Sia Σ un insieme di coppie (ordinate) consistente di punti della superficie σ_s e direzioni di uscita Ω_s a quei punti (o più precisamente, per ogni x in σ_s, consideriamo le direzioni di uscita in un insieme $\Omega_s(x)$ che dipende da x). Quindi $\Sigma = \{s \mapsto (\sigma_s, \Omega_s)\}$, dove s varia su un appropriato insieme di parametri. Con abuso di notazione, scriviamo $\Sigma = \sigma_s \times \Omega_s$. Allora il flusso $\Phi(\Sigma)$ si calcola così:

$$\Phi(\Sigma) = \int_{\sigma_s} \int_{\Omega_s} L(x \to \mathbf{\Theta}) \cos(N_x, \mathbf{\Theta})\, d\omega(\mathbf{\Theta})\, d\sigma(x)\,.$$

Possiamo riscrivere questa formula integrando su tutta la superficie σ e su tutto l'emisfero Ω per tutti i punti della scena, nel modo seguente

$$\Phi(\Sigma) = \int_{\sigma} \int_{\Omega} L(x \to \mathbf{\Theta}) W_e(x \leftarrow \mathbf{\Theta}) \cos(N_x, \mathbf{\Theta})\, d\omega(\mathbf{\Theta})\, d\sigma(x)\,.$$

dove abbiamo posto

$$W_e(x \leftarrow \mathbf{\Theta}) = \begin{cases} 1 & \text{se } (x, \mathbf{\Theta}) \in \Sigma, \\ 0 & \text{se } (x, \mathbf{\Theta}) \notin \Sigma. \end{cases} \tag{7.8.6}$$

Il valore medio del flusso associato all'insieme di punti e direzioni si esprime allora così:

$$\Phi_{\text{medio}} = \frac{\int_\sigma \int_\Omega L(x \to \mathbf{\Theta}) W_e(x \leftarrow \mathbf{\Theta}) \cos(N_x, \mathbf{\Theta})\, d\omega(\mathbf{\Theta})\, d\sigma(x)}{\int_\sigma \int_\Omega W_e(x \leftarrow \mathbf{\Theta}) \cos(N_x, \mathbf{\Theta})\, d\omega(\mathbf{\Theta})\, d\sigma(x)}$$

7.9. L'equazione dell'importanza

Ogni pixel di un'immagine ottenuta dal rendering di una scena si può pensare come un sensore che risponde alla luce che riceve. È opportuno determinare la sua funzione di risposta (o *funzione potenziale*, o anche *funzione di importanza*) che stabilisce quali parti o punti della scena sono maggiormente importanti nel determinare l'illuminazione del pixel. Chiamiamo W la funzione di importanza: allora $W(x, \mathbf{\Theta})$ determina l'importanza del contributo della direzione $\mathbf{\Theta}$ all'illuminazione del punto x. Ora, l'importanza si scinde come somma di due termini. Se un punto od una superficie i sono visibili dal punto x, essi assumono per questo una importanza diretta, in quanto inviano luce direttamente a x: indichiamo questa importanza diretta con $W_e(x, \mathbf{\Theta})$. Ma se invece da i è visibile un altro punto o superficie j che è visibile anche da x, allora i invia luce anche a j, e parte di questa luce può essere riflessa su x. Pertanto, dal punto di vista di x, i acquisisce una importanza indiretta aggiuntiva, in quanto fonte di ulteriore illuminazione indiretta. Osserviamo due fatti:

- l'importanza si propaga in senso opposto alla radianza: il fatto che la radianza, ossia l'energia, viaggi da i a j fa sì che l'importanza si trasmetta da j a i;

- a parte questa trasmissione all'indietro, l'importanza si propaga con lo stesso meccanismo di riflessioni ricorsive della radianza; in altre parole, l'importanza deve verificare la seguente equazione integrale ricorsiva analoga a (7.8.1):

$$\begin{aligned} W(x \to \mathbf{\Theta}) &= W_e(x \to \mathbf{\Theta}) \\ &\quad + \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta})\, W(x \leftarrow \mathbf{\Psi})\, \langle \boldsymbol{N}_x, \mathbf{\Psi} \rangle\, d\omega(\mathbf{\Psi})\,. \end{aligned} \tag{7.9.1}$$

Per dimostrare (7.9.1), consideriamo il flusso $\Phi(\Sigma)$ che parte da un insieme Σ di punti in 5 dimensioni, tre spaziali e due angolari, ossia consistente di punti e di direzioni. Calcoliamo la

possibile influenza di ogni coppia $(x, \mathbf{\Theta})$ sul flusso $\Phi(\Sigma)$. Più precisamente, supponiamo di avere un singolo valore di radianza (una fonte di luce "infinitesima" data da un'area differenziale ed un angolo solido differenziale) $L(x \to \mathbf{\Theta})$: se non ci sono altre fonti di illuminazione, qual è il valore del flusso $\Phi(\Sigma)$? Il peso che dobbiamo attribuire ad $L(x, \to \Omega)$ per ottenere il flusso $\Phi(\Sigma)$ viene chiamato l'importanza di $(x, \mathbf{\Theta})$ tenendo in considerazione Σ, e si denota con $W(x \leftarrow \Omega)$.

L'importanza $W(x \to \mathbf{\Theta})$ si ottiene dai due modi in cui la radianza differenziale $L(x \to \mathbf{\Theta})$ contribuisce al flusso $\Phi(\Sigma)$:

- *Auto contributo:* se $(x, \mathbf{\Theta}) \in \Sigma$, allora $L(x \to \mathbf{\Theta})$ contribuisce totalmente a $\Phi(\Sigma)$. Questa viene chiamata l'auto-importanza dell'insieme Σ ed è espressa dalla quantità $W_e(x \leftarrow \mathbf{\Theta})$ in (7.8.6).
- *Contributo attraverso una o più riflessioni:* il secondo modo di contribuire al flusso è attraverso tutti i contributi indiretti, ossia attraverso una o più riflessioni su diverse superficie. La radianza $L(x \to \mathbf{\Theta})$ viaggia lungo un percorso rettilineo e raggiunge un punto $r(x, \mathbf{\Theta})$ di una superficie. L'energia viene riflessa su questo punto secondo una distribuzione emisferica determinata dalla BRDF. Così abbiamo un emisfero di direzioni su $r(x, \mathbf{\Theta})$: ognuna emette un valore di radianza differenziale come il risultato della riflessione della radianza $L(r(x, \mathbf{\Theta}) \leftarrow -\mathbf{\Theta})$. Integrando i valori importanti per tutte queste nuove direzioni, abbiamo un nuovo termine per $W(x \leftarrow \mathbf{\Theta})$.

Prendendo in considerazione la riflessione, e i valori di BRDF, otteniamo la seguente equazione ricorsiva per entrambi i termini combinati:

$$\begin{aligned} W(x \leftarrow \mathbf{\Theta}) &= W_e(x \leftarrow \mathbf{\Theta}) \\ &\quad + \int_{\Omega_x} f_r(z, \mathbf{\Psi} \leftrightarrow -\mathbf{\Theta}) W(z \leftarrow \Psi) \cos(\boldsymbol{N}_{r(x,\mathbf{\Theta})}, \mathbf{\Psi})\, d\omega(\mathbf{\Psi})\,, \end{aligned} \tag{7.9.2}$$

dove $z = r(x, \mathbf{\Theta})$.

7.9.1. Importanza incidente e uscente. L'equazione (7.9.2) è formalmente identica all'equazione del trasporto della radianza incidente. Quindi possiamo associare i concetti di radianza incidente e di importanza, dal momento che si comportano allo stesso modo.

La funzione sorgente W_e dipende dalla natura dell'insieme Σ. In un modello di macchina da ripresa virtuale, nel calcolo dei valori del flusso per pixel, $W_e(x \leftarrow \Omega) = 1$ se x è visibile attraverso il pixel e $\mathbf{\Theta}$ è una direzione che punta attraverso il pixel.

Per rafforzare maggiormente l'analogia con la radianza, possiamo anche introdurre l'importanza uscente definendo $W(x \to \mathbf{\Theta})$ come segue:

$$W(x \to \mathbf{\Theta}) = W(r(x, \mathbf{\Theta}) \leftarrow -\mathbf{\Theta})\,.$$

Questa definizione estende all'importanza la proprietà dell'invarianza lungo le rette. A questo punto è immediato che l'importanza uscente ha esattamente la stessa equazione di trasporto della radianza uscente:

$$W(x \to \mathbf{\Theta}) = W_e$$

7.9.2. Calcolo del flusso a partire dalla radianza creata e dall'importanza. Deduciamo un'espressione per il flusso di un insieme Σ di punti e direzioni, basato sulla funzione di importanza. Le sorgenti di luce sono i soli punti che forniscono energia luminosa all'ambiente. I loro valori di radianza danno luogo l'illuminazione dell'intera scena. Analogamente, per calcolare il flusso, basta considerare solo i valori di importanza di punti sulle sorgenti luminose;

$$\Phi(\Sigma) = \int_S \int_{\Omega_x} L_e(x \to \mathbf{\Theta}) W(x \leftarrow \mathbf{\Theta}) \cos(\boldsymbol{N}_x, \mathbf{\Theta})\, d\omega(\mathbf{\Theta})\, d\sigma(x)\,.$$

Possiamo integrare sopra tutte le superficie della scena S, dal momento che $L_e = 0$ su punti e direzioni che non appartengono alle sorgenti di luce. Questa equazione e (7.9.2) forniscono un approccio alternativo al problema dell'illuminazione globale.

È anche possibile scrivere $\Phi(\Sigma)$ anche nelle forme seguenti:

$$\Phi(\Sigma) = \int_S \int_{\Omega_x} L(x \to \mathbf{\Theta}) W_e(x \leftarrow \mathbf{\Theta}) \cos(N_x, \mathbf{\Theta})\, d\omega(\mathbf{\Theta})\, d\sigma(x)$$

$$\Phi(\Sigma) = \int_S \int_{\Omega_x} L(x \leftarrow \mathbf{\Theta}) W_e(x \to \mathbf{\Theta}) \cos(N_x, \mathbf{\Theta})\, d\omega(\mathbf{\Theta})\, d\sigma(x)$$

Così il flusso di un dato insieme Σ di coppie punti e direzioni può essere calcolato con quattro diversi integrali, ciascuno dei quali può essere calcolato attraverso una doppia integrazione su tutte le superficie o gli emisferi.

Ora siamo in possesso di due diversi approcci per risolvere il problema dell'illuminazione globale. Il primo approccio parte dalla scelta dell'insieme Σ e richiede il calcolo dei valori di radianza per queste coppie. I valori di radianza devono essere calcolati risolvendo una delle equazioni ricorsive della radianza. In questo modo, cominciamo dall'insieme e risaliamo verso le sorgenti di luce.

Il secondo approccio calcola il flusso su un dato insieme di coppie punto-direzione comiciando dalle sorgenti di luce, e per ogni sorgente di luce calcola l'importanza che essa ha per quell'insieme. Il calcolo di questo valore di importanza si ottiene tramite una delle equazioni integrali ricorsive dell'importanza. Quindi questo tipo di algoritmo comincia dalle sorgenti di luce e procede tracciando l'importanza verso l'insieme Σ.

7.10. L'equazione della misura

Quando abbiamo introdotto il Ray Tracing, abbiamo modellato il problema della colorazione dei pixel tracciando raggi proiettori dal punto di osservazione alla scena attraverso il centro dei pixel. Quindi, in quel modello, i pixel si colorano grazie alla radianza che passa attraverso di essi e va verso il punto di osservazione. Ma un modello più corrispondente alla struttura delle immagini digitali, prodotte dall'illuminazione dei pixel dei loro sensori, si basa su quanta radianza arriva a ciascuno di questi pixel (che sono piccole celle del sensore, che possiamo supporre rettangolari, anche se i sensori veri hanno pixel di forma più complessa), e non solo nella direzione dell'osservatore. Quindi occorre calcolare, per ciascun pixel j, una misura della sua luminosità e colore dovuta alla radianza che lo raggiunge.

Ora, in base all'equazione dell'importanza (7.9.1) la luminosità del pixel j è

$$L_j = \int W_e(x \to \mathbf{\Psi})\, L_i(x \leftarrow \mathbf{\Psi})\, \cos(\boldsymbol{N}_x,\, \mathbf{\Psi})\, d\sigma(x)\, d\omega(\mathbf{\Psi})\,,$$

dove integriamo al variare di $\mathbf{\Psi}$ nell'emisfero frontale al pixel j ed al variare del punto x sulla superficie del pixel j. Questa si chiama l'equazione della misura (la misura è quella della luminosità del pixel j).

CAPITOLO 8

Integrazione numerica con estimatori probabilistici

8.1. Introduzione alla probabilità

DEFINIZIONE 8.1.1 (**σ-algebra**). Sia $\mathcal{Q}$ un insieme. Indichiamo con $\mathcal{P}(\mathcal{Q})$ l'insieme delle parti di $\mathcal{Q}$, ovvero l'insieme formato dai suoi sottoinsiemi. Una famiglia $\mathcal{A} \subset \mathcal{P}(\mathcal{Q})$ di parti di $\mathcal{Q}$ si dice una σ-algebra su $\mathcal{Q}$ se verifica le seguenti proprietà:

(1) $\{\emptyset, \mathcal{Q}\} \subset \mathcal{A}$
(2) $X \in \mathcal{A} \Longrightarrow X^{\complement} \in \mathcal{A}$ (dove $X^{\complement}$ indica il complementare di X)
(3) se $\{X_i\}_{i\in\mathbb{Z}} \subset \mathcal{A}$, allora

$$\bigcap_{i\in\mathbb{Z}} X_i \in \mathcal{A}$$

Si osservi che se $\mathcal{A}$ è una σ-algebra, le proprietà (2) e (3) implicano che per ogni $\{X_i\}_i \subset \mathcal{A}$ si abbia $\bigcup_{i\in\mathbb{Z}} X_i \in \mathcal{A}$. Infatti per ogni collezione numerabile di insiemi $\{Y_k\}_k$, vale l'identità $\bigcup_k Y_k = \left(\bigcap_k Y_k^{\complement}\right)^{\complement}$.

DEFINIZIONE 8.1.2 (**Probabilità**). Dato lo spazio misurabile $(\mathcal{Q}, \mathcal{A})$, dove $\mathcal{Q}$ è un insieme e $\mathcal{A}$ è una σ-algebra su $\mathcal{Q}$, una misura $\mathbf{P}$ su $(\mathcal{Q}, \mathcal{A})$ è detta *misura di probabilità* o, più spesso, *probabilità.* se verifica le seguenti proprietà:

(1) $\mathbf{P} : \mathcal{A} \longrightarrow [0, 1]$
(2) $\mathbf{P}(\Omega) = 1$
(3) $\mathbf{P}$ è σ-additiva, ovvero per ogni $\{X_j\} \subset \mathcal{A}$ tale che $X_n \cap X_k = \emptyset$, $\forall k \neq n$, si ha

$$\mathbf{P}\left(\bigcup_{j\geqslant 0} X_j\right) = \sum_{j\geqslant 0} \mathbf{P}\left(X_j\right)$$

DEFINIZIONE 8.1.3 (**Spazio di probabilità**). Uno *spazio di probabilità* è una tripla $(\Omega, \mathcal{A}, \mathbf{P})$, dove Ω è uno spazio, detto *spazio dei campioni*; $\mathcal{A}$ è una σ-algebra su Ω che rappresenta l'insieme di tutti gli eventi possibili (pertanto un elemento $E \in \mathcal{A}$ si dice *evento*); $\mathbf{P}$ è una probabilità su $(\Omega, \mathcal{A})$.

Dalla definizione seguono alcune fondamentali proprietà

PROPOSIZIONE 8.1.4. *Sia* $(\Omega, \mathcal{A}, \mathbf{P})$ *uno spazio di probabilità .*

(i) $\forall A, B \in \mathcal{A}$ *tali che* $A \subseteq B$, $\mathbf{P}(A) \leqslant \mathbf{P}(B)$.
(ii) $\forall \{A_i\}_{i\in\mathbb{N}} \subset \mathcal{A}$ *si ha*

$$\mathbf{P}\left(\bigcup_{i\in\mathbb{N}} A_i\right) \leqslant \sum_{i\in\mathbb{N}} \mathbf{P}(A_i). \tag{8.1.1}$$

(iii) $\forall A \in \mathcal{A}, \quad \mathbf{P}\left(A^{\complement}\right) = 1 - \mathbf{P}(A)$.
(iv) $\forall A, B \in \mathcal{A}, \quad \mathbf{P}(A \cup B) = \mathbf{P}(A) + \mathbf{P}(B) - \mathbf{P}(A \cap B)$.

*Dimostrazione*Per mostrare (1) basta osservare che $B = A \cup (A^{\complement} \cap B)$. Certamente $A \cap (A^{\complement} \cap B) = \emptyset$: pertanto dalla definizione di probabilità si ha subito

$$\mathbf{P}(B) = \mathbf{P}(A) + \mathbf{P}\left(A^{\complement} \cap B\right) \geqslant \mathbf{P}(A).$$

Mostrariamo dapprima la (2) nel caso di due insiemi. Sia $C = A \cap B^{\complement}$ evidentemente $C \subseteq A$ e $B \cap C = \emptyset$, inoltre $B \cup C = B \cup A$ e pertanto segue dalla definizione di probabilità e da (1) che

$$\mathbf{P}(A \cup B) = \mathbf{P}(B \cup C) = \mathbf{P}(B) + \mathbf{P}(C) \leqslant \mathbf{P}(B) + \mathbf{P}(A)$$

Si può ragionare analogamente nel caso di più insiemi. Supponiamo data $\{A_i\}_{i\in\mathbb{N}}$, poniamo $C_j = A_j \cap (\cup_{i=0}^{j-1} A_i)$, ne segue $\cup_i C_i = \cup_i A_i$ e $C_i \subseteq A_i$, dunque la tesi. La (3) è pressoché immediata. Certamente $A \cap A^{\complement} = \emptyset$, d'altra parte $A \cup A^{\complement} = \Omega$. Dalla definizione di probabilità segue, pertanto, $1 = \mathbf{P}(\Omega) = \mathbf{P}(A) + \mathbf{P}(A^{\complement})$. Infine, per provare (4) basta porre

$$A \cup B = \left(A \cap B^{\complement}\right) \cup \left(B \cap A^{\complement}\right) \cup (A \cap B)$$

dove i tre insiemi a destra dell'ugualianza sono disgiunti. Dall'osservare che $A = (A \cap B^{\complement}) \cup (A \cap B)$ e $B = (B \cap A^{\complement}) \cup (A \cap B)$ segue che

$$\begin{aligned}\mathbf{P}(A \cup B) &= \left(\mathbf{P}(A \cap B^{\complement}) + (A \cap B)\right) + \mathbf{P}\left((B \cap A^{\complement}) + (A \cap B)\right) - \mathbf{P}(A \cap B) \\ &= \mathbf{P}(A) + \mathbf{P}(B) - \mathbf{P}(A \cap B).\end{aligned}$$

□

Proposizione 8.1.5. *Sia $(\Omega, \mathcal{A}, \mathbf{P})$ uno spazio di probabilità . Siano $\{E_i\}_{i\in\mathbb{N}} \subset \mathcal{A}$ ed $\{F_i\}_{i\in\mathbb{N}} \subset \mathcal{A}$ due successioni di eventi tali che $E_i \subset E_{i+1}$, $F_i \supset F_{i+1}$ per ogni $i \in \mathbb{N}$. Allora*

$$\mathbf{P}\left(\bigcup_{n\in\mathbb{N}} E_n\right) = \lim_{n\to+\infty} \mathbf{P}(E_n), \quad \mathbf{P}\left(\bigcap_{n\in\mathbb{N}} F_n\right) = \lim_{n\to+\infty} \mathbf{P}(F_n) \tag{8.1.2}$$

*Dimostrazione*Consideriamo la successione $\mathcal{A} \supset \{\mathcal{E}_i\}_{i\in\mathbb{N}}$ ove $\mathcal{E}_1 = E_1$, $\mathcal{E}_i = E_i \setminus E_{i-1}$. Si ha $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$, per ogni $i \neq j$, $\cup_{i\in\mathbb{N}} \mathcal{E}_i = \cup_{i\in\mathbb{N}} E_i$. Allora

$$\begin{aligned}\mathbf{P}\left(\bigcup_{i\in\mathbb{N}} E_i\right) = \mathbf{P}\left(\bigcup_{i\in\mathbb{N}} \mathcal{E}_i\right) &= \sum_{i\in\mathbb{N}} \mathbf{P}(\mathcal{E}_i) = \lim_{n\to+\infty} \sum_{i=1}^{n} \mathbf{P}(\mathcal{E}_i) \\ &= \lim_{n\to+\infty} \mathbf{P}\left(\bigcup_{i=1}^{n} \mathcal{E}_i\right) = \lim_{n\to+\infty} \mathbf{P}\left(\bigcup_{i=1}^{n} E_i\right) = \lim_{n\to+\infty} \mathbf{P}(E_n).\end{aligned}$$

Con un ragionamento del tutto analogo si dimostra la seconda ugualianza in (8.1.2) □

Definizione 8.1.6 (**Indipendenza**). Sia $(\Omega, \mathcal{A}, \mathbf{P})$ uno spazio di probabilità, n eventi $A_1, \ldots, A_n \in \mathcal{A}$ si dicono *indipendenti* se per ogni sottoinsieme $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ si ha

$$\mathbf{P}\left(\bigcap_{j=1}^{k} A_{i_j}\right) = \prod_{j=1}^{k} \mathbf{P}(A_{i_j})$$

DEFINIZIONE 8.1.7 (**Probabilità condizionale**). Sia $(\Omega, \mathcal{A}, \mathbf{P})$ uno spazio di probabilità . Siano $A, B \in \mathcal{A}$ con $\mathbf{P}(A) > 0$, si dice *probabilità condizionale di B dato A* il valore

$$\mathbf{P}(B \mid A) = \frac{\mathbf{P}(A \cap B)}{\mathbf{P}(A)}$$

8.1.1. Variabili aleatorie, distribuzioni, funzioni di ripartizione.

DEFINIZIONE 8.1.8 (**Variabile aleatoria**). Sia $(\Omega, \mathcal{A}, \mathbf{P})$ uno spazio di probabilità . Una variabile aleatoria è una funzione $X : \Omega \longrightarrow \mathbb{R}$ tale che, per ogni $x \in \mathbb{R}$, si abbia

$$\{\omega \in \Omega \mid X(\omega) \leqslant x\} \in \mathcal{A} \tag{8.1.3}$$

OSSERVAZIONE 8.1.9. Per semplicità di notazione, nel seguito, indicheremo con $\{X \leqslant x\}$ l'insieme $\{\omega \in \Omega \mid X(\omega) \leqslant x\}$ e scriveremo $\mathbf{P}(X \leqslant x)$ al posto di $\mathbf{P}(\{X \leqslant x\})$.

Una variabile aleatoria X si dice *discreta* se assume al più una infinità numerabile di valori, ovvero se $X(\Omega) \subseteq \{x_1, x_2, \dots, x_n, x_{n+1}, \dots\}$, dove con $X(\Omega)$ intendiamo l'immagine di X; si dice *continua* altrimenti. Si osserva facilmente che se X è una variabile aleatoria discreta, per la definizione data, gli insiemi $\{X = x_i\} = \{\omega \in \Omega \mid X(\omega) = x_i\}$ sono eventi (ovvero sono in $\mathcal{A}$), pertanto ha senso la definizione che segue

DEFINIZIONE 8.1.10 (**Distribuzione discreta di probabilità**). Sia X una variabile aleatoria discreta su $(\Omega, \mathcal{A}, \mathbf{P})$, spazio di probabilità . Siano $\{x_i\}_{i\in\mathbb{N}}$ i valori assunti da X. La funzione $p_X : \mathbb{R} \longrightarrow [0, 1]$ definita da $p_X(x) = \mathbf{P}(X = x)$ si dice *distribuzione discreta di probabilità* di X.

Osserviamo che valgono le due proprietà seguenti

(1) $p_X(x) = 0$ per quasi tutti gli $x \in \mathbb{R}$, tranne al più un insime numerabile.
(2)

$$\sum_{i\in\mathbb{N}} p_X(x_i) = 1 \tag{8.1.4}$$

Infatti gli eventi $\{X = x_j\}$ sono a due a due disgiunti al variare di $j \in \mathbb{N}$, inoltre, essendo $X(\Omega) = \{x_i\}_{i\in\mathbb{N}}$, necessariamente $\cup_{i\in\mathbb{N}}\{X = x_i\} = \Omega$. Ne segue che

$$\sum_{i\in\mathbb{N}} p_X(x_i) = \sum_{i\in\mathbb{N}} \mathbf{P}(X = x_i) = \mathbf{P}\left(\bigcup_{i\in\mathbb{N}}\{X = x_i\}\right) = \mathbf{P}(\Omega) = 1$$

DEFINIZIONE 8.1.11 (**Funzione di ripartizione**). La *funzione di ripartizione* di una variabile aleatoria X è la funzione così definita

$$F_X : \mathbb{R} \longrightarrow [0, 1]; \quad F_X(x) = \mathbf{P}(X \leqslant x)$$

Osserviamo che una funzione di ripartizione è sempre non-decrescente, infatti per ogni coppia $(x, y) \in \mathbb{R}^2$ tali che $x \leqslant y$ si ha $\{X \leqslant x\} \subseteq \{X \leqslant y\}$ e dalla Proposizione 8.1.4 (i) segue $F_X(x) \leqslant F_X(y)$. Inoltre, essendo $\{X = x_i\} \cap \{X = x_j\} = \emptyset$ per ogni $i \neq j$, risulta

$$F_X(x) = \sum_{x_i \leqslant x} p_X(x_i). \tag{8.1.5}$$

Tale formula mostra come la funzione di ripartizione e la distribuzione della variabile aleatoria X, siano sostanzialmente equivalenti. Infatti da (8.1.5) e dalla Proposizione 8.1.4 punto (4), si ha

$$p_X(x_i) = \mathbf{P}\left(\{X \leqslant x_i\} \cap \{X > x_{i-1}\}\right) = \mathbf{P}\left(\{X \leqslant x_i\} \cap \{X \leqslant x_{i-1}\}^{\complement}\right)$$
$$= \mathbf{P}\left(X \leqslant x_i\right) + \left(1 - \mathbf{P}\left(X \leqslant x_{i-1}\right)\right) - \mathbf{P}(\Omega) = F_X\left(x_i\right) - F_X\left(x_{i-1}\right) \quad (8.1.6)$$

Dunque, conoscere p_X equivale a conoscere F_X e viceversa.

Come è intuibile, una variabile aleatoria non ha ragione di essere a valori in $\mathbb{R}$ piuttosto che in $\mathbb{R}^2$ o $\mathbb{R}^n$, si parlerà in questi casi di vettori aletori n-dimensionali. Diamone una definizione rigorosa.

Definizione 8.1.12 (**Vettore aleatorio discreto)**. Siano $X_1, \ldots, X_n$ delle variabili aleatorie discrete tali che $X_i : \Omega \longrightarrow \mathbb{R}$ per ogni i. Poniamo

$$X := (X_1, \ldots, X_n), \quad X : \Omega \longrightarrow \mathbb{R}^n. \quad (8.1.7)$$

X si dice *vettore aleatorio discreto* n-dimensionale.

Sia $\mathbf{x} \in \mathbb{R}^n$, analogamente al caso monodimensionale, indicheremo con $p_X(\mathbf{x})$ la probabilità seguente

$$p_X(\mathbf{x}) = \mathbf{P}\left(X = \mathbf{x}\right) = \mathbf{P}\left(\{X_1 = x_1\} \cap \cdots \cap \{X_n = x_n\}\right) \quad (8.1.8)$$

dove $x_1, \ldots, x_n$ sono le componenti del vettore $\mathbf{x}$. La distribuzione del vettore X si dice *distribuzione congiunta* delle variabili $X_1, \ldots, X_n$. Analogamente si definisce la funzione di ripartizione di un vettore aleatorio.

Osservazione 8.1.13. Nel seguito, per semplicità di notazione, scriveremo $\mathbf{P}(X = x, Y = y)$ al posto di $\mathbf{P}(\{X = x\} \cap \{Y = y\})$.

In analogia con la definizione 8.1.6 di eventi indipendenti, diamo la seguente

Definizione 8.1.14 (**Variabili aleatorie indipendenti)**. Siano $X_1, \ldots, X_n$ variabili aleatorie. Se per ogni possibile n-upla di intervalli $\mathcal{J}^{(1)}, \ldots, \mathcal{J}^{(n)}$, $\mathcal{J}^{(i)} \subset \mathbb{R}$, si ha

$$\mathbf{P}\left(X_1 \in \mathcal{J}^{(1)}, \ldots, X_n \in \mathcal{J}^{(n)}\right) = \prod_{i=1}^{n} \mathbf{P}\left(X_i \in \mathcal{J}^{(i)}\right) \quad (8.1.9)$$

diremo che le variabili aleatorie $X_1, \ldots, X_n$ sono indipendenti.

Se le variabili aleatorie considerate sono un infinità numerabile

$$\{X_1, \ldots, X_n, X_{n+1}, \ldots\},$$

diremo che sono indipendenti se e solo se lo sono un qualsiasi loro sottoinsieme finito.

Osservazione 8.1.15. Dalla definizione di indipendenza fra variabili aleatorie segue che, per ogni $X_1, \ldots, X_n$ variabili aleatorie indipendenti, posto $X = (X_1, \ldots, X_n)$, si ha

$$p_X(\mathbf{x}) = p_{X_1}(x_1) \ldots p_{X_n}(x_n), \qquad F_X(\mathbf{x}) = F_{X_1}(x_1) \ldots F_{X_n}(x_n) \quad (8.1.10)$$

e viceversa.

Osservazione 8.1.16. Se X, Z sono due variabili aleatorie indipendenti e $\lambda : \mathbb{R} \to \mathbb{R}$, $\mu : \mathbb{R} \to \mathbb{R}$ due funzioni, allora le variabili aleatorie $\lambda(X)$, $\mu(Z)$ sono indipendenti.

*Dimostrazione*Volgiamo verificare la (8.1.10). Indichiamo con $\lambda^{-1}(x)$, $\mu^{-1}(x)$ la controimmagine di x rispettivamente tramite λ e μ. Essendo X e Z indipendeti, si ha

$$\begin{aligned} p_{(\lambda(X),\mu(Z))}(\xi_i,\zeta_i) &= \mathbf{P}\left(\lambda(X)=\xi_i, \mu(Z)=\zeta_i\right) \\ &= \mathbf{P}\left(X\in\lambda^{-1}(\xi_i), Z\in\mu^{-1}(\zeta_i)\right) = \mathbf{P}\left(X\in\lambda^{-1}(\xi_i)\right)\mathbf{P}\left(Z\in\mu^{-1}(\zeta_i)\right) \\ &= \mathbf{P}\left(\lambda(X)=\xi_i\right)\mathbf{P}\left(\mu(Z)=\zeta_i\right) = p_{\lambda(X)}\left(\xi_i\right)p_{\mu(Z)}\left(\zeta_i\right) \end{aligned}$$

□

8.1.2. Attesa, varianza e covarianza.

DEFINIZIONE 8.1.17 (**Attesa di una variabile aleatoria)**. Sia X una variabile aleatoria discreta con distribuzione p_X. Siano $\{x_0, x_1, \dots, x_n, x_{n+1}, \dots\}$ i valori assunti da X. Se $\{x_k\, p_X(x_k)\}_k \in \ell^1$, si dice che X ha *media* finita. La quantità

$$E\left(X\right) := \sum_{i=0}^{+\infty} x_i\, p_X(x_i). \tag{8.1.11}$$

si chiama *attesa* di X.

Se X è un vettore aleatorio n-dimensionale, si definisce attesa di X la quantità

$$E(X) = \sum_{i_1=0}^{+\infty}\cdots\sum_{i_n=0}^{+\infty}(x_1)_{i_1}\dots(x_n)_{i_n}\, p_X((x_1)_{i_1},\dots,(x_n)_{i_n}) \tag{8.1.12}$$

Si osserva facilmente che l'operatore *attesa* è lineare, ovvero che $E(\alpha X+\beta Y) = \alpha E(X)+\beta E(Y)$, $\forall\alpha,\beta\in\mathbb{R}$, e per ogni coppia di variabili aleatorie X e Y.

PROPOSIZIONE 8.1.18. *Siano X una variabile aleatoria con distribuzione p_X e $\Phi:\mathbb{R}^n\longrightarrow\mathbb{R}$ una funzione. Posta $U=\Phi(X)$, se U ha media finita, allora*

$$E\left(U\right) = E\left(\Phi(X)\right) = \sum_{i=1}^{+\infty}\Phi\left(x_i\right)p_X\left(x_i\right) \tag{8.1.13}$$

*Dimostrazione*Siano $\{u_i\}_{i\in\mathbb{N}}$ i valori assunti da U. Indichiamo con $\mathcal{R}(u_i)$ la controimmagine di u_i tramite Φ, ovvero

$$\mathcal{R}(u_i) = \Phi^{-1}(u_i) = \{x_j\in X(\Omega)\mid u_i=\Phi(x_j)\}.$$

Allora, per ogni u_i si ha $\{U=u_i\} = \cup_{x_j\in\mathcal{R}(u_i)}\{X=x_j\}$ e dunque $\mathbf{P}(U=u_i) = \sum_{x_j\in\mathcal{R}(u_i)}\mathbf{P}(X=x_j) = p_U(u_i)$, essendo l'unione disgiunta. Avendo supposto finita la media di U, ne segue che

$$\begin{aligned} E(U) &= \sum_{i=1}^{+\infty}u_i p_U(u_i) = \sum_{i=1}^{+\infty}u_i\sum_{x_j\in\mathcal{R}(u_i)}p_X(x_j) \\ &= \sum_{i=1}^{+\infty}\sum_{x_j\in\mathcal{R}(u_i)}u_i p_X(x_j) = \sum_{i=1}^{+\infty}\sum_{x_j\in\mathcal{R}(u_i)}\Phi(x_j)p_X(x_j) \\ &= \sum_{i=1}^{+\infty}\Phi(x_i)p_X(x_i) \end{aligned}$$

□

OSSERVAZIONE 8.1.19. Ricalcando i passaggi della dimostrazione precedente si può osservare che, se $U=\Phi(X)$, la media di U è finita se e solo se la serie $\sum_{i\geqslant 0}\Phi(x_i)p_X(x_i)$ è assolutamente convergente.

OSSERVAZIONE 8.1.20. Se due variabili aleatorie U e V sono indipendenti ed a media finita, allora la variabile $X = UV$ ha media finita e si ha

$$E(X) = E(UV) = E(U)E(V) \tag{8.1.14}$$

*Dimostrazione*Sia $\Phi : \mathbb{R}^2 \to \mathbb{R}$ definita da $\Phi(a, b) = ab$, per la proposizione 8.1.18 e la definizione di indipendenza si ha

$$E(X) = E(\Phi(U,V)) = \sum_{i,j} u_i v_j p_{(U,V)}(u_i, v_j)$$
$$= \sum_i u_i p_U(u_i) \sum_j v_j p_V(v_j) = E(U)E(V)$$

□

DEFINIZIONE 8.1.21 (**Varianza)**. Sia X una variabile aleatoria discreta con media finita. L'operatore definito come segue

$$\mathrm{Var}(X) = \sigma^2(X) = E\left[(X - E(X))^2\right] \tag{8.1.15}$$

si chiama *varianza* della variabile X. La radice quadrata della varianza di X prende il nome di *deviazione standard* e si indica con $\sigma(X)$.

PROPOSIZIONE 8.1.22. *Sia X una variabile aleatoria con media finita. Risulta*

$$\mathrm{Var}(X) = E(X^2) - E(X)^2 \tag{8.1.16}$$

*Dimostrazione*Sviluppando il quadrato nella definizione 8.1.21 di varianza e ricordando che l'attesa è un operatore lineare, si ha

$$\mathrm{Var}(X) = E\left[X^2 + E(X)^2 - 2XE(X)\right]$$
$$= E(X^2) + E(X)^2 - 2E(X)E(X) = E(X^2) - E(X)^2$$

□

DEFINIZIONE 8.1.23 (**Covarianza)**. Siano X e Y due variabili aleatorie con media finita. Si definisce *covarianza* del vettore aleatorio (X, Y) la quantità

$$\mathrm{cov}(X, Y) := E(X)E(Y) - E(XY). \tag{8.1.17}$$

Due variabili aleatorie X, Y si dicono *scorrelate* se risulta

$$\mathrm{cov}(X, Y) = 0.$$

OSSERVAZIONE 8.1.24. Per ogni coppia di variabili aleatorie (X, Y) scorrelate, risulta

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$$

*Dimostrazione*Per la (8.1.17) si ha $E(XY) = E(X)E(Y) - \mathrm{cov}(X, Y) = E(X)E(Y)$. Ne segue che

$$\mathrm{Var}(X + Y) = E(X^2 + Y^2 + 2XY) - (E(X) + E(Y))^2$$
$$= \mathrm{Var}(X) + \mathrm{Var}(Y) + 2E(XY) - 2E(X)E(Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$$

□

OSSERVAZIONE 8.1.25. Se due variabili aleatorie X, Y sono indipendenti, allora sono scorrelate.

*Dimostrazione*Dalla definizione di indipendenza e di media, risulta

$$\begin{aligned}\operatorname{cov}(X,Y) &= \left(\sum_i x_i p_X(x_i)\right)\left(\sum_i y_i p_Y(y_i)\right) - \left(\sum_{i,j} x_i y_j p_{(X,Y)}(x_i,y_j)\right) \\ &= \left(\sum_i x_i p_X(x_i)\right)\left(\sum_i y_i p_Y(y_i)\right) - \left(\sum_{i,j} x_i p_X(x_i) y_j p_Y(y_j)\right) = 0\end{aligned}$$

Si noti che, in generale, l'implicazione inversa è falsa. □

Concludiamo il paragrafo con una disugualianza, di cui si fa spesso uso quando si opera con quanità casuali. Per provarne la veridicità faremo uso del seguente

LEMMA 8.1.26. *Siano U, V due variabili aleatorie con media finita. Se $U \geqslant V$ allora $E(U) \geqslant E(V)$.*

*Dimostrazione*Si ponga $X = U - V$, per ipotesi $X \geqslant 0$ pertanto $\mathbf{P}(X \geqslant 0) = 1$ e l'immagine di X è fatta di soli valori positivi. Ne segue che

$$E(X) = E(U) - E(V) = \sum_{x_i \in X(\Omega)} x_i p_X(x_i) > 0$$

□

PROPOSIZIONE 8.1.27 (Disugualianza di Tschebyscheff). *Sia X una variabile aleatoria con media e varianza finite. Per ogni $\varepsilon > 0$ risulta*

$$\mathbf{P}\left(|X - E(X)| > \varepsilon\right) \leqslant \frac{\sigma^2(X)}{\varepsilon^2} \tag{8.1.18}$$

*Dimostrazione*Introduciamo la variabile aleatoria

$$U = \varepsilon^2 \, \chi_{\{|X-E(X)|>\varepsilon\}} \,,$$

che vale costantemente ε^2 se $|X - E(X)| > \varepsilon$ e 0 altrimenti. Osserviamo che $(X - E(X))^2 \geqslant U$ poiché se $|X - E(X)| > \varepsilon$ allora $U = \varepsilon^2$ e vale con il maggiore stretto, se $|X - E(X)| \leqslant \varepsilon$ allora $U = 0$ mentre $(X - E(X))^2 \geqslant 0$. Pertanto, per il lemma 8.1.26,

$$\begin{aligned}\sigma^2(X) = E\left((X - E(X))^2\right) &\geqslant E(U) \\ &= E\left(\varepsilon^2 \chi_{\{|X-E(X)|>\varepsilon\}}\right) = \varepsilon^2 \mathbf{P}\left(|X - E(X)| > \varepsilon\right)\end{aligned}$$

□

8.1.3. Variabili aleatorie continue. Molto di frequente si ha a che fare con processi aleatori che possono assumere più che una infinità numerabile di valori. Questo è il caso di variabili aleatorie continue, di cui ci occuperemo in questa sottosezione.

Nella definizione 8.1.11 abbiamo osservato che per ogni variabile aleatoria X ed ogni $x \in \mathbb{R}$, è ben definita la probabilità che si verifichi l'evento $\{X \leqslant x\}$, probabilità che abbiamo indicato con $F_X(x)$, funzione di ripartizione di X. A partire da questa si può definire la densità di distribuzione di X, nel modo seguente

DEFINIZIONE 8.1.28 (**Densità di distribuzione**). Sia X una variabile aleatoria sullo spazio $(\Omega, \mathcal{A}, \mathbf{P})$. Sia $F_X(x)$ la sua funzione di ripartizione. La funzione $p_X : \mathbb{R} \longrightarrow [0,1]$ tale che

$$F_X(x) = \int_{-\infty}^{x} p_X(\xi)\, d\xi, \quad \forall x \in \mathbb{R} \tag{8.1.19}$$

è detta *densità di distribuzione* (o, più semplicemente *distribuzione*) della variabile X.

Da tale definizione si evince la proprietà seguente, che caratterizza una distribuzione continua di probabilità

PROPOSIZIONE 8.1.29. *Sia X una variabile aleatoria e sia p_X la sua densità di distribuzione. Allora*

$$\int_{\mathbb{R}} p_X(x)\, dx = 1 \tag{8.1.20}$$

*Dimostrazione*Mostrare la tesi equivale a mostrare che

$$\lim_{x \to +\infty} F_X(x) = 1.$$

Sia $\{\xi_n\}_n \subset \mathbb{R}$ una successione non decrescente tale che $\xi_n \to +\infty$ per n che diverge. Consideriamo l'insieme

$$\Delta = \bigcup_{n \in \mathbb{N}} \Delta_n = \bigcup_{n \in \mathbb{N}} \{\omega \,:\, X(\omega) \leqslant \xi_n\}$$

Mostriamo che la probabilità $\mathbf{P}(\Delta)$ è pari ad 1. L'inclusione $\Delta \subseteq \Omega$ è vera per definizione, viceversa supponiamo che esista $\omega \in \Omega$ tale che $\omega \notin \Delta$ allora si avrebbe $X(\omega) \geqslant \xi_n$ per ogni $n \in \mathbb{N}$. D'altra parte per come è definita $\{\xi_n\}_n$, per ogni $\mu \in \mathbb{R}$ esiste un intero n_μ tale che, per ogni $n \geqslant n_\mu$ si ha $\xi_n > \mu$. Ne segue che per ogni $\mu \in \mathbb{R}$ dovrebbe essere $X(\omega) > \mu$ e questo è un assurdo, essendo $X(\omega)$ un valore fissato, immagine di ω tramite X. Pertanto $\Omega \subseteq \Delta$ e dunque $\Omega = \Delta$, $\mathbf{P}(\Delta) = \mathbf{P}(\Omega) = 1$. Dunque, per la proposizione 8.1.5, risulta

$$\lim_{x \to \infty} F_X(x) = \lim_{n \to \infty} \mathbf{P}\left(X \leqslant \xi_n\right) = \mathbf{P}\left(\bigcup_{n \in \mathbb{N}} \{X \leqslant \xi_n\}\right) = \mathbf{P}\left(\Delta\right) = 1$$

Si noti che la prima ugualianza è vera perché F_X è una funzione non decrescente. □

Definita la distribuzione di una variabile aleatoria continua, possiamo facilmente estendere le definizioni di *attesa*, *varianza*, *covarianza* date nel caso di variabili aleatorie discrete.

DEFINIZIONE 8.1.30. Sia X una variabile aleatoriea con distribuzione p_X. Se $(x\, p_X(x)) \in L^1(\mathbb{R})$ si dice che la variabile X ha media finita e la quantità

$$E(X) := \int_{\mathbb{R}} x\, p_X(x)\, dx \tag{8.1.21}$$

è detta *attesa* o *media* di X.

OSSERVAZIONE 8.1.31. Con questa definizione restano vere tutte le proprietà che abbiamo osservato per il caso discreto, in particolare se $\Phi : \mathbb{R} \longrightarrow \mathbb{R}$ è una funzione e X una variabile aleatoria, posta $U = \Phi(X)$, risulta

$$E(U) = \int_{\mathbb{R}} \Phi(x) p_X(x)\, dx. \tag{8.1.22}$$

DEFINIZIONE 8.1.32 (**Densità marginale**). Sia $X = (X_0, \ldots, X_n)$ un vettore aleatorio con densità di distribuzione congiunta p. La quantità

$$p_{X_i}(x) = \int_{\mathbb{R}^n} p(x_0, \ldots, x_n)\, dx_0, \ldots, dx_{i-1}, dx_{i+1}, \ldots, dx_n$$

si dice *densità marginale* della variabile X_i.

Si noti che tale definizione ha senso anche nel caso di variabili (vettori) aleatorie diescrete, con la sola differenza che in luogo dell'integrale si pone la sommatoria (serie) sull'insieme dei valori assunti dalle n variabili integrande.

DEFINIZIONE 8.1.33 (**Densità condizionale**). Siano X, Y due variabili aleatorie con densità (marginale) p_X e p_Y, rispettivamente. Sia $p_{(X,Y)}$ la densità congiunta del vettore aleatorio (X, Y). Dalla definizione 8.1.7 segue che le densità delle variabili aleatorie $X|Y$ e $Y|X$ sono date da

$$p_{X|Y}(x) = \frac{p_{(X,Y)}(x,y)}{p_Y(y)}, \quad p_{Y|X}(y) = \frac{p_{(X,Y)}(x,y)}{p_X(x)}$$

ed si dicono, rispettivamente, *densità condizionale di X data Y e di Y data X.*

OSSERVAZIONE 8.1.34. La definizione appena data è ben posta. Infatti certamente $p_{X|Y}(x) \geqslant 0$; inoltre

$$\int_{\mathbb{R}} p_{X|Y}(x)\,dx = \frac{1}{p_Y(y)} \int_{\mathbb{R}} p_{(X,Y)}(x,y)\,dx = \frac{p_Y(y)}{p_Y(y)} = 1$$

ed analogamente per $p_{Y|X}(y)$.

8.2. Il problema dell'integrazione

In questa sezione presentiamo il metodo di integrazione numerica basato sulla tecnica di *Monte Carlo*. L'idea è quella di costruire una approssimazione numerica, entro una certa precisione prefissata, dell'integrale di una funzione f su un dominio di integrazione $\mathcal{Q} \subset \mathbb{R}^d$, limitato. Tale tecnica, a differenza delle formule di quadratura deterministiche, si basa su scelte probabilistiche dei *nodi* sui quali "campionare" f per costruire l'approssimazione voluta. Questo tipo di approccio, come vedremo, assicura un notevole vantaggio computazionale quando la dimensione $d > 0$ del dominio della funzione da integrare cresce.

DEFINIZIONE 8.2.1. Sia $\mathcal{Q} \subset \mathbb{R}^d$ un insieme limitato. Data una funzione $f : \mathbb{R}^d \longrightarrow \mathbb{R}$ tale che $f \in L^1(\mathcal{Q})$, indichiamo con $\mathcal{J}$ la quantià

$$\mathcal{J} = \mathcal{J}(f, \mathcal{Q}) = \int_{\mathcal{Q}} f(\mathbf{x})\,d\mathbf{x} \tag{8.2.1}$$

dove $\mathbf{x} = (x_1, \ldots, x_d) \in \mathcal{Q}$.

Vogliamo produrre una formula che approssimi $\mathcal{J}$. Supponiamo di considerare n vettori aleatori $X_1, \ldots, X_n$ d-dimensionali, ovvero $X_i = (X_i^{(1)}, \ldots, X_i^{(d)})$ per $i = 1, \ldots, n$, con stessa legge di distribuzione congiunta $p = p_{X_1} = p_{X_n}$ ed indipendenti (*indipendenti identicamente distribuiti*). È ben definita la variabile aleatoria

$$\langle \mathcal{J} \rangle_n = \langle \mathcal{J}(f, \mathcal{Q}) \rangle_n := \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)}{p(X_i)} \chi_{\mathcal{Q}}(X_i) \tag{8.2.2}$$

dove indichiamo con $\chi_{\mathcal{Q}}$ la funzione caratteristica dell'insieme $\mathcal{Q}$, tale che $\chi_{\mathcal{Q}}(\mathbf{y}) = 1$ se $\mathbf{y} \in \mathcal{Q}$, mentre vale zero altrimenti.

Tale variabile aleatoria è detta *estimatore* di $\mathcal{J}$.

PROPOSIZIONE 8.2.2. *Siano $f : \mathbb{R}^d \longrightarrow \mathbb{R}$ e $\mathcal{Q} \subset \mathbb{R}^d$ un insieme limitato. Per ogni intero $n > 0$ si ha*

$$E(\langle \mathcal{J} \rangle_n) = \mathcal{J} \tag{8.2.3}$$

*Dimostrazione*Per la proposizione 8.1.16, essendo i vettori $\{X_i\}_{i=1,\dots,n}$ indipendenti identicamente distribuiti, anche le $\left\{\frac{f(X_i)}{p(X_i)}\chi_I(X_i)\right\}_{i=1,\dots,n}$ lo sono. Allora, dalla Proposizione 8.1.18 e dalla linearità dell'operatore E, segue che

$$E\left(\langle\mathcal{J}\rangle_n\right) = \frac{1}{n}\sum_{i=1}^{n} E\left(\frac{f\left(X_i\right)}{p\left(X_i\right)}\chi_{\mathcal{Q}}(X_i)\right)$$
$$= \frac{1}{n}\sum_{i=1}^{n}\int_{\mathbb{R}^d}\frac{f\left(\mathbf{x}\right)}{p\left(\mathbf{x}\right)}\chi_{\mathcal{Q}}(\mathbf{x})p(\mathbf{x})\,d\mathbf{x} = \int_{\mathcal{Q}} f(\mathbf{x})\,d\mathbf{x} = \mathcal{J}$$

□

Abbiamo pertanto prodotto una famiglia di variabili casuali $\{\langle\mathcal{J}\rangle_n\}_n$ il cui valore atteso, indipendentemente da n e dalla dimensione d dello spazio su cui f agisce, coincide con l'integrale che volgiamo calcolare.

Essendo le $\langle\mathcal{J}\rangle_n$ aleatorie, anche l'errore che commettiamo nell'approssimare $\mathcal{J}$ facendo uso di esse, è aleatorio. Indichiamo con

$$\rho_n := \mathcal{J} - \langle\mathcal{J}\rangle_n \tag{8.2.4}$$

la variabile aleatoria dell'errore. Affinché la scelta di approssimare l'integrale con gli estimatori considerati sia una buona scelta, è necessario che ρ_n assuma, in media, valori piccoli. Osserviamo che vale la seguente

Proposizione 8.2.3. *Siano $\mathcal{J}$, $\langle\mathcal{J}\rangle_n$ e ρ_n come precedentemente definiti. Per ogni $n > 0$ si ha*

$$E(\rho_n^2) = \frac{1}{n}\,\sigma^2\left(\frac{f(X)}{p(X)}\chi_{\mathcal{Q}}(X)\right) = \frac{1}{n}\int_{\mathcal{Q}}\left(\frac{f(\mathbf{x})}{p(\mathbf{x})} - \mathcal{J}\right)^2 p(\mathbf{x})d\mathbf{x} \tag{8.2.5}$$

dove X è uno qualsiasi degli n vettori aleatori X_i considerati e p è la sua densità di distribuzione congiunta.

*Dimostrazione*Notiamo anzitutto che, per le osservazioni 8.1.24 e 8.1.25, se $U_1, \dots, U_m$ sono delle variabili aleatorie indipendenti con stessa legge di distribuzione allora

$$\sigma^2\left(\sum_{i=1}^{m} U_i\right) = \sum_{i=1}^{m}\sigma^2\left(U_i\right) = m\,\sigma^2(U)$$

essendo $\sigma^2(U) = \sigma^2(U_i)$ per ogni $i = 1, \dots, m$. Allora, indicata con $\Phi(U) := \frac{f(U)}{p(U)}\chi_{\mathcal{Q}}(U)$, si ha

$$E(\rho_n^2) = \sigma^2\left(\langle\mathcal{J}\rangle_n\right)$$
$$= \sigma^2\left(\frac{1}{n}\sum_{i=1}^{n}\Phi(X_i)\right) = \frac{1}{n^2}\sum_{i=1}^{n}\sigma^2(\Phi(X_i)) = \frac{1}{n}\sigma^2(\Phi(X))$$

essendo $\Phi(X) = \Phi(X_i)$ per ogni $i = 1, \dots, n$.

La seconda ugualianza è un immediata conseguenza della definizione di varianza e della relazione $E(\Phi(X)) = \mathcal{J}$. □

Osservazione 8.2.4. Dalla proposizione precedente segue il segue che, indipendentemente dalla dimensione dello spazio $\mathbb{R}^d$ ed indipendentemente dalla regolarità della funzione f,

$$\lim_{n\to+\infty} E\left(\rho_n^2\right) = \lim_{n\to+\infty}\mathcal{O}\left(n^{-1}\right) = 0 \tag{8.2.6}$$

poiché abbiamo assunto $f \in L^1(\mathcal{Q})$ e $0 \leqslant \int_{\mathcal{Q}} p(\mathbf{y})d\mathbf{y} < 1$ essendo p una distribuzione di probabilità congiunta su $\mathbb{R}^d$.

Pertanto gli estimatori $\langle \mathcal{J} \rangle_n$ sono, al crescere del numero n dei nodi, scelti casualmente su $\mathbb{R}^d$, delle buone approssimazioni dell'integrale $\mathcal{J}$.

Osservazione 8.2.5. La disuguaglianza di Tschebyscheff consente di dare una stima ulteriore sull'accuratezza della scelta degli estimatori $\langle \mathcal{J} \rangle_n$, infatti segue da (8.1.18) che

$$\mathbf{P}\left(|\langle \mathcal{J} \rangle_n - \mathcal{J}| > \varepsilon\right) \leqslant \frac{\sigma^2\left(\langle \mathcal{J} \rangle_n\right)}{\varepsilon^2}. \tag{8.2.7}$$

Se indichiamo con $\Phi(U) = \frac{f(U)}{p(U)} \chi_{\mathcal{Q}}(U)$, per la Proposizione 8.2.3 si ha $\sigma^2(\langle \mathcal{J} \rangle_n) = E(\rho_n^2) = \frac{1}{n}\sigma^2(\Phi(X))$, dove X è una delle n variabili aleatorie indipendenti identicamente distribuite che definiscono $\langle \mathcal{J} \rangle_n$. Pertanto, fissata una tolleranza $\tau > 0$, per ogni $\varepsilon > 0$, scelto $n(\tau) \geqslant \frac{1}{\tau\varepsilon^2}\sigma^2(\Phi(X))$ si ha

$$\mathbf{P}\left(\left|\langle \mathcal{J} \rangle_{n(\tau)} - \mathcal{J}\right| > \varepsilon\right) \leqslant \tau \tag{8.2.8}$$

e dunque con scelte opportune del numero dei nodi n si può ridurre arbitrariamente la probabilità che l'estimatore differisca dall'integrale effettivo.

D'altra parte la quantità $n(\tau)$ non è in generale calcolabile, perchè richiede di conoscere $\sigma^2(\Phi(X))$, ovvero di sapere il valore dell'integrale $\mathcal{J}$. In taluni casi è possibile dare delle stime per il valore di $\sigma^2(\Phi(X))$, ad esempio costruendo a sua volta un estimatore $\langle \mathcal{I} \rangle_m$ che approssimi l'integrale $\mathcal{I} = \sigma^2(\Phi(X))$.

Esempio 8.2.6. Vogliamo calcolare l'irradianza dovuta ad una fonte di luce $\mathbf{x}$. Dobbiamo calcolare l'integrale

$$\mathcal{J} = \int_{\Omega_{\mathbf{x}}} L(\mathbf{x} \to \Theta) \cos(N_{\mathbf{x}}, \Theta)\, d\omega(\Theta). \tag{8.2.9}$$

Poichè limitiamo il nostro interesse alla sola luce emessa da $\mathbf{x}$, indichiamo con $L_{\mathbf{x}}(\Theta) = L(\mathbf{x} \to \Theta)$ la luce emessa dal diffusore. L'integrale è esteso sul dominio

$$\Omega_{\mathbf{x}} = \left\{\Theta \in \mathbb{R}^3 \,:\, \|\mathbf{x} - \Theta\| = 1,\, \langle N_{\mathbf{x}}, \Theta \rangle \geqslant 0\right\}$$

ovvero l'emisfero "sovrastante" $\mathbf{x}$, di raggio unitario. Il consueto cambio di coordinate in funzione dei due parametri angolari di Eulero, θ e φ, permette di riscrivere l'integrale $\mathcal{J}$ nel modo seguente

$$\mathcal{J} = \int_0^{2\pi} \int_0^{\pi/2} f(\theta, \varphi)\, d\varphi\, d\theta = \int_0^{2\pi} \left\{ \int_0^{\pi/2} L_{\mathbf{x}}(\theta, \varphi) \cos\varphi \sin\varphi\, d\varphi \right\} d\theta.$$

Consideriamo uno spazio di probabilità $(\Omega, \mathcal{A}, \mathbf{P})$, scegliamo n vettori aleatori $(\theta_1, \varphi_1), \ldots, (\theta_n, \varphi_n)$ indipendenti identicamente distribuiti con densità di distribuzione congiunta p, tali che $(\theta_i, \varphi_i) : \Omega \times \Omega \to [0, 2\pi] \times [0, \pi/2]$. Costruiamo l'estimatore per $\mathcal{J}$,

$$\langle \mathcal{J} \rangle_n = \frac{1}{n} \sum_{i=1}^{n} \frac{f(\theta_i, \varphi_i)}{p(\theta_i, \varphi_i)} = \frac{1}{n} \sum_{i=1}^{n} \frac{L_{\mathbf{x}}(\theta_i, \varphi_i) \cos\varphi_i \sin\varphi_i}{p(\theta_i, \varphi_i)}.$$

Resta da scegliere con che tipo di distribuzione generare i vettori (θ_i, φ_i). Una possibilità, che semplifica la complessità di calcolo dell'estimatore, è generare casualmente valori con distribuzione

$$p(u, v) = \frac{1}{\pi} \cos v \sin v,$$

(la presenza del fattore $1/\pi$ assicura che $\int_0^{2\pi} \int_0^{\pi/2} p(u, v)\, du\, dv = 1$), con la quale si ottiene

$$\langle \mathcal{J} \rangle_n = \frac{\pi}{n} \sum_{i=1}^{n} L_{\mathbf{x}}(\theta_i, \varphi_i). \tag{8.2.10}$$

Procediamo con la nostra stima, senza preoccuparci, per ora, di come sia possibile generare campioni con una tale distribuzione. Affronteremo il problema più in là, nel paragrafo 8.4.

Poniamo $\Phi = \Phi(\mathbf{y}) = \frac{f(\mathbf{y})}{p(\mathbf{y})}\chi_{\Omega_\mathbf{x}}(\mathbf{y})$, ne segue che

$$\sigma^2(\Phi) \leqslant E(\Phi^2) = \int_{\Omega_\mathbf{x}} \frac{f^2(\theta,\varphi)}{p(\theta,\varphi)}\, d\theta\, d\varphi \leqslant \pi \left\|L_\mathbf{x}\right\|^2_{L^2(\Omega_\mathbf{x})} ,$$

allora per la (8.2.8), scegliere $n(\alpha,\beta) \geqslant \pi \left\|L_\mathbf{x}\right\|^2_{L^2(\Omega_\mathbf{x})} 10^{2\alpha+\beta}$ è una condizione sufficiente affinché

$$\mathbf{P}\left(\left|\langle \mathcal{J} \rangle_{n(\alpha,\beta)} - \mathcal{J}\right| > 10^{-\alpha}\right) \leqslant 10^{-\beta}$$

per ogni $\alpha > 0$, $\beta > 0$

8.2.1. Formule di quadratura deterministiche. L'importanza dei metodi di integrazione probabilistici presentati è evidente se paragonati alle formule di quadratura deterministiche. La costruzione di una formula che approssimi un integrale numericamente può avvenire seguendo due strade distinte: una possibilità è quella di scegliere in maniera "intelligente" i nodi di integrazione all'interno del dominio in modo da rendere minimo l'errore numerico della formula, fissato il numero di nodi n all'interno del dominio di integrazione; una seconda possibilità, computazionalmente spesso più agevole, è quella di minimizzare l'errore aumentando il numero di nodi di integrazione su cui basare la formula. Le formule della prima classe vengono dette *formule di quadratura Gaussiane*, le seconde *formule di quadratura Newtoniane*. Presentiamo brevemente, a titolo comparativo, le due tipoligie di metodi nel caso unidmensionale, benché il ventaggio dell'utilizzo dei metodi di Monte Carlo piuttosto che metodi deterministici sia marcatamente più evidente quando si vogliono approssimare integrali su domini d-dimensionali con $d > 1$.

Siano $a < b$ due valori reali ed $f \in L^1([a,b])$ una funzione da $[a,b]$ in $\mathbb{R}$, supponiamo di voler sprossimare il valore dell'integrale

$$I = \int_a^b f(x)\, dx \tag{8.2.11}$$

tramite una formula di quadratura del tipo

$$S_n(f) = \sum_{i=0}^{n} \omega_i f(x_i). \tag{8.2.12}$$

I punti $x_i \in \mathbb{R}$ per $i = 0,\dots,n$ si dicono *nodi della formula di quadratura*, i valori $\omega_i \in \mathbb{R}$ per $i = 0,\dots,n$, *pesi della formula*.

Definizione 8.2.7 (**Ordine di una formula di quadratura**). Indicato con $r_n(f) = I - S_n(f)$, una formula del tipo (8.2.12) si dice di ordine m se $r_n(p) = 0$ per ogni polinomio p di grado minore o uguale ad m e $r_n(p) \neq 0$ se $\deg(p) > m$.

Si può dimostrare che maggiore è l'ordine di una formula di quadratura, migliore è l'approssimazione che questa fornisce.

Teorema 8.2.8 (Formula di quadratura di Gauss). *Sia p_{n+1} il polinomio di grado minore uguale ad $n+1$ ortogonale ad ogni polinomio di grado minore di $n+1$ in $[a,b]$. Allora p_{n+1} esiste, è unico ed ha esattamente $n+1$ radici reali distinte $\xi_0,\dots,\xi_n$ nell'intervallo $[a,b]$. Inoltre, la formula*

$$\mathcal{G}_n(f) = \sum_{i=0}^{n} \omega_i f(\xi_i), \quad \omega_i = \int_a^b \prod_{j\neq i} \frac{(x-\xi_j)}{(\xi_i-\xi_i)}\, dx \tag{8.2.13}$$

ha ordine $2n+1$.

PROPOSIZIONE 8.2.9. *L'ordine massimo di una formula del tipo (8.2.12) è* $2n+1$.

*Dimostrazione*Se poniamo $\pi(x) = \prod_{i=0}^{n}(x-x_i)^2$ dove gli x_i sono i nodi di (8.2.12), allora $\deg(\pi) = 2n+2$ e si ha $S_n(\pi) = 0$, mentre $\int_a^b \pi(x)\,dx > 0$, pertanto $r_n(\pi) \neq 0$ e per il teorema 8.2.8 si ha la tesi. □

Ciò che rende spesso poco agevole l'utilizzo delle formule di Gauss è il calcolo del polinomio p_{n+1}, delle sue radici ξ_i e dei pesi ω_i, che da esse dipendono.

Un approccio computazionalmente più semplice è quello delle formule di Newton-Cotes, che si basano sull'idea di suddividere il dominio di integrazione $[a,b]$ in m sottointervalli $[\nu_k, \nu_{k+1}]$, $k = 0,\dots,m-1$ tali che $\nu_0 = a$, $\nu_m = b$ ed approssimare la funzione f in ognuno di tali intervalli con il polinomio $p_{n,k}$, di grado n, interpolante f in $x_0^{(k)},\dots,x_n^{(k)}$, nodi distinti in $[\nu_k,\, \nu_{k+1}]$. Possiamo scrivere $p_{n.k}$ nella forma di Lagrange, ovvero

$$p_{n,k}(x) = \sum_{i=0}^{n} f\left(x_i^{(k)}\right)\ell_{i,n}^{(k)}(x), \quad \ell_{i,n}^{(k)}(x) = \prod_{j\neq i}\frac{x - x_j^{(k)}}{x_i^{(k)} - x_j^{(k)}}\,. \tag{8.2.14}$$

Dalla linearità dell'integrale segue che

$$I = \int_a^b f(x)\,dx = \sum_{k=0}^{m-1}\int_{\nu_k}^{\nu_{k+1}} f(x)\,dx, \tag{8.2.15}$$

la formula interpolatoria di quadratura di Newton–Cotes è data da

$$\mathcal{N}_n(f) = \sum_{k=0}^{m-1}\int_{\nu_k}^{\nu_{k+1}} p_{n,k}(x)\,dx = \sum_{i=0}^{n}\left(\sum_{k=0}^{m-1}\omega_{ik}\right) f\left(x_i^{(k)}\right),$$

$$\omega_{ik} = \int_{\nu_k}^{\nu_{k+1}} \ell_{i,n}^{(k)}(x)\,dx \tag{8.2.16}$$

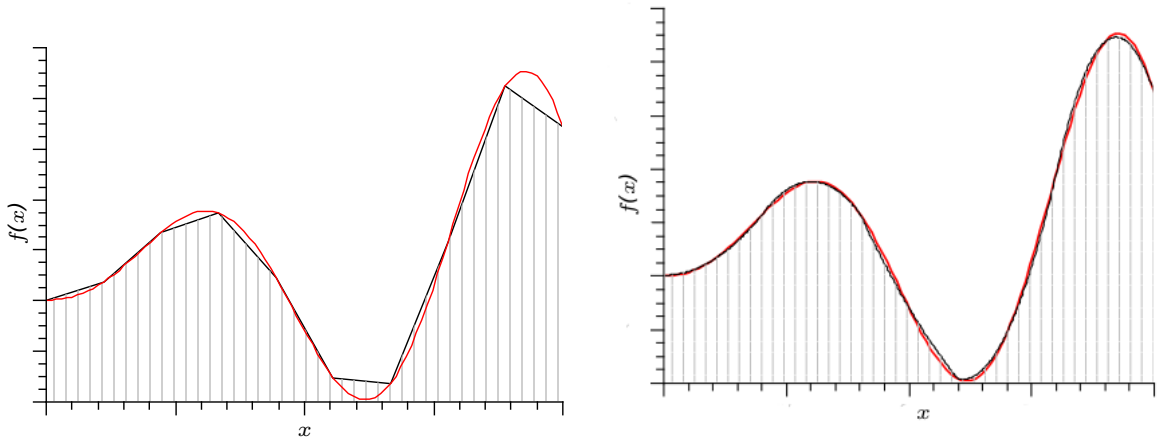
I casi $n = 1$ ed $n = 2$ sono anche noti come *formula dei trapezi* e *formula di Cavalieri-Simpson*, rispettivamente. Il primo corrisponde ad approssimare l'area del sottografico di f in $[a,b]$ con dei trapezi passanti per i vari punti dei nodi, il secondo con degli pseudo-rettangoli dove una delle due basi è sostituita da una parabola che approssima f.

In generale questo tipo di metodi danno luogo ad un errore $e_n(f) = I - N_n(f)$ che decresce come H^n dove $H = \max_{k=0,\dots,m-1}\{|\nu_{k+1} - \nu_k|\}$.

Se volessimo applicare queste formule ad un integrale esteso su un dominio $\mathcal{Q} \subset \mathbb{R}^d$ con $d > 1$, nel caso delle formule di Gauss si andrebbe in contro a problemi di complessità troppo elevata, nel caso di Newton-Cotes dovremmo suddividere $\mathcal{Q}$ in m sotto intervalli $S_i \subset \mathbb{R}^d$ (ammesso che questo sia possibile) e su di ognuno interpolare f con polinomi $\pi : \mathbb{R}^d \to \mathbb{R}$, il che richiederebbe, per un polinomio di grado n, un numero di nodi dell'ordine di n^d (ad esempio per $d = 2$ sono necessari $2^{-1}(n+1)(n+2)$ nodi). Questo rende immediatamente evidenti i vantaggi dell'integrazione probabilistica su quella deterministica quando $d > 1$. Come è già stato fatto notare, integrare con metodi di Monte Carlo garantisce gli stessi risultati di convergenza indipendentemente dalla dimensione d dello spazio e dalla regolarità del dominio $\mathcal{Q}$ di integrazione.

8.3. Riduzione dell'errore

Quando ci si appresta ad approssimare un integrale con il metodo di Mote Carlo ci si può trovare in due situazioni: il caso in cui non si hanno informazioni sulla funzione f da integrare ed il caso in cui, invece, si conosce in parte o totalmente f. Nel primo caso in generale si tende ad utilizzare variabili aleatorie uniformemente distribuite per la scelta dei campioni, mentre nel secondo caso si possono fare delle scelte *function based* per la distribuzione dei campioni, in modo da ridurre la



(A) Metodo dei trapezi (B) Metodo di *Cavalieri–Simpson*

FIGURA 8.2.1. Metodi di quadratura deterministica più comuni

varianza dell'estimatore $\langle \mathcal{J} \rangle_n$. In questa sezione ci occuperemo dapprima del secondo caso, poi del primo caso dando delle stime per la varianza che questo comporta.

8.3.1. Moltiplicatori di Lagrange. Quella dei moltiplicatori di Lagrange è una tecnica analitica per la ricerca dei punti estremali vincolati di una funzione f. Ci limitiamo, per semplicità, al caso di $\mathbb{R}^2$. Sia $A \subset \mathbb{R}^2$ aperto e siano $f : A \to \mathbb{R}$, $v : A \to \mathbb{R}$ due funzioni in $\mathcal{C}^1(A)$ con $\|\nabla v(x,y)\|^2 > 0$ per ogni $(x,y) \in A$. Indichiamo con W l'insieme degli zeri di v, detta condizione di *vincolo*, ovvero

$$W = \{(x,y) \in A \,:\, v(x,y) = 0\}\,.$$

Supponiamo che $W \neq \emptyset$, vogliamo cercare delle condizioni per i punti di massimo o minimo di $f(x,y)$ quando $(x,y) \in W$.

Sia (x_0, y_0) un punto di W tale che si verifichino una delle due condizioni

$$\partial_x v(x_0,y_0) \neq 0, \quad \partial_y v(x_0,y_0) \neq 0$$

per il teorema della funzione implicita, l'equazione $v(x,y) = 0$ definisce localmente una funzione $x = \alpha(y)$ oppure una $y = \beta(x)$ tali che $x_0 = \alpha(y_0)$ oppure $y_0 = \beta(x_0)$. Supponiamo che si abbia la seconda condizione (il ragionamento nel caso della prima condizione è del tutto analogo): allora il punto x_0 è estremale per la funzione di una sola variabile $f(x, \beta(x))$ e pertanto soddisfa la condizione $f'(x_0) = \partial_x f(x_0) + \partial_y f(x_0)\beta'(x_0) = 0$. Inoltre, derivando la condizione $v(x,\beta(x)) = 0$, si ottiene $\beta'(x) = -\left(\partial_y v(x,y)\right)^{-1} \partial_x v(x,y)$ in un intorno in cui $y = \beta(x)$. Ne segue che

$$\partial_x f(x_0,y_0) - \partial_y f(x_0,y_0) \frac{\partial_x v(x_0,y_0)}{\partial_y v(x_0,y_0)} = 0. \tag{8.3.1}$$

Poniamo

$$\Psi(x,y,\lambda) = f(x,y) - \lambda v(x,y). \tag{8.3.2}$$

Se in $(x_0, y_0 = \beta(x_0))$ si ha $\partial_y f \neq 0$, ne segue che esiste

$$\lambda_0 = \left(\partial_x v(x_0,y_0)\right)^{-1} \partial_y v(x_0,y_0) = \left(\partial_x f(x_0,y_0)\right)^{-1} \partial_y f(x_0,y_0)$$

soluzione del sistema

$$\nabla \Psi(x_0,y_0,\lambda_0) = 0\,. \tag{8.3.3}$$

Pertanto la ricerca di un punto estremale (x_0, y_0) di f vincolato alla condizione $v(x.y) = 0$ si può ricondurre alla ricerca dei valori di $\lambda \in \mathbb{R}$ (detti *moltiplicatori di Lagrange*), x e y, che risolvono il problema di estremo non vincolato dato dal sistema di tre equazioni in tre incognite (8.3.3).

Benché non sia del tutto immediato, è possibile estendere questo ragionamento al caso di due funzionali $F \in L^1(A)^*$ e $V \in L^1(A)^*$ ed ottenere un risultato analogo. Precisamente, se indichiamo con

$$\Lambda = \left\{ p \in L^1(A) \,:\, V(p) = 0 \right\},$$

la ricerca di $p \in \Lambda$ estremale per il funzionale F, può essere ricondotta alla ricerca delle soluzioni in $L^1(A)$ del sistema

$$\nabla \Psi(p, \lambda) = 0 \Longleftrightarrow \begin{cases} D_p F(p) - \lambda D_p V(p) = 0 \\ V(p) = 0 \end{cases} \tag{8.3.4}$$

dove $\lambda \in \mathbb{R}$, $\Psi(p, \lambda) = F(p) - \lambda V(p)$ e la derivata rispetto a p si fa formalmente come se operasse su funzioni di variabile reale. Assumiamo veri questi risultati di analisi funzionale e vediamo come si applicano al problema della riduzione dell'errore di un estimatore di Monte Carlo $\langle \mathcal{J} \rangle_n$.

Per la proposizione 8.2.3 la media dell'errore dovuto all'approssimare $\mathcal{J}$ con $\langle \mathcal{J} \rangle_n$ è dato da $E(\rho_n^2) = (1/n) \operatorname{Var}(\Phi(X))$, dove indichiamo con $\Phi(X) = f(X)p(X)^{-1}\chi_{\mathcal{Q}}(X)$. Cerchiamo la distribuzione di probabilità p che minimizzi il funzionale $E(\rho_n^2)$ con il vincolo $V(p) = \int_{\mathcal{Q}} p(x)\, dx - 1$. Il funzionale Ψ da minimizzare assume la forma

$$\Psi(p, \lambda) = \lambda + \frac{1}{n} \int_{\mathcal{Q}} \left\{ \left(\frac{f}{p} - \mathcal{J} \right)^2 p - \lambda p \right\}. \tag{8.3.5}$$

La derivata formale di Ψ rispetto a p è nulla se è nulla la derivata della funzione integranda: pertanto la distribuzione p ottimale per l'estimatore $\langle \mathcal{J} \rangle_n$ deve essere tale che

$$\partial_p \Psi(p, \lambda) = 0 \Leftrightarrow \frac{f^2}{p^2} + \lambda n - \mathcal{J}^2 = 0 \Leftrightarrow p = \frac{|f|}{\sqrt{\mathcal{J}^2 - \lambda n}} \tag{8.3.6}$$

Imponendo anche la condizione del vincolo, ovvero $\partial_\lambda \Psi(p, \lambda) = 0$, segue che $\sqrt{\mathcal{J}^2 - \lambda n} = \int_{\mathcal{Q}} |f(x)|\, dx$. Dunque la densità di distribuzione ottima per stimare l'integrale $\mathcal{J}$ con il metodo di Monte Carlo è

$$p_X(x) = \frac{|f(x)|}{\int_{\mathcal{Q}} |f(x)|\, dx} \tag{8.3.7}$$

Questa scelta per la distribuzione degli n campioni dell'estimatore darebbe luogo ad un errore a media nulla (si avrebbe $E(\rho_n^2) = 0$); tuttavia la formula ottenuta richiede essa stessa la conoscenza dell'integrale $\mathcal{J}$, ed è, pertanto, inutilizzabile. Non è possibile scegliere la distribuzione p ottima per l'estimatore di Monte Carlo. Quello che l'intuito ci suggerisce è che, in generale, la scelta di p per stimare l'integrale di f è tanto migliore quanto più le funzioni p ed f si "assomigliano", in un senso che può essere in parte chiarito osservando come è stata scelta la distribuzione congiunta $p(\theta, \varphi)$ nell'Esempio 8.2.6.

8.3.2. Campionamento stratificato. Un altro approccio possibile per ridurre la varianza dell'estimatore è quello del *campionamento stratificato*. L'idea è di suddividere il dominio di integrazione in m sotto domini e forzare i campioni aleatori a stare almeno uno in ognuno di tali sotto domini. Operare in questo modo garantisce che la varianza dell'estimatore così ottenuto sia minore che quella dell'estimatore di Monte Carlo puro. Osserviamo questo fatto nel caso unidimensionale. Cosideriamo gli $m + 1$ punti $a = \nu_0, \nu_1, \ldots, \nu_m = b$ e suddividiamo l'integrale $\mathcal{J}$ nel modo seguente

$$\mathcal{J} = \int_a^b f(x)\, dx = \sum_{k=1}^{m} \int_{\nu_{k-1}}^{\nu_k} f(x)\, dx = \sum_{k=1}^{m} \mathcal{J}_k \,. \tag{8.3.8}$$

Per ogni $k = 1, \dots, m$ introduciamo $n(k)$ variabili aleatorie $X_1^{(k)}, \dots, X_{n(k)}^{(k)}$ tali che $X_i^{(k)} : \Omega \to [\nu_{k-1}, \nu_k]$ per $i = 1, \dots, n(k)$, indipendenti identicamente distribuite, con densità $p_k = p_{X_1^{(k)}} = \cdots = p_{X_{n(k)}^{(k)}}$. Indichiamo con $\chi_k = \chi_{[\nu_{k-1}, \nu_k]}$ la funzione caratteristica dell'intervallo $[\nu_{k-1}, \nu_k]$ e costruiamo m estimatori

$$\langle \mathcal{J}_k \rangle_{n(k)} = \frac{1}{n(k)} \sum_{i=1}^{n(k)} \frac{f\left(X_i^{(k)}\right)}{p_k\left(X_i^{(k)}\right)} \chi_k\left(X_i^{(k)}\right), \quad k = 1, \dots, m.$$

L'estimatore complessivo di $\mathcal{J}$ è dato da $\left\langle \mathcal{J}^{(S)} \right\rangle = \sum_{k=1}^m \langle \mathcal{J}_k \rangle_{n(k)}$. Se le variabili $X_i^{(k)}$ ($k = 1, \dots, m$, $i = 1, \dots, n(k)$) sono indipendenti, dall'osservazione 8.1.24 e la Proposizione 8.2.3 segue che

$$\mathrm{Var}\left(\left\langle \mathcal{J}^{(S)} \right\rangle\right) = \sum_{k=1}^m \frac{1}{n(k)} \left(\int_{\nu_{k-1}}^{\nu_k} \frac{f^2(x)}{p_k(x)} dx - (\mathcal{J}_k)^2 \right).$$

Per confrontare la varianza dell'estimatore stratificato e di quello classico assumiamo che le variabili $X_i^{(k)}$ siano identicamente distribuite. Indichiamo con n il numero totale di campioni (ovvero $\sum_k n(k) = n$), e con $\langle \mathcal{J} \rangle_n$ l'estimatore di Monte Carlo standard, pesato con la stessa distribuzione di probabilità delle variabili $X_i^{(k)}$, rinormalizzata in maniera che $p(x) = 1/(b-a)$. Consideriamo il prodotto scalare in $\mathbb{C}^m$ con pesi $(n(1), n(2), \dots, n(m))$, ossia definito da

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_{k=1}^m n(k)\, u_k\, \overline{v_k}$$

per $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{C}^m$. La norma associata a questo prodotto scalare è

$$\|\boldsymbol{u}\| = \sqrt{\sum_{k=1}^m n(k) |u_k|^2},$$

e la disuguaglianza di Cauchy–Schwarz, $|\langle \boldsymbol{u}, \boldsymbol{v} \rangle| \leqslant \|\boldsymbol{u}\|^2 \|\boldsymbol{v}\|^2$, diventa

$$\left| \sum_{k=1}^m n(k) u_k \overline{v_k} \right|^2 \leqslant \sum_{k=1}^m n(k) |u_k|^2 \sum_{k=1}^m n(k) |v_k|^2.$$

Da questo e (8.3.8) segue

$$\sum_{k=1}^m \frac{1}{n(k)} (\mathcal{J}_k)^2 \geqslant \frac{1}{n} \left(\sum_{k=1}^m \mathcal{J}_k \right)^2 = \frac{1}{n} \mathcal{J}^2,$$

e quindi

$$\begin{aligned} \mathrm{Var}\left(\left\langle \mathcal{J}^{(S)} \right\rangle\right) - \mathrm{Var}\left(\langle \mathcal{J} \rangle_n\right) & \\ \leqslant \sum_{k=1}^m \left(\frac{1}{n(k)} \int_{\nu_{k-1}}^{\nu_k} \frac{f^2(x)}{p_k(x)}\, dx \right) - \frac{1}{n} \int_a^b \frac{f^2(x)}{p(x)}\, dx & \\ = \sum_{k=1}^m \int_{\nu_{k-1}}^{\nu_k} \left(\frac{\nu_{k+1} - \nu_k}{n(k)} - \frac{b-a}{n} \right) f^2(x)\, dx\,. & \quad (8.3.9) \end{aligned}$$

Vorremmo una condizione sufficiente affinché, per ogni f, la varianza si riduca facendo uso dell'estimatore stratificato. Questo è vero se e solo se la quantità (8.3.9) è non positiva, ossia

$$\frac{b-a}{n} \geqslant \frac{\nu_{k+1} - \nu_k}{n(k)} \quad \forall\, k. \qquad (8.3.10)$$

La condizione (8.3.10) va studiata caso per caso, tuttavia esiste almeno una scelta banale degli intervalli della partizione, delle variabili $X_i^{(k)}$ e dei campioni $n(k)$ per cui è soddisfatta. Infatti, supponiamo che:

- gli intervalli siano tutti della stessa lunghezza;
- le variabili $X_1^{(k)}, \dots, X_{n(k)}^{(k)}$ siano uniformemente distribuite nei loro rispettivi intervalli $[\nu_{k-1}, \nu_k]$;
- in ognuno di questi intervalli si estragga un solo campione (ovvero $n(k) = 1$: ma l'argomento vale in maniera identica se si estrae lo stesso numero di campioni in ogni intervallo, non necessariamente uno).

In queste ipotesi $m = n$ e $\nu_k - \nu_{k-1} = (b-a)/m = (b-a)/n$: quindi in (8.3.10) vale esattamente l'uguaglianza. Quindi, se stratifichiamo in questa maniera equiripartita, l'estimatore stratificato garantisce una stima migliore dell'estimatore classico (comporta un errore medio minore), indipendentemente dalla funzione cui questo si applichi.
Più in generale, la condizione (8.3.10) vale se $\frac{n(k)}{n} \geqslant \frac{\nu_{k+1}-\nu_k}{b-a}$ per ogni k, ma questo non è possibile perché la somma di entrambi i membri su tutti i k vale 1. Il meglio che si possa ottenere è che il numero di campioni per ciascun sottointervallo sia scelto in proporzione alla lunghezza del sottointervallo, ossia

$$\frac{n(k)}{n} = \frac{\nu_{k+1} - \nu_k}{b-a} \quad \forall\, k.,$$

ma poiché i numeri $n(k)$ sono interi questo in generale non è possibile esattamente, ma solo approssimativamente. Se questa identità vale approssimativamente, allora l'estimatore stratificato ha varianza minore di quello non stratificato, oppure le due varianze sono approssimativamente uguali. In generale, se la disuguaglianza $\frac{n(k)}{n} \geqslant \frac{\nu_{k+1}-\nu_k}{b-a}$ vale per alcuni k ma non per altri, allora segue da (8.3.9) che la varianza dell'estimatore stratificato è minore della varianza dell'estimatore non stratificato non per tutte le funzioni f, bensì per quelle che sono sufficientemente grandi negli intervalli $[\nu_{k-1}, \nu_k]$ con i k tali che valga la disuguaglianza ma sufficientemente piccole (ad esempio vicine a zero) per gli altri k. Questo significa che l'estimatore deve avere numeri di campioni $n(k)$ elevati negli strati in cui f ha varianza elevata. Ad esempio, l'estimatore stratificato per l'integrale di una fissata funzione f ha varianza minima se manda uguali numeri di campioni in sottointervalli di uguale variazione dei valori di f.

8.4. Come generare campioni di variabili aleatorie con una prefissata distribuzione di probabilità

Abbiamo presentanto i caratteri generali del metodo di Monte Carlo per l'integrazione numerica. Al fine di implementare tale metodo è necessario poter generare valori casuali entro un intervallo con una particolare distribuzione di probabilità, essendo noto come generare campioni uniformemente distribuiti in maniera casuale (o, più precisamente, *pseudo-casuale*). Consideriamo dapprima il caso discreto e poi quello continuo, nonostante il modo di operare sia sostanzialmente analogo.

Variabili discrete. Consideriamo la variabile aleatoria discreta U, uniformemente distribuita nell'intervallo $[0, 1)$. Data una variabile aleatoria discreta X con distribuzione p_X, che assume valori $1, 2, \dots, n, \dots$ vogliamo selezionare un campione j con probabilità $p_X(j)$. Se F_X è la funzone di ripartizione di X, dalla (8.1.6) risulta $p_X(i) = F_X(i) - F_X(i-1)$ per ogni i nell'immagine di X, inoltre, essendo U uniformemente distribuita, per ogni intervallo $[\alpha, \beta) \subset [0, 1)$ si ha $\mathbf{P}(U \in [\alpha, \beta)) = F_U(\beta) - F_U(\alpha) = \beta - \alpha$, ne segue che la probabilità che un campione j nell'immagine di U sia tale che $F_X(j) \leqslant U \leqslant F_X(j-1)$ è

$$\mathbf{P}\Big(U \in \Big[F_U(j-1), F_U(j)\Big)\Big) = F_U(j) - F_U(j-1) = p_X(j),$$

ovvero la probabilità voluta.

Pertanto, un possibile algoritmo per la generazione di campioni con una data distribuzione di probabilità p è il seguente:

Algorithm 1 Generare un campione k con probabilità $p_X(k)$

1: Sia X aleatoria con distribuzione p_X.
2: $u = random(0,1)$
3: $k = random(0,1)$
4: **while** $u \notin \big[F_X(k-1), F_X(k)\big)$ **do**
5: $\quad k = random(0,1)$
6: **end while**
7: k è stato scelto con probabilità $p_X(k)$.

Variabili continue. Nel caso di variabili aleatorie continue si procede in maniera del tutto analoga. Supponiamo di voler costruire un campione X con densità p_X, ovvero tale che $\mathbf{P}(X \leqslant x) = \int_{-\infty}^{x} p_X(\xi)\, d\xi$, nota la densità p_X. Se U è uniformemente distribuita in $[0,1]$ allora $F_U(u) = \mathbf{P}(U \leqslant u) = u$, posto $X = F_X^{-1}(U)$, essendo una funzione di ripartizione non-decrescente, si ha

$$\mathbf{P}(X \leqslant x) = \mathbf{P}(F_X^{-1}(U) \leqslant x) = \mathbf{P}(U \leqslant F_X(x)) = F_X(x) = \int_{-\infty}^{x} p_X(\xi)\, d\xi$$

Dunque basta scegliere $X = F_X^{-1}(U)$, con U variabile aleatoria unforme su $[0,1]$, per ottenere un campione con la probabilità cercata. Si noti, tuttavia, che tale metodo per la generazione di campioni aleatori richiede di conoscere analiticamente la funzione di ripartizione F_X e di saperla invertire, cosa che non è sempre possibile.

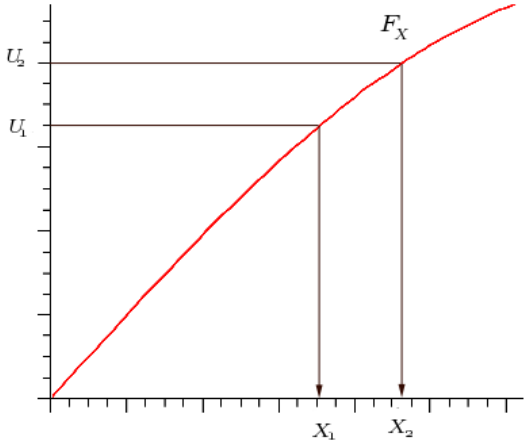


FIGURA 8.4.1. Campionamento di variabili aleatorie continue

ESEMPIO 8.4.1. Vediamo come, nel caso dell'esempio 8.2.6, avremmo potuto produrre dei campioni sull'emisfero che fossero distribuiti secondo la densità scelta. Parametrizziamo l'emisfero con gli angoli di Eulero $0 \leqslant \varphi < 2\pi$, $0 \leqslant \theta < \pi$.

Per ridurre l'errore di approssimazione con l'estimatore (ovvero ridurre la varianza) avevamo scelto intuitivamente la distribuzione

$$p(\xi,\zeta)=\frac{1}{\pi}\cos(\zeta)\sin(\zeta).$$

Calcoliamo la sua funzione di ripartizione $F(\theta,\varphi)$

$$\begin{aligned}F(\theta,\varphi)=\frac{1}{\pi}\int_0^\theta\int_0^\varphi p(\xi,\zeta)\,d\zeta\,d\xi&\\ =\frac{1}{\pi}\int_0^\theta d\xi\int_0^\varphi\cos\zeta\sin\zeta\,d\zeta=&\\ =\frac{\theta}{2\pi}\left(1-\cos^2\varphi\right)=f(\theta)g(\varphi).&\quad(8.4.1)\end{aligned}$$

Osserviamo che esistono f e g tali che $F=f\cdot g$, ne segue che le variabili aleatorie θ, φ sono indipendenti. Pertanto possiamo considerare due generatori casuali $U:\Omega\to\{u_i\}_i$ e $V:\Omega\to\{v_i\}_i$, indipendenti, uniformemente distribuiti in $[0,1]$, e calcolare i campioni invertendo le funzioni $f(x)=x/2\pi$ e $g(x)=1-\cos^2 x$. Si noti che la funzione g è la composizione delle funzioni $u:x\mapsto\cos x$, $v:y\mapsto y^2$ e $w:z\mapsto 1-z$, in questo ordine, e che l'inversa di $v(y)$ è la funzione $\sqrt{y}$ (la radice va presa col segno positivo, perché la variabile y qui vale $\cos x=\cos\theta$ e nel dominio assegnato per x, ossia $0\leqslant x=\theta<\pi$, la quantità $\cos x$ è non negativa). Infine, la funzione w coincide con la propria inversa (se la si applica due volte si trova l'identità, come è anche ovvio dal fatto che w è la riflessione rispetto al punto $1/2$). Pertanto la funzione inversa di $g(x)=1-\cos^2 x$ è $g^{-1}(v)=\arccos\left(\sqrt{1-v}\right)$. Riassumendo,

$$\theta_i=f^{-1}(u_i)=2\pi u_i,\quad \varphi_i=g^{-1}(v_i)=\arccos\left(\sqrt{1-v_i}\right),$$
$$i=1,2,3,\dots\,.$$

Tali valori saranno distribuiti secondo la densità p inizialmente considerata. □

8.4.1. Un metodo euristico: Rejection sampling. Non sempre è possibile inverire la funzione di ripartizione di una varibile aleatoria, ancor più se questa è definita su uno spazio a più dimensioni (se si tratta di un vettore aleatorio, come nell'esempio poc'anzi proposto). Vi è un altro metodo per generare campioni con una distribuzione assegnata, detto *rejection sampling*.

Consideriamo il vettore aleatorio $X=(X_1,\dots,X_n)$, con densità di distribuzione congiunta $p:\mathcal{Q}\subset\mathbb{R}^n\longrightarrow[0,\mu]$. Indichiamo con $\mathcal{S}(p)$ il sottografico di p, ovvero l'insieme

$$\mathcal{S}(p):=\{(\mathbf{x},\gamma)\in\mathcal{Q}\times[0,\mu]\mid p(\mathbf{x})\geqslant\gamma\},\tag{8.4.2}$$

il metodo consiste nel generare campioni $\mathbf{u}_i=\left(u_1^{(i)},\dots,u_n^{(i)},u_{n+1}^{(i)}\right)$ uniformemente distribuiti in $\mathcal{Q}\times[0,\mu]$ considerando, al variare di i, solo quelli tali che $\mathbf{u}_i\in\mathcal{S}(p)$, scartando gli altri. La distribuzione dei campioni così considerati è una buona approssimazione della distribuzione p che volevamo campionare. Questo metodo, d'altra parte, è fortemente condizionato dalla forma della funzione p o, analogamente, dall'insieme $\mathcal{S}(p)$. Infatti se $|\mathcal{S}(p)|<<\mu|\mathcal{Q}|$, la maggior parte dei campioni $\mathbf{u}_i$ generati viene scartata e per ottenere una distribuzione che bene approssimi p sono neccessari molti campioni, causando un aumento consistente della varianza del metodo.

8.5. Esempi di generazione di variabili aleatorie

Si è visto che il metodo di Monte Carlo utilizza variabili aleatorie che hanno una certa PDF, ma non è stato ancora detto come queste variabili aleatorie vengano generate. Per affrontare questo problema partiremo dalla distribuzione più semplice, quella uniforme, e vedremo come da questa si possano ricavare variabili aleatorie con una PDF qualsiasi.

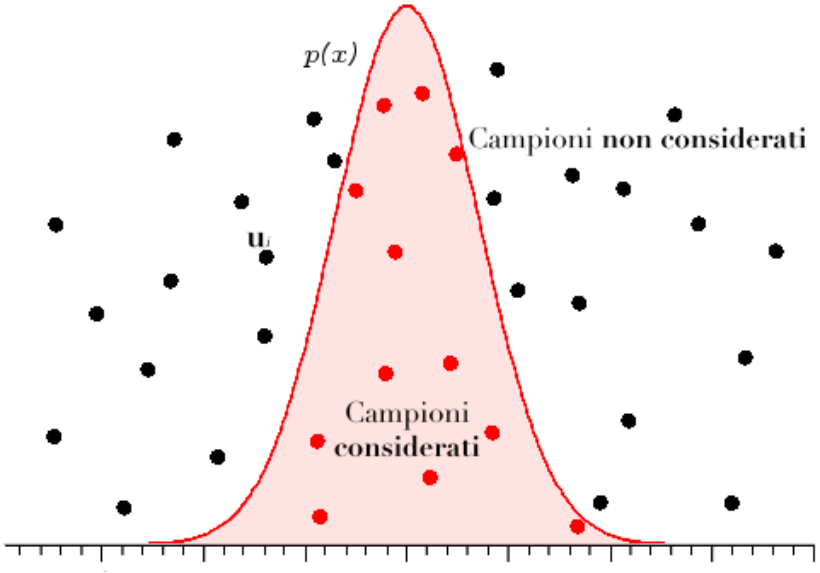


FIGURA 8.4.2. Rejection sampling

8.5.1. La distribuzione continua uniforme. La distribuzione continua uniforme è una densità di distribuzione che attribuisce la stessa probabilità a due insiemi A,$B \in \mathcal{A}$ della stessa misura. Supponiamo ad esempio che la nostra variabile aleatoria sia uniformemente distribuita in un intervallo $[a, b]$, allora in questo intervallo la PDF $p(x)$ della variabile deve rimanere costante e deve annullarsi al di fuori di esso.

$$p(x) = \begin{cases} c & \text{se } a < x < b, \\ 0 & \text{altrimenti.} \end{cases}$$

Essa deve essere tale che:

$$\int_{\mathbb{R}} p(x)\,dx = \int_a^b c\,dx = c\int_a^b dx = c(b-a) = 1$$

La PDF di una variabile aleatoria uniformemente distribuita in $[a, b]$ è:

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{se } a < x < b, \\ 0 & \text{altrimenti.} \end{cases}$$

In generale ci basterà conoscere la misura dell'insieme su cui la variabile aleatoria deve essere distribuita uniformemente:

$$p(x) = \begin{cases} \frac{1}{m(A)} & \text{se } x \in A, \\ 0 & \text{altrimenti.} \end{cases} \qquad \text{con } A \in \mathcal{A}. \tag{8.5.1}$$

Per quanto riguarda gli intervalli osserviamo che possiamo ricondurci a variabili aleatorie distribuite uniformemente su qualsiasi intervallo $[a, b]$ semplicemente scalando e traslando una variabile aleatoria uniforme su $[0, 1]$. Per ottenere questa variabile abbiamo bisogno di un generatore di numeri pseudo-casuali, ovvero un algoritmo deterministico che produce una sequenza con le stesse proprietà statistiche di una sequenza di numeri generata da un processo casuale. I processi di generazione di numeri pseudo-casuali non sono tutti uguali e possono essere valutati in base a dei precisi requisiti:

(1) I numeri generati devono essere uniformemente distribuiti e statisticamente indipendenti nella sequenza.

(2) La sequenza generata deve essere riproducibile, in modo che ripetendo il processo non otteniamo un risultato diverso.
(3) La sequenza deve poter avere un periodo di lunghezza il più ampio possibile; le sequenze pseudo-casuali infatti possono essere periodiche, ciò avviene se esiste un q intero e positivo tale che:
$$X_{n+q} = X_n \qquad \text{per ogni } n > 0,$$
Il minimo intero q per cui questo si verifica, è detto *periodo della successione*.

Nel programma si può scegliere tra tre alternativi tipi di generazione randomica: `RND`, `LCG` e `SOB`.

```
<<utilities.h>>
```

```
// tipologie di generazione di variabili aleatorie in [0,1]
enum Rand_t { RND, LCG, SOB };
```

8.5.1.1. *RND.* In questo caso il generatore utilizza la funzione `rand` nativa nel C. Questa funzione, dopo l'impostazione di un seme iniziale attraverso la funzione `srand`, restituisce numeri pseudo-casuali tra 0 e la costante `RAND_MAX`. Per poter ottenere un numero uniformemente distribuito in $[0, 1]$ è necessario quindi dividere il valore restituito con `RAND_MAX`.

```
<<utilities.h>>
<<inline float generateRandom>>
```

```
    case RND:

        if(samps==1){
            /* se la generazione riguarda il primo campione bisogna impostare il
               seme */
            srand(x);
        }
        //uso della funzione rand() normalizzata in [0,1]
      return (float)rand()/RAND_MAX;
```

8.5.1.2. *LCG.* `LCG`, acronimo di *Linear Congruential Generator*, è l'algoritmo più vecchio e più utilizzato nella generazione di numeri casuali. Anche questo generatore ha bisogno di un seme iniziale[1] ed è definito ricorsivamente come:
$$X_{n+1} = (aX_n + c) \bmod m$$
dove:

- X_n è l'n-esimo valore della successione di numeri peseudo-casuali, il seme iniziale è quindi X_0.
- $m > 0$, preferibilmente un numero primo, è il *modulo*[2]. Quando il periodo della successione è uguale a m si ha il *massimo periodo*, ovvero il massimo numero di numeri pseudocasuali che si possano ottenere senza ripetizione.
- a $(0 < a < m)$ è il *moltiplicatore*

[1] in realtà il seme iniziale stesso dovrebbe essere un numero pseudo-casuale; nel programma, però, questo particolare viene tralasciato.

[2] Nel linguaggio C la funzione modulo è rappresentata dal simbolo `%`, anziché mod, e restituisce il resto della divisione tra le due variabili. Per convenzione si pone $a \bmod 0 = a$.

- c $(0 \leqslant c < m)$ è l'*incremento*

La qualità dell'algoritmo è molto influenzata dalla scelta di questi parametri, tutti naturali e maggiori di 0, che danno vita ad una `LCG` più o meno buona. Come vantaggi offre velocità, in quanto è computazionalmente molto leggero, e semplicità di implementazione. Gli svantaggi sono però una generazione di variabili correlate tra loro (otteniamo la variabile successiva tramite la precedente) ed un periodo limitato da m.
Analizziamo più nel dettaglio questa funzione: la relazione di congruenza è definita dal modulo con m e si scrive

$$a \equiv b \qquad \text{se} \qquad a \mod m = b \mod m.$$

e garantisce che il numero generato non superi mai il valore di $(m-1)$. Può succedere però che la generazione di numeri si ripeta da capo (ovvero che il periodo finisca) prima che tutti gli m numeri siano stati scelti. Per evitare questo problema prendiamo in considerazione il teorema che fornisce le condizioni necessarie e sufficienti affinché la sequenza abbia una lunghezza m massima (massimo periodo), per la cui dimostrazione rinviamo il lettore a [**8**].

Teorema 8.5.1 (Hull e Dobell, 1962). *La sequenza lineare congruenziale definita da m, c, a e X_0 ha lunghezza del periodo m se e solo se:*

(1) *c è relativamente primo rispetto a m;*
(2) *$a-1$ è un multiplo di p, per ogni primo p che divide m;*
(3) *$a-1$ è un multiplo di 4, se m è un multiplo di 4.*

Esempio: Se scegliamo $c = 5$, $a = 13$, $X_0 = 0$, $m = 12$, in modo che vengano soddisfatte le tre condizioni, la sequenza generata sarà:

$$5, 10, 3, 8, 1, 6, 11, 4, 9, 2, 7, 0, 5, \ldots$$

e quindi la lunghezza del periodo del generatore lineare congruenziale è uguale a m. Nel programma sono stati scelti come parametri:

- $c = 1013904223$
- $a = 1664525$
- $m = 4294967296$

restituiamo infine una variabile in $[0, 1]$ semplicemente dividendo per $(m-1)$.

```
<<utilities.h>>
```

```
// Linear congruential generator
//restituisce il prossimo campione randomico
inline void lcg(unsigned int &x){
   x=(1664525u*(x)+1013904223u)%4294967296u;
}
```

```
<<utilities.h>>
<<inline float generateRandom>>
```

```
case LCG:
    //viene richiamata la funzione per il calcolo dell'LCG
       lcg(x);
    //si divide per il numero massimo raggiunto dal metodo LCG
           return float(x)/4294967295.0f;
```

8.5.1.3. *SOB.* Per poter valutare l'uniformità di una successione di numeri è stato sviluppato il concetto della *discrepanza*. L'idea di base è che la "qualità" di un insieme di punti contenuti nell'ipercubo $[0,1]^n$ può essere valutata osservando i sotto-insiemi di tale ipercubo e determinando quanto la proporzione di punti in tale sotto-insieme differisca dalla misura dell'insieme.

DEFINIZIONE 8.5.2 (Discrepanza). Data una famiglia B di sottoinsiemi di $[0,1]^n$ e un insieme di punti $P = x_1, \ldots, x_s$, definiamo la *discrepanza*

$$D_s(B,P) = \sup_{b \in B} \left| \frac{\#\{x_i \in b\}}{s} - V(b) \right|,$$

dove $\#\{x_i \in b\}$ sta ad indicare il numero di punti di P all'interno di b e $V(b)$ corrisponde al volume di b.

Il valore $\frac{\#\{x_i \in b\}}{s}$ è utilizzato come approssimazione del volume di b. La discrepanza è quindi l'errore più grande che possiamo ottenere approssimando il volume in questo modo.

Esempio:
La discrepanza che si ottiene utilizzando la famiglia di sottoinsiemi

$$B = \{[0, v_1] \times [0, v_2] \times .... \times [0, v_n]\}$$

con $v_1, \ldots v_n \in [0,1]$, otteniamo una viene chiamata $D_s(B,P)^*$.

Esistono numerose tecniche per ottenere sequenze a bassa discrepanza in modo deterministico. Nel programma impostando la generazione random su `SOB` si potrà utilizzerà la sequenza SOBOL. Questa lavora con i numeri binari e aggiunge due ulteriori condizioni che garantiscono l'uniformità della sequenza:

- ogni segmento binario della sequenza n-dimesionale di lunghezza 2^n ha esattamente un elemento in ognuno dei 2^n ipercubi, creati dividendo l'ipercubo iniziale successivamente a metà in ogni sua dimensione;
- ogni segmento binario della sequenza n-dimesionale di lunghezza 4^n ha esattamente un elemento in ognuno dei 4^n ipercubi, creati dividendo l'ipercubo iniziale successivamente in quattro parti uguali in ogni sua dimensione.

Nel programma è stato inserito il codice open source presente in [`?sob`] che è contenuto nei file *sobol.h* e *sobol.cpp*.

8.5.2. Campionamento uniforme su emettitori diffusivi per il calcolo dell'illuminazione diretta. In questa Sottosezione e nella prossima, anticipiamo metodi di rendering per l'illuminazione rispettivamente diretta ed indiretta, incluso il loro codice nel linguaggio C, che in seguito diventeranno parte del codice di Illuminazione Globale sviluppato nei prossimi capitoli. Per una comprensione più completa rimandiamo il lettore a quei capitoli.

Per calcolare tramite estimatore di Monte Carlo l'illuminazione diretta del punto $\boldsymbol{x}$ illuminato da un emettitore, dobbiamo definire una densità di distribuzione $p(\boldsymbol{y})$ sui punti dell'emettitore. Nel nostro codice le luci sono oggetti come tutti gli altri, a questi viene però assegnato un materiale con parametro L_e non nullo. Nei brandelli di codice qui sotto le luci emettono in maniera diffusiva[3] da

[3]quindi in egual misura in tutte le direzioni dell'emisfero.

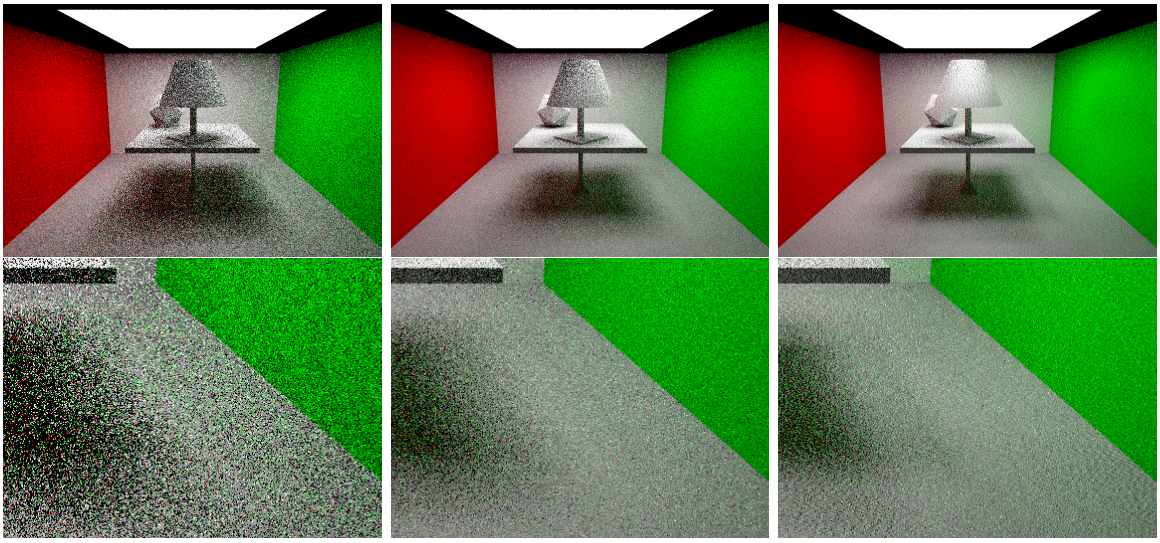


FIGURA 8.5.1. **confronto tra i diversi metodi di generazione randomica**: per la realizzazione delle immagini si è utilizzato il Ray Tracing stocastico con un numero di sample pari a 5. A partire dalla colonna di sinistra fino a quella di destra i metodi utilizzati sono stati `RND,LCG,SOB`. Nella seconda riga è stato effettuato un ingrandimento per riuscire a visualizzare meglio le differenze tra le tre immagini. Notiamo inoltre come un buon algoritmo di generazione randomica uniforme genera campioni anche vicino agli estremi del dominio, infatti solo nella terza immagine, realizzata con la sequenza a bassa discrepanza SOBOL, viene visualizzata la luce che colpisce il paralume.

tutta la loro superficie, che chiameremo S. Il valore L_e rappresenta il valore di radianza uscente dalla luce

$$L_e = L_e(\boldsymbol{y}_i \to \overrightarrow{\boldsymbol{y_i x}}).$$

Segue da (7.4.1) che, per ottenere la potenza della luce dobbiamo moltiplicare per $\pi A(S)$, dove $A(S)$ rappresenta l'area della superficie S. Poiché l'emissione è diffusiva, al momento della creazione del materiale, viene scelto direttamente il valore di radiosità (B_e) della superficie che sarà poi diviso per π dal costruttore del materiale, ricavandoci così automaticamente il parametro L_e. La densità di distribuzione scelta è uniforme lungo tutto S, dipende quindi dall'area di questa superficie ed è definita come:

$$p(\boldsymbol{y}) = \frac{1}{A(S)}$$

Applicando il metodo di Monte Carlo all'equazione (7.8.2) del rendering per l'illuminazione diretta, otteniamo il seguente estimatore:

$$\begin{aligned}\langle L_{\text{dir}}(\boldsymbol{x} \to \boldsymbol{\theta})\rangle &= \frac{1}{N_s}\sum_{i=1}^{Ns} \frac{L_e(\boldsymbol{y}_i \to \overrightarrow{\boldsymbol{y_i x}})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{x y_i}})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i)}{p(\boldsymbol{y}_i)} \\ &= .\frac{1}{N_s}\sum_{i=1}^{Ns} L_e(\boldsymbol{y}_i \to \overrightarrow{\boldsymbol{y_i x}})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{x y_i}})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i) A(S)\end{aligned}$$

Per ogni punto $\boldsymbol{y}_i$ della sorgente dobbiamo calcolare il valore dell'energia irradiata al punto $\boldsymbol{x}$. Per fare questo è necessario tenere in considerazione la BRDF del punto $\boldsymbol{x}$, il fattore di visibilità V, il fattore geometrico G e la radianza emessa nel punto $\boldsymbol{y}_i$. La varianza dipende direttamente dal parametro `dirsamps`, cioè il numero di campioni usati, definito nel «`main.h`» e viene visualizzata sotto forma di rumore. Questo, in particolare, avviene nelle zone di penombra dove non tutti i campioni avranno il fattore di visibilità V non nullo. In presenza di più luci la scelta di quella da campionare può essere casuale, utilizzando o una PDF uniforme o basandosi sulla potenza delle luci. Qui di seguito proponiamo il codice relativo all'illuminazione diretta con campionamento su ciascuna luce della scena[4].

```
<<srt.cpp>>
```

```
//Calcolo dell'illuminazione diretta.
//I parametri in input sono:
// r raggio di entrata
// o puntatore all'oggetto dal quale partirà il nuovo raggio
// x e y indici del seme iniziale per le generazioni randomiche RND e LGC
float3 directIllumination(Ray& r,Obj* o,int x,int y){
    //si inizializzano le variabili aleatorie usate per campionare la luce in modo
        equidistribuito
    float rnd1=0;
    float rnd2=0;
    float rnd3=0;
    //si inizializza il valore in uscita dal processo
    float3 radianceOutput=float3(0);
    //si carica la normale all'oggetto nel punto r.o
    float3 n1= o->normal(r.o);
    //si carica l'identificativo del materiale
    int mId= o->matId;
    //si alloca la memoria per il puntatore all'oggetto che sarà intersecato
    Obj** objX=new Obj*();

    //per ogni luce
    for(int i=0; i<nLight; i++){

        //si carica l'area della luce in esame
        float area= luci[i]->areaObj;
        //per ogni campione, s, della luce (per un massimo di dirsamps campioni )
        for(int s=0; s<dirsamps; s++){

            //si utilizzano tre parametri per campionare uniformemente la
                superficie della luce
            //metodo SOB
            if(aST==SOB){
                rnd1=generateRandom(aoSId[0],3,aST);
                rnd2=generateRandom(aoSId[0],4,aST);
```

[4] in scene con numerose luci però è essenziale effettuare una scelta randomica della luce.

```
            rnd3=generateRandom(aoSId[0],4,aST);
        }
        else{
        //metodi RND e LCG
            rnd1= generateRandom(dirSamplesX[x+y*w],s+1,aST);
            rnd2= generateRandom(dirSamplesX[x+y*w],s+1,aST);
            rnd3= generateRandom(dirSamplesX[x+y*w],s+1,aST);
        }

        //si carica all'interno di p il punto campionato
        float3 p= luci[i]->randomPoint(rnd1,rnd2,rnd3);
        //la direzione del nuovo raggio é quella che congiunge il punto r.o al
            punto campionato
        float3 dir= (p-r.o);
        //si salva la distanza tra i due punti
        float norma= dir.normval();
        dir.norm();
        //si crea il raggio d'ombra diretto verso la luce
        Ray directRay(r.o,dir,SHADOW);

        //si inizializza objX con il puntatore all'oggetto che vorremo
            intersecare
        *objX=luci[i];
        //si inizializza la massima distanza a cui il raggio può arrivare
        float t=inf;

        //verifica del fattore di visibilità
        if(intersect(directRay,t,objX)){

            //vengono caricati i dati della luce:
            //normale nel punto p
            float3 n2=luci[i]->normal(p);
            //identificativo del materiale della luce:
            int lid= luci[i]->matId;
            //si calcola la BRDF nel punto osservato
            float3 BRDF= material[mId].C_T_BRDF(directRay,r,n1);
            //calcolo dell'illuminazione diretta:
            radianceOutput=radianceOutput+material[lid].Le.mult(BRDF)
            *(area)*(-dir.dot(n1)*dir.dot(n2))/(pow(norma,2));
        }
    }
}
 delete objX;

//si divide il risultato per tutti i sample utilizzati nell'estimatore di Monte
    Carlo
radianceOutput= radianceOutput/(dirsamps*nLight);
return radianceOutput;
}
```

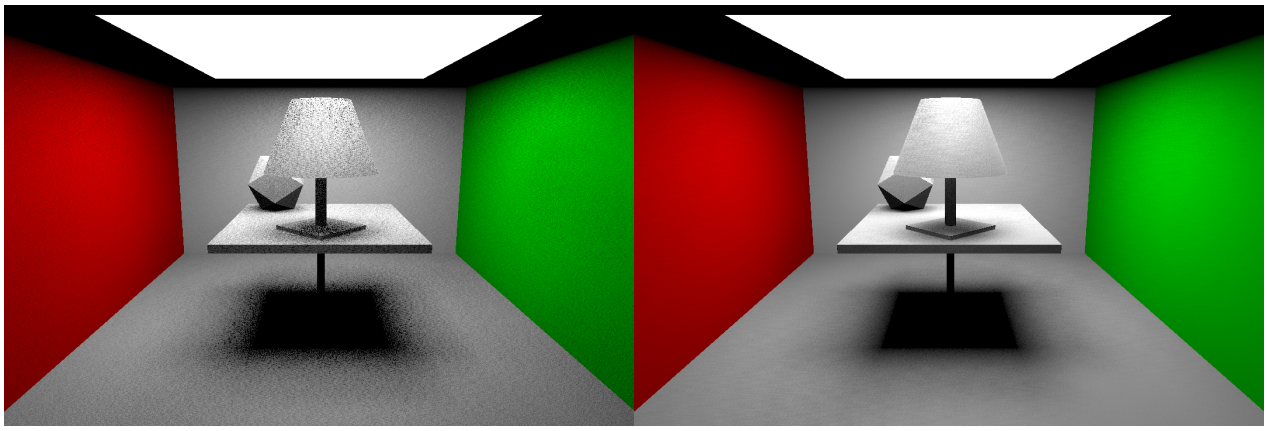


FIGURA 8.5.2. Calcolo dell'illuminazione diretta di una scena con 10 (figura a sinistra) e 100 sample (figura a destra) nell'estimatore di Monte Carlo.

8.5.3. Campionamento dell'emisfero per il calcolo dell'illuminazione indiretta. Vediamo ora come si può campionare un emisfero, utilizzando come distribuzione di probabilità il coseno dell'angolo di colatitudine, ovvero quello tra la normale alla superficie su cui poggia l'emisfero e il vettore campionato:

$$p(\xi,\zeta) = \frac{1}{\pi}\cos(\zeta)\sin(\zeta)$$

dove $\sin(\zeta)$ è lo Jacobiano della trasformazione in coordinate sferiche e π è il fattore che normalizza la PDF[5]. Parametrizziamo, quindi, l'emisfero con gli angoli di Eulero $0 \leqslant \varphi < 2\pi$, $0 \leqslant \theta < \pi$. Calcoliamo la sua funzione di ripartizione:

$$\begin{aligned} F(\theta,\varphi) &= \frac{1}{\pi}\int_0^\theta \int_0^\varphi p(\xi,\zeta)\,d\zeta\,d\xi \\ &= \frac{1}{\pi}\int_0^\theta d\xi \int_0^\varphi \cos\zeta\sin\zeta\,d\zeta \\ &= \frac{\theta}{2\pi}(1-\cos^2\varphi) = f(\theta)g(\varphi). \end{aligned}$$

Osserviamo che esistono f e g tali che $F = f\cdot g$, ne segue che le variabili aleatorie θ, φ sono indipendenti. Pertanto possiamo considerare due generatori casuali $U : \Omega \to \{u_i\}_i$ e $V : \Omega \to \{v_i\}_i$ indipendenti, uniformemente distribuiti in $[0,1]$, e calcolare i campioni invertendo le funzioni $f(x) = x/2\pi$ e $g(x) = 1-\cos^2 x$. Si noti che la funzione g è la composizione delle funzioni $u : x \mapsto \cos x$, $v : y \mapsto y^2$ e $w : z \mapsto 1-z$, in questo ordine, e che l'inversa di $v(y)$ è la funzione $\sqrt{y}$. Infine, la funzione w coincide con la propria inversa (se la si applica due volte si trova l'identità, come è anche ovvio dal fatto che w è la riflessione rispetto al punto $1/2$). Pertanto la funzione inversa di $g(x) = 1-\cos^2 x$ è $g^{-1}(v) = \arccos\left(\sqrt{1-v}\right)$.
Riassumendo,

$$\theta_i = f^{-1}(u_i) = 2\pi u_i, \quad \varphi_i = g^{-1}(v_i) = \arccos(\sqrt{1-v_i}), \tag{8.5.2}$$

$$i = 1,2,3,\ldots\,. \tag{8.5.3}$$

Tali valori saranno distribuiti secondo la densità p inizialmente considerata. Utilizziamo questa PDF nell'estimatore di Monte Carlo per il calcolo della radianza indiretta nella formulazione emisferica

[5] l'integrale di $\cos(\zeta)$ su un emisfero è infatti uguali a π.

(equazione 7.8.1 a pagina 349); dopo aver cancellato il fattore coseno al numeratore abbiamo:

$$< L_{\text{ind}}(x \to \mathbf{\Theta}) >_n = \frac{\pi}{n} \sum_{i=1}^{n} \frac{f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta})\, L_i(x \leftarrow \mathbf{\Psi})\, \langle N_x, \mathbf{\Psi} \rangle}{\langle N_x, \mathbf{\Psi} \rangle}$$

$$= \frac{\pi}{n} \sum_{i=1}^{n} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta})\, L_i(x \leftarrow \mathbf{\Psi})\,.$$

Notiamo però che questa definisce un'equazione ricorsiva: per poter fermare questo procedimento senza creare errori di bias dobbiamo utilizzare la tecnica della *Roulette Russa*, che spiegheremo fra poco nella Sottosezione 9.3.2, ma anticipiamo qui. Preso un certo numero α, che equivale alla probabilità che la ricorsività si arresti (probabilità di assorbimento), al momento dell'estrazione di un campione X uniforme in $[0, 1]$, per il calcolo dell'estimatore di Monte Carlo, si verifica che sia:

$$X \leqslant \alpha \tag{8.5.4}$$

Poiché X è distribuita uniformemente, la probabilità che questo evento si verifichi è:

$$P(X \leqslant \alpha) = \alpha$$

se la (8.5.4) è soddisfatta allora il metodo continua. È indispensabile effettuare un cambiamento per evitare che si verifichi un bias: i valori per cui $X > \alpha$ non sono mai assunti dalla funzione integranda, per questo scaliamo la funzione in modo che il valore atteso resti immutato. L'integrale diventa quindi:

$$I_{RR} = \int_0^a \frac{1}{a} f\left(\frac{x}{a}\right) dx\,.$$

Nel programma questa operazione viene svolta numerose volte nel calcolo dell'illuminazione indiretta ed è parte essenziale del raytracing stocastico. La probabilità di assorbimento della Roulette Russa è data dal prodotto di un peso, il quale assicura che la probabilità di assorbimento sia sempre minore di 1, e il massimo tra le tre componenti RGB del coefficiente di diffusione del materiale in esame (K_d). Questo ultimo fattore ci permette di assegnare maggiore probabilità ai calcoli che daranno maggiore contributo, infatti alla fine del processo l'illuminazione indiretta deve essere moltiplicata per la BRDF del materiale che, in quanto diffusiva, è gestita principalmente dal fattore K_d.
Nonostante la Roulette Russa, nel programma deve essere inserito un numero massimo di interriflessioni, `MAX_DEPTH`, che fermano in modo assolutamente certo il processo. Questo numero dipende infatti dalla memoria e dalla capacità del computer che fa girare il programma. Il numero di campioni usati per stimare l'integrale ricorsivo di illuminazione indiretta è chiamato `aosamps`, utilizzare questo parametro è però sconsigliato, in quanto comporterebbe un rallentamento esponenziale se si volessero visualizzare le riflessioni multiple nella scena. Per questa ragione, nel calcolo ricorsivo dell'illuminazione indiretta, per ogni raggio devono essere creati un numero di raggi pari al valore di `aosamps`, per un totale di raggi uguale a `aosamps`^{n}, dove n è il numero di iterazioni gestito dalla Roulette Russa. Di conseguenza vengono utilizzati, solitamente, più sample per ciascun pixel, in modo che, per ogni interriflessione, il numero di raggi usati sia sempre uguale al numero di sample. Per ogni sample del pixel, infatti, si utilizza solamente un raggio di illuminazione indiretta (impostando *aosamps* = 1 nel «main.h»). Inseriamo di seguito il codice relativo al calcolo di illuminazione indiretta tramite estimatore di Monte Carlo:

```
<<srt.cpp>>
```

```
//Calcolo dell'illuminazione indiretta.
//I parametri in input sono:
```

```
// r raggio di entrata
// o puntatore all'oggetto dal quale partirà il nuovo raggio
// x e y indici del seme iniziale per le generazioni randomiche RND e LGC
// n numero totale delle riflessioni effettuate
float3 indirectIllumination(Ray& r,Obj* o,int x,int y,int& n){
    //si inizializza il valore in uscita dal processo
    float3 radianceOutput=float3(0);
    //si calcola la normale dell'oggetto in esame nel punto r.o
    float3 n1= o->normal(r.o);
    //viene caricato l'identificativo del materiale dell'oggetto o
    int mId= o->matId;
    //viene considerato un peso per la roulette russa che andrà a moltiplicare la
        probabilità di assorbimento data dal fattore Kd
    float peso=0.9;

    //si calcola il coefficente di assorbimento per la roulette russa. Esso varia
        con il coefficiente di riflessione diffusiva Kd, quindi nel caso il raggio
        intersechi un oggetto che assorbe tutta la luce le riflessioni si
        fermeranno immediatamente, infatti il calcolo dell'illuminazione indiretta
        non darebbe alcun contributo
    //il totale del coefficente è quindi il valore massimo tra le componenti RGB
        del Kd del materiale moltiplicato per il peso
    float alpha= material[mId].Kd.max()*peso;
    float rndX=0.0f;
    float rndY=0.0f;

    //si genera una variabile aleatoria uniforme in [0,1]
    //attraverso la variabile aST nel main.h si può scegliere il metodo di
        generazione randomica
    //metodo SOBOL
    if(aST==SOB){
        rndX=generateRandom(aoSId[0],3,aST);
        rndY=generateRandom(aoSId[0],4,aST);
    }
    else{
        // per le altre due tipologie Rand_t abbiamo bisogno del numero del
            campione s
        rndX=generateRandom(aoSamplesX[x+y*w],s+1,aST);
        rndY=generateRandom(aoSamplesY[x+y*w],s+1,aST);
    }

    //Roulette Russa:
    //verifico che non ci sia assorbimento
    if(rndX<alpha){

    //si scala il valore di rndX in modo da ottenere il valore dell'integrale
        originale
    rndX=rndX*1/alpha;
    //si aumentano il numero di riflessioni totali
```

```
    n++;

    //per ogni sample s tra gli aosamps sample il cui numero può essere impostato
        nel main.h
    for(int s=0; s<aosamps; s++){

           //sono create le variabili aleatorie distribuite secondo il coseno di
              deviazione dalla normale
            float rndPhi=2*M_PIf*(rndX);
            float rndTeta=acosf(sqrtf(rndY));

            //creazione della base ortonormale per la generazione dell'emisfero nel
               punto p interessato
            float3 u,v,w;
            //viene impostata la normale all'oggetto come primo vettore della base
               ortonormale
            w=n1;
            //vettore up
            float3 up(0.0015f,1.0f,0.021f);
            v=w%up;
            v.norm();
            u=v%w;

            //grazie alla creazione di una base ortonormale ci si può ricavare il
               punto nell'emisfero grazie alla parametrizzazione con gli angoli di
               eulero
            float3 dir=u*(cos(rndPhi)*sinf(rndTeta))+
            v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
            dir.norm();

        //viene creato il raggio da r.o ad un punto a caso sull'emisfero
        Ray reflRay(r.o,dir,RADIANCE);
        //si inizializza la massima distanza che il raggio può raggiungere
        float t=inf;
        //viene creato il puntatore all'oggetto che sarà intersecato dal raggio
        Obj** objX=new Obj*();
        *objX=NULL;

        //si interseca il raggio e si verifica che l'oggetto intersecato non sia
            una luce
        if(intersect(reflRay,t,objX)&&(material[(*objX)->matId].Le.max()==0)){

            //si calcola il punto di intersezione del raggio riflesso
            float3 iP=reflRay.o+reflRay.d*t;
            //viene creato il nuovo raggio per il calcolo della radianza
            Ray r2(iP,dir*(-1),RADIANCE);

        //si calcola la BRDF di cook e torrance tramite il raggio di entrata e il
            raggio di uscita e la normale all'oggetto
```

```
        float3 BRDF=material[mId].C_T_BRDF(reflRay,r,n1);

        //ricorsività della funzione di illuminazione indiretta:
        //per ottenere il valore esatto dell'integrale si divide per la probabilità
            di assorbimento alpha
        radianceOutput=radianceOutput+radiance(r2,objX,x,y,n)
        .mult(BRDF)*M_PIf*(1/alpha);
      }
    }
 }
    //una volta che la riflessione è stata eseguita si diminuisce il numero di
        riflessioni totali
    n--;

    //il risultato finale viene diviso per il numero totale di sample usati (ovvero
        aosamps)
  return radianceOutput/((float)aosamps);
}
```

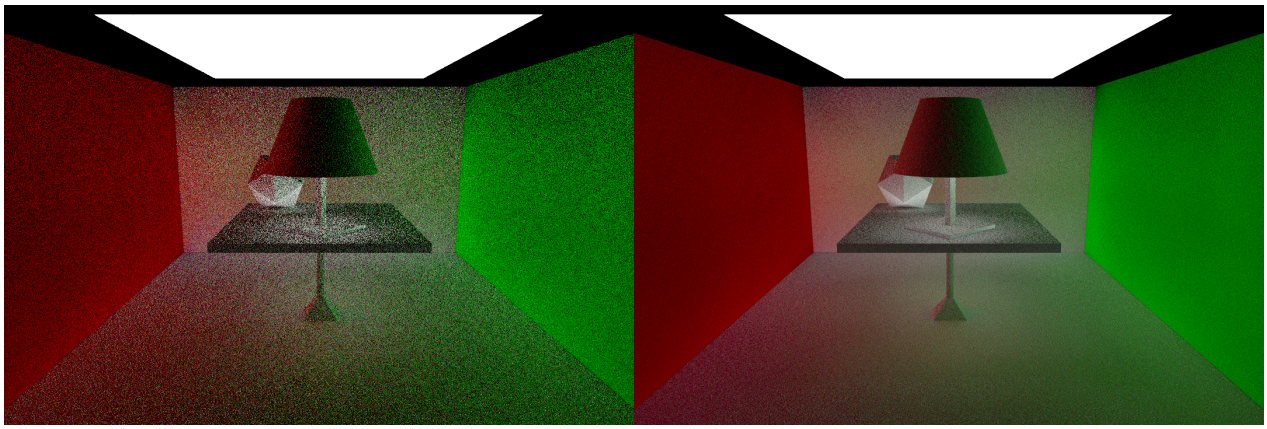


FIGURA 8.5.3. calcolo della sola illuminazione indiretta di una scena con 10 (figura a sinistra) e 100 sample (figura a destra) nell'estimatore di Monte Carlo.

Notiamo come in questo caso la PDF, al denominatore dell'estimatore di Monte Carlo, annulli il coseno presente nell'integrando. Quest'ultimo sarebbe ancora presente se avessimo distribuito i campioni uniformemente sull'emisfero. In questo caso, i campioni più distanti dalla normale avrebbero la stessa probabilità di essere estratti di quelli più vicini, ma quelli più lontani darebbero un contributo minore all'integrale, poiché sarebbero moltiplicati per il fattore coseno. Il concetto di dare più importanza ai campioni che danno un contributo maggiore nell'estimatore è definito *importance sampling*. Infatti, abbiamo visto in (8.3.7) che la PDF che minimizza (anzi addiritttura annulla) la varianza, nel calcolo dell'integrale di una funzione f tramite estimatore di Monte Carlo, è esattamente

$$p(x) = \frac{|f(x)|}{\int_{\mathcal{Q}} f(x)\,dx}\,,$$

ma non è mai possibile azzerare la varianza senza conoscere già il valore dell'integrale che vogliamo calcolare. Da qui però sappiamo che la PDF da usare deve seguire il più possibile il profilo della funzione da integrare.

8.5.4. Rejection Sampling . Quando non è sempre possibile derivare una formula analitica per l'Inverse Cumulative Distribution Function (Sezione8.4), si usa il Rejection sampling (Sottosezione 8.4.1), che ci permette di ottenere dei campioni, distribuiti in qualsiasi modo, semplicemente conoscendo la giusta densità di probabilità. Il metodo aumenta di 1 la dimensione della funzione da integrare, così da poter campionare la stessa PDF. Abbiamo visto che i campioni possono essere accettati o rifiutati, ma quelli accettati hanno la distribuzione scelta. Rammentiamo che esso consiste della seguente procedura. Considerando una densità di probabilità di dimensione 1 con valore massimo in un intervallo $[a, b]$ pari ad M, definiamo una nuova funzione a due dimensioni $[a, b] \times [0, M]$; essa viene campionata uniformemente attraverso coppie di valori (x, y). A questo punto vengono scartati tutti i valori x tali che $p(x) < y$, tutti gli altri sono accettati e sono quindi utilizzati per la stima dell'integrale. Ricordiamo che, se la proporzione dell'area sotto la PDF rispetto all'area del rettangolo $[a, b] \times [0, M]$ è piccola, questa tecnica risulta inefficiente poiché molti campioni devono essere scartati (si veda la Figura 8.4.2).

Seguendo un suggerimento in [**24**], abbiamo utilizzato questo metodo per il campionamento uniforme di una superficie sferica:

```
<<geometry.h>>
<<Struct Obj>>
<<metodo randomPoint>>
```

```
float x=2*s->rad*rnd1-s->rad;
float y=2*s->rad*rnd2-s->rad;
float z=2*s->rad*rnd3-s->rad;
//rejection sampling
while(x*x+y*y+z*z!=s->rad*s->rad){
rnd1=generateRandom(aoSId[0],5,aST);
rnd2=generateRandom(aoSId[0],6,aST);
rnd3=generateRandom(aoSId[0],7,aST);
x=2*s->rad*rnd1-s->rad;
y=2*s->rad*rnd2-s->rad;
z=2*s->rad*rnd3-s->rad;
}
return s->p+float3(x,y,z);
```

CAPITOLO 9

Algoritmi stocastici di Path Tracing

In questo capitolo viene illustrata una classe di algoritmi per il calcolo dell'illuminazione globale conosciuti con il nome di Path Tracing. La caratteristica peculiare di tali algoritmi è generare dei tracciati di trasporto della luce fra le sorgenti di luce, i punti sui quali vogliamo calcolare il valore della radianza ed il punto di osservazione della scena stessa. Un'altra importante caratteristica di tali algoritmi è che il valore della radianza viene calcolato direttamente per ogni singolo pixel, per questa ragione vengono definiti *basati sui pixel*.
All'interno del capitolo verranno presentati: una breve storia dell'algoritmo Path Tracing nel contesto degli algoritmi dell'illuminazione globale, il setup della camera e vari metodi per il calcolo dell'illuminazione diretta e indiretta. Infine, viene presentato l'algoritmo Light-Tracing evidenziando la dualità con il Ray Tracing.

9.1. Breve storia sulla nascita degli algoritmi Path Tracing

Gli algoritmi Path Tracing, come metodo risolutivo del calcolo dell'illuminazione globale, furono introdotti negli articoli sul Ray Tracing di Whitted [**52**]. Questi articoli descrivevano un nuovo modo per estendere gli algoritmi a tracciamento di raggi per determinare la visibilità delle superfici in una scena e per includere riflessioni e rifrazioni perfettamente speculari. Al tempo il Ray Tracing era un algoritmo veramente molto lento, con tempo di calcolo direttamente dipendente dal numero di raggi che venivano tracciati attraverso la scena. Il Path Tracing rappresenta una delle tante tecniche sviluppate per migliorare la velocità del calcolo di queste scene. Nel 1984, Cook descrisse il Ray Tracing stocastico [**11**]. I raggi venivano tracciati su percorsi multidimensionali cosí da poter rendere realistici fenomeni come: riflessioni e rifrazioni, sfumatura e la profondità di campo. Kajiya applicò la tecnica del Ray Tracing [**25**] per risolvere l'equazione del rendering (descritta nel Capitolo 7) che, per essere calcolata, segue esattamente i principi dell'illuminazione globale, includendo tutte le possibili riflessioni tra ogni tipo di superficie all'interno della scena. Altre tecniche di campionamento, che usano il metodo Monte Carlo, furono applicate all'equazione del rendering. Il metodo bidirezionale per il calcolo del Ray Tracing più completo ed efficiente fu introdotto in seguito da Lafortune e Veach [**48**].

9.2. Fase iniziale del procedimento di Ray Tracing

In un'immagine calcolata con il metodo dell'illuminazione globale si deve attribuire a ciascun pixel il valore della radianza L_{pixel}. Questo valore rappresenta una stima pesata della radianza incidente sul piano dell'immagine lungo il raggio proveniente dalla scena, passante attraverso il pixel p e diretto verso l'osservatore, come mostrato in Figura 9.2.1.

Questo valore è descritto più accuratamente dall'integrale pesato sul piano dell'immagine, calcolato nel modo seguente:

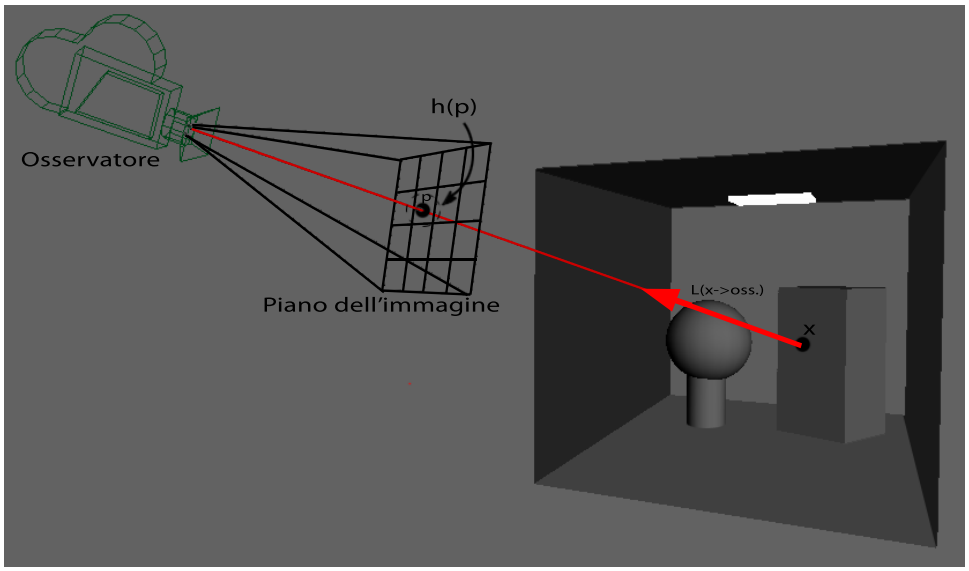


FIGURA 9.2.1. Costruzione geometrica del piano dell'immagine

$$\begin{aligned} L_{pixel} &= \int_{\text{piano}} L(p \to \text{oss})\, h(p)\, dp \\ &= \int_{\text{piano}} L(\boldsymbol{x} \to \text{oss})\, h(p)\, dp\,, \end{aligned}$$

dove p rappresenta un punto dell'immagine ed $h(p)$ il peso della funzione filtro, mentre $\boldsymbol{x}$ è il punto visibile dall'osservatore attraverso p. Spesso $h(p)$ rappresenta un semplice filtro che media il valore della radianza incidente sull'area del pixel. Il settaggio del Ray Tracing si riferisce quindi alla specifica configurazione della scena, del posizionamento della camera e del piano sul quale viene costruita l'immagine. Dobbiamo quindi conoscere il posizionamento esatto della camera, il suo orientamento e la risoluzione del piano dell'immagine. Presupponiamo che l'immagine sia centrata sull'asse di visuale, per stimare $L(p \to \Delta)$, viene tracciato un raggio dall'osservatore Δ attraverso il punto p, sulla direttrice del punto $\boldsymbol{x}$. Fintanto che rispettiamo la formula $L(p \to oss.) = L(\boldsymbol{x} \to \overrightarrow{xp})$, possiamo calcolare il valore della radianza tramite l'equazione del rendering. Un algoritmo di rendering, per disegnare l'intera scena, deve tracciare un raggio attraverso ogni pixel, tramite il quale può stimare la radianza per quel punto. Per ottenere il rendering finale l'algoritmo deve tracciare un raggio primario lungo il quale calcolare la radianza emessa dal punto $\boldsymbol{x}$ direttamente verso l'osservatore per ogni punto p del piano dell'immagine.

9.3. Ray Tracing stocastico semplice

9.3.1. Contributo esatto dei tracciati. Nel metodo a partire dai pixel sopra descritto la radianza viene calcolata esattamente utilizzando l'equazione del rendering. L'algoritmo piú semplice per calcolare tale valore è basato sul metodo di integrazione Monte Carlo nella sua forma standard, applicato all'equazione del rendering. Supponiamo di voler stimare la radianza dal punto $\boldsymbol{x}$ lungo il versore $\boldsymbol{\theta}$, ossia $L(\boldsymbol{x} \to \boldsymbol{\theta})$:

$$L(\boldsymbol{x} \to \boldsymbol{\theta}) = L_e(\boldsymbol{x} \to \boldsymbol{\theta}) + L_r(\boldsymbol{x} \to \boldsymbol{\theta})\,,$$

dove L_r è la *radianza riflessa*,

$$L_r(\boldsymbol{x} \to \boldsymbol{\theta}) = \int_{\Omega_{\boldsymbol{x}}} L(\boldsymbol{x} \leftarrow \boldsymbol{\psi})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \; d\omega(\boldsymbol{\psi})\,. \tag{9.3.1}$$

L'integrale può essere stimato utilizzando il metodo di Monte Carlo, generando N direzioni casuali $\boldsymbol{\psi}_i$ sull'emisfero Ω_i , distribuite secondo la funzione di probabilità $p(\boldsymbol{\psi})$.
Il valore stimato della radianza $L(\boldsymbol{x} \to \boldsymbol{\theta})$ è dato da:

$$\langle L_r(\boldsymbol{x} \to \boldsymbol{\theta}) \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{L(\boldsymbol{x} \leftarrow \boldsymbol{\psi}_i)\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi}_i)\, \langle \boldsymbol{\psi}_i, \boldsymbol{n}_{\boldsymbol{x}} \rangle}{p(\boldsymbol{\psi}_i)}\,.$$

Il prodotto scalare $\langle \boldsymbol{\psi}_i, \boldsymbol{n}_{\boldsymbol{x}} \rangle$ e la BRDF nell'integrando possono essere calcolati direttamente dai dati geometrici della scena. Il termine $L(\boldsymbol{x} \leftarrow \boldsymbol{\psi}_i)$, che rappresenta la radianza incidente nel punto $\boldsymbol{x}$ è ancora sconosciuto. Considerando il termine

$$L(\boldsymbol{x} \leftarrow \boldsymbol{\psi}_i) = L(r(\boldsymbol{x}, \boldsymbol{\psi}_i) \to -\boldsymbol{\psi}_i)\,,$$

dobbiamo tracciare il raggio uscente dal punto $\boldsymbol{x}$ in direzione $\boldsymbol{\psi}_i$ attraverso la geometria della scena per trovare eventuali punti di intersezione $r(\boldsymbol{x}, \boldsymbol{\psi}_i)$ con altri elementi della scena stessa. In questo modo abbiamo ottenuto una stima ricorsiva della radianza $L(\boldsymbol{x} \leftarrow \Psi_i)$, ed un raggio, o un albero di raggi, viene tracciato attraverso la scena. Ognuno dei raggi sparati in questo modo fornisce una stima della radianza diversa da 0 solo se la superficie colpita ha una L_e diversa da 0. In altre parole, il raggio, durante il suo cammino, per ottenere una stima >0 deve colpire una superficie luminosa o una sorgente di luce che di solito ricoprono un'area molto piccola della scena. Il risultato finale è quindi un'immagine quasi completamente nera. Solamente quando un raggio colpisce una sorgente di luce, al corrispondente pixel viene associato un colore. In teoria questo tipo di algoritmi può essere reso più efficiente scegliendo in modo opportuno il termine $p(\boldsymbol{\psi})$, in maniera proporzionale al termine coseno o alla BRDF. In pratica, a causa del grandissimo numero di campioni (ossia percorsi tracciati) con contributo zero, questo miglioramento non aumenta significativamente l'efficienza. In ogni modo, se aumentiamo enormemente il numero di raggi sparati, questo algoritmo elementare è in grado di produrre una scena correttamente illuminata.

9.3.2. Il metodo della Roulette Russa. Il generatore di percorsi stocastici descritto precedentemente riproduce il suo funzionamento in maniera ricorsiva fino ad un punto di arresto. Arrestare le iterazioni dopo un determinato numero di rimbalzi significa però introdurre un *errore sistematico* nell'algoritmo. Fisicamente la luce si riflette infinite volte all'interno di una scena ed alcuni rimbalzi potrebbero essere molto importanti al fine di una corretta visualizzazione. Il calcolo del nostro algoritmo, però, deve avere un numero di ricorsioni finito senza apportare un errore sistematico al calcolo della scena. Il metodo della Roulette Russa riduce sufficientemente il numero di iterazioni per rendere la scena computabile e mantiene casuale la lunghezza dei tracciati percorsi in modo da non introdurre un errore sistematico. Per illustrare tale metodo ci avvaliamo del seguente esempio. Supponiamo di voler calcolare l'integrale:

$$I = \int_0^1 f(x)\, dx\,.$$

Il metodo d'integrazione *Monte Carlo* calcola la media dei valori $f(x_i)$ nei punti generati a caso x_i appartenenti al dominio $[0, 1]$. Se il calcolo dell'integrale risultasse particolarmente difficile verrebbero generati numerosi punti x_i, per ridurre il numero di stime di $f(x_i)$ diventa quindi necessario stimare I scalando la funzione $f(x)$ orizzontalmente tramite un fattore P e verticalmente tramite un fattore $1/P$.

$$I_{RR} = \int_0^P \frac{1}{P} f\left(\frac{x}{P}\right) dx,$$

con $P \leqslant 1$. È chiaro che $I_{RR} = I$. Questo principio è illustrato nel grafico della Figura 9.3.1.

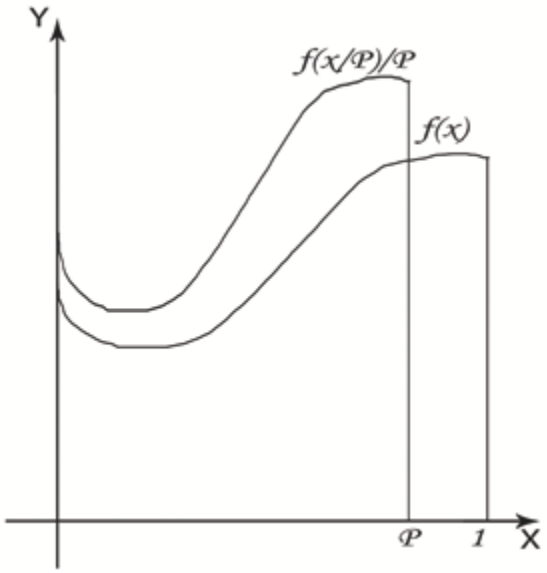


FIGURA 9.3.1. Principio della Roulette Russa

L'applicazione del medodo Monte Carlo produce il seguente estimatore dell'integrale:

$$\langle I_{RR} \rangle = \begin{cases} \frac{1}{P} f(\frac{x}{P}) & se\ x \leqslant P \\ 0 & se\ x > P\,. \end{cases}$$

Se $f(x)$ è un altro integrale ricorsivo, come nel calcolo dell'equazione del rendering, il risultato di questo metodo è che la recursione viene interrotta con una probabilità $\alpha = 1 - P$ per ogni punto stimato. Di conseguenza α è chiamata *probabilità di assorbimento.*

Ovviamente i campioni che cadranno nell'intervallo $[P, 1]$ avranno valore 0, ma questo effetto viene compensato pesando la funzione nell'intervallo $[0, P]$ con il fattore $1/P$. In questo modo non aggiungiamo all'estimatore alcun errore sistematico. Se α è piccolo l'estimatore risulta accurato perché avverranno un gran numero di recursioni, se α è grande le recursioni si fermeranno prima e quindi avremo una varianza maggiore. In ogni caso non abbiamo introdotto alcun errore sistematico nel calcolo dell'estimatore. La varianza dell'estimatore è di conseguenza:

$$\sigma^2 = \frac{1}{P^2} \int_0^P \frac{f^2}{P}\left(\frac{x}{P}\right) dx - I^2 = \frac{1}{P^2} \int_0^1 f^2\, dx - \left(\int_0^1 f\, d\boldsymbol{x}\right)^2.$$

Da questa espressione segue immediatamente che la varianza aumenta al decrescere di P, come del resto è intuitivo visto che, per valori di P via via minori, una percentuale via via più elevata di campioni non contribuisce al calcolo dell'integrale.
La Figura 9.3.2 mostra il funzionamento dell'algoritmo Path-Tracyng per il calcolo della radianza

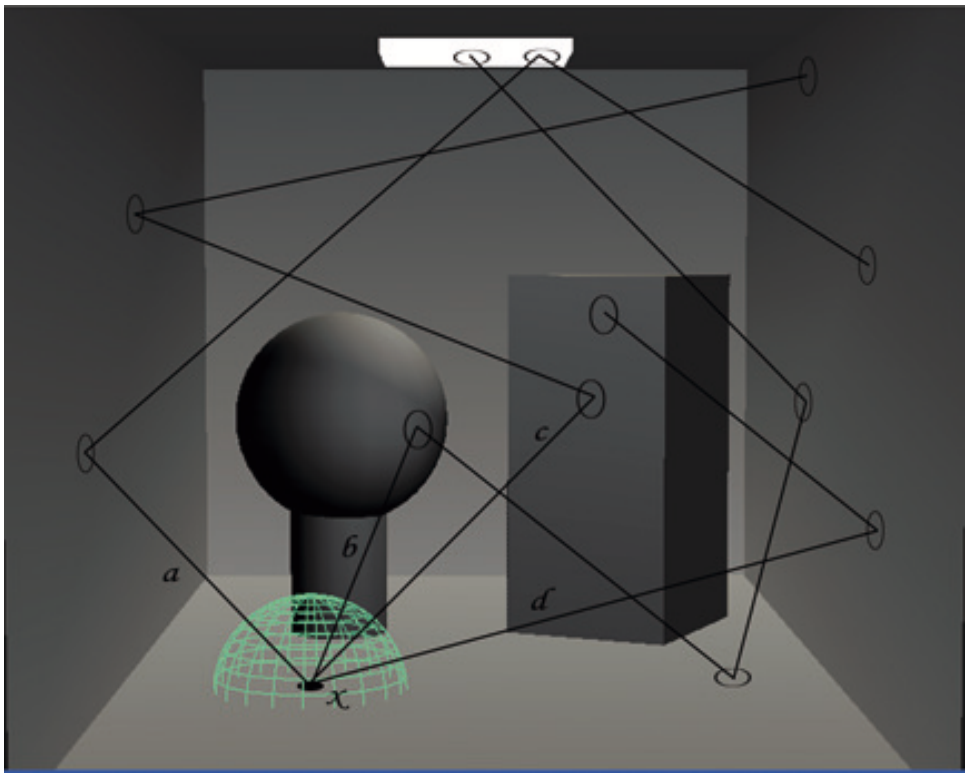


FIGURA 9.3.2. Schema di tracciati generati dal Ray Tracing stocastico semplice

nel punto $\boldsymbol{x}$: da $\boldsymbol{x}$ partono 4 percorsi di lunghezza stocastica, il percorso a ed il percorso b daranno un contributo positivo perché durante, o alla fine del loro cammino, incontrano la sorgente luminosa, i percorsi c e d, invece, non daranno nessun contributo alla radianza del punto $\boldsymbol{x}$.

9.4. Illuminazione diretta

L'algoritmo di Path Tracing precedentemente illustrato è molto inefficiente, in quanto non tiene assolutamente conto del posizionamento delle sorgenti di luce che di solito coprono una piccola parte della scena: quindi la maggior parte dei tracciati stocastici restituisce un contributo nullo. Questo spreca le risorse di calcolo ed aumenta la varianza del risultato.

9.4.1. Illuminazione diretta ed illuminazione indiretta. Come illustrato nel Capitolo 7, per il calcolo della radianza emessa da un punto $\boldsymbol{x}$ in una direzione $\boldsymbol{\theta}$, l' equazione del rendering suddivide in maniera esatta le componenti della luce che incidono nel punto $\boldsymbol{x}$ in luce diretta e luce indiretta. Prendiamo in considerazione la radianza riflessa:

$$\begin{aligned} L_r(\boldsymbol{x} \to \boldsymbol{\theta}) &= \int_{\Omega_{\boldsymbol{x}}} L(\boldsymbol{x} \leftarrow \boldsymbol{\psi})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \; d\omega(\boldsymbol{\psi}) \\ &= \int_{\Omega_{\boldsymbol{x}}} L(r(\boldsymbol{x}, \boldsymbol{\psi}) \to -\boldsymbol{\psi})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \; d\omega(\boldsymbol{\psi})\,. \end{aligned}$$

Riscrivendo $L(r(\boldsymbol{x}, \boldsymbol{\psi}) \to -\boldsymbol{\psi})$ come la somma della radianza autoe- messa e della radianza riflessa $r(\boldsymbol{x}, \boldsymbol{\psi})$ otteniamo:

$$\begin{aligned} L_r(\boldsymbol{x} \to \boldsymbol{\theta}) = & \int_{\Omega_{\boldsymbol{x}}} L_e(r(\boldsymbol{x}, \boldsymbol{\psi}) \to -\boldsymbol{\psi})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \; d\omega(\boldsymbol{\psi}) \\ & + \int_{\Omega_{\boldsymbol{x}}} L_r(r(\boldsymbol{x}, \boldsymbol{\psi}) \to -\boldsymbol{\psi})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \; d\omega(\boldsymbol{\psi}) \\ & = L_{diretta}(\boldsymbol{x} \to \boldsymbol{\theta}) + L_{indiretta}(\boldsymbol{x} \to \boldsymbol{\theta})\,. \end{aligned}$$

Il termine di illuminazione diretta $L_{diretta}(\boldsymbol{x} \to \boldsymbol{\theta})$ esprime il contributo incidente sulla superficie proveniente direttamente dalla sorgente di luce, quindi possiamo trasformare l'integrale emisferico del punto $\boldsymbol{x}$ in un integrale sull'area della sorgente luminosa:

$$L_{dir}(\boldsymbol{x} \to \boldsymbol{\theta}) = \int_{A_s} L_e(\boldsymbol{y} \to \overrightarrow{xy})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{xy})\, G(\boldsymbol{x}, \boldsymbol{y})\, V(\boldsymbol{x}, \boldsymbol{y})\, dA_{\boldsymbol{y}}\,,$$

dove $\boldsymbol{y} = r(\boldsymbol{x}, \boldsymbol{\psi}$.
Nel caso in cui sono presenti nella scena diverse sorgenti luminose basta sommare i loro integrali:

$$L_{dir}(\boldsymbol{x} \to \boldsymbol{\theta}) = \sum_{k=1}^{N_L} \int_{A_s} L_e(r \to \overrightarrow{xy})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{xy})\, G(\boldsymbol{x}, \boldsymbol{y})\, V(\boldsymbol{x}, \boldsymbol{y})\, dA_{\boldsymbol{y}}\,. \tag{9.4.1}$$

Nelle precedenti formule il termine G rappresenta il fattore geometrico che contiene il termine cos mentre il termine V rappresenta il fattore di visibilità.

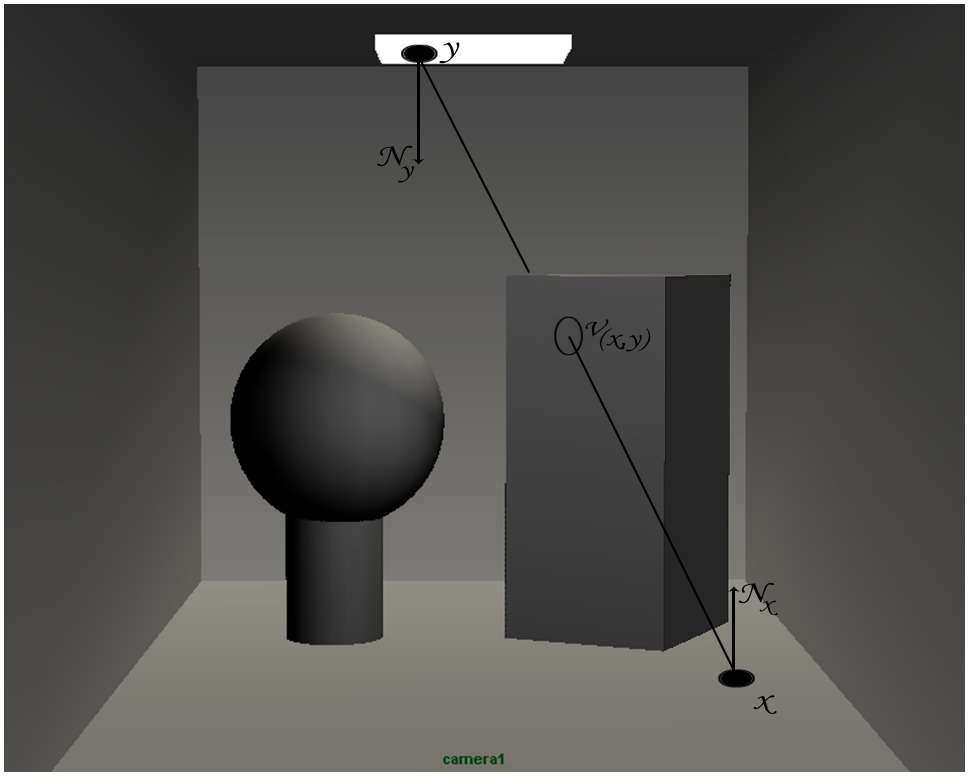


FIGURA 9.4.1. Integrazione per area sulle sorgenti luminose per l'illuminazione diretta

Nella Figura 9.4.1 è rappresentato un semplice schema di interazione del modello trattato. In questo modo il numero di percorsi che interessano ciascun punto viene ridotto enormemente e quindi possiamo sparare un numero di campioni maggiore per unità di area, rendendo molto più accurato il calcolo dell'estimatore L_r rispetto al Path-Tracyng classico. I punti $\boldsymbol{x}$ che vedono interamente le sorgenti luminose sono semplici da calcolare e quindi non richiedono un gran numero di campioni, questo metodo, però, permette anche la stima accurata delle aree di penombra. Per queste zone la stima diventa più accurata e quindi, per ottenere una buona resa, dovremmo concentrare qui un maggior numero di tracciati per diminuire la varianza dell'estimatore. Si veda la Figura 9.4.3.

9.4.2. Illuminazione proveniente da una singola sorgente. Per calcolare l'illuminazione diretta del punto $\boldsymbol{x}$ proveniente da una singola superficie luminosa dobbiamo definire una densità di probabilità $p(\boldsymbol{y})$ relativa all'emettitore. Applicando il metodo di Monte Carlo all'equazione (9.4.1) otteniamo il seguente estimatore:

$$\langle L_{dir}(\boldsymbol{x} \to \boldsymbol{\theta})\rangle = \frac{1}{N_s}\sum_{i=1}^{Ns} \frac{L_e(\boldsymbol{y}_i \to \overrightarrow{y_i x})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{x}\boldsymbol{y}_i})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i)}{p(\boldsymbol{y}_i)}\,.$$

Per ogni punto $\boldsymbol{y}_i$ della sorgente dobbiamo calcolare il valore dell'energia irradiata al punto $\boldsymbol{x}$. Per fare questo dobbiamo tenere in considerazione la BRDF del punto $\boldsymbol{x}$, il fattore di visibilità V, il fattore geometrico G e la radianza emessa nel punto $\boldsymbol{y}_i$. La varianza dell'estimatore dipende quindi direttamente dalla densità di campioni definita da $p(\boldsymbol{y})$.

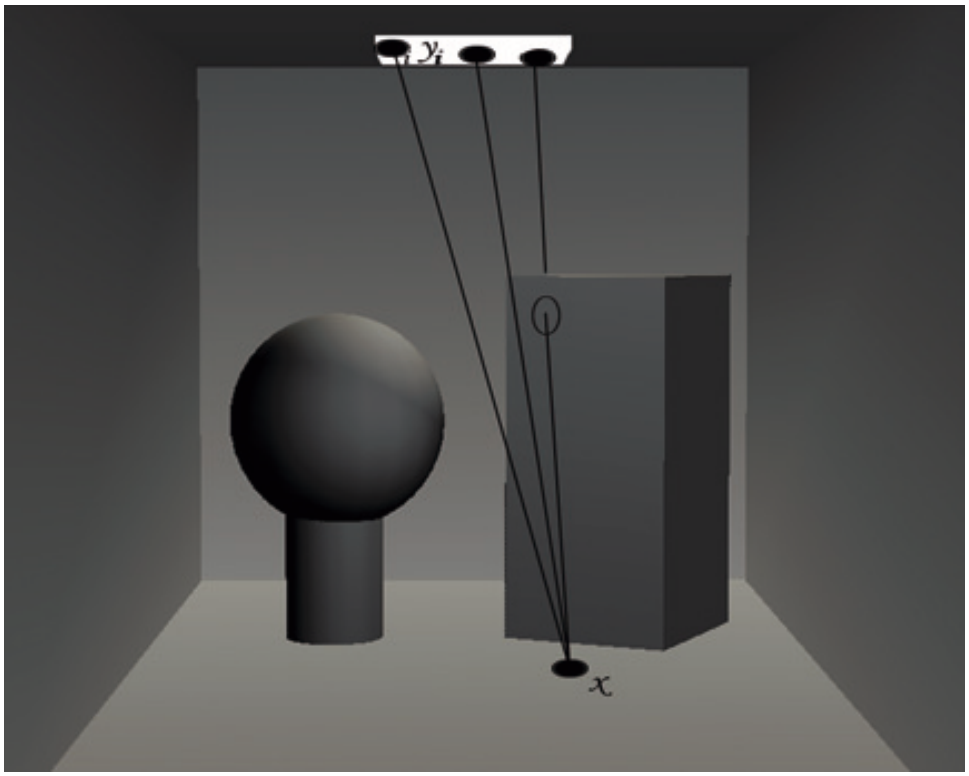


FIGURA 9.4.2. Campionamento uniforme della sorgente di luce per l'illuminazione diretta

Teoricamente, per essere accurati, dovremmo campionare tutti i punti della sorgente di luce, ma questo renderebbe il calcolo troppo oneroso. Si può procedere ad esempio campionando in maniera

uniforme la superficie luminosa (Figura 9.4.2). In questo caso i punti $\boldsymbol{y}_i$ sono equamente distribuiti sulla superficie luminosa e la loro densità di probabilità è $p(\boldsymbol{y}) = \frac{1}{A_{\text{sorgente}}}$. Questa soluzione è la piu' veloce da calcolare, ma è anche quella che genera la varianza più elevata. Come si può vedere nella Figura 9.4.3 è calcolata tracciando per ogni punto $\boldsymbol{x}$ rispettivamente 1, 2, 10 e 40 raggi, le differenze si possono apprezzare soprattutto nelle aree di penombra. In queste zone il fattore V di occlusione $V(\boldsymbol{x}, \boldsymbol{y}_i) = 0$ determina se il raggio colpisca o meno la sorgente, di conseguenza, se il numero di raggi è basso la varianza è elevata.

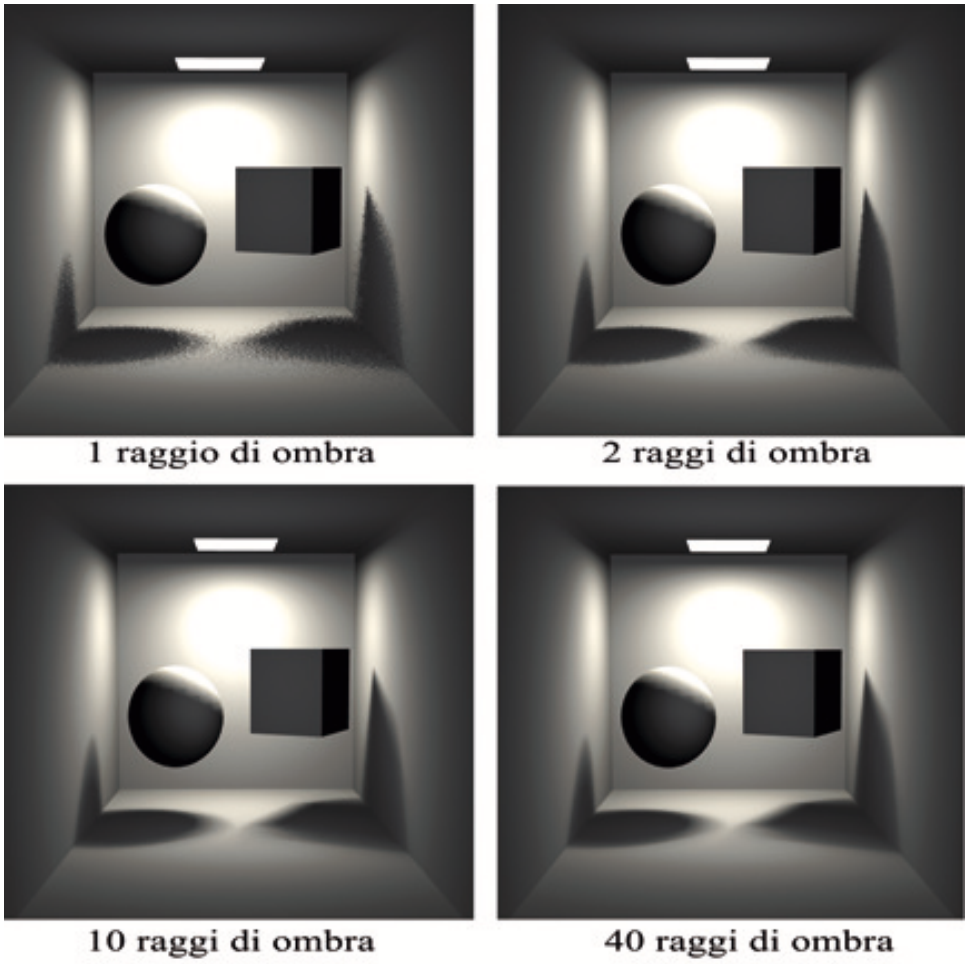


FIGURA 9.4.3. Campionamento uniforme della sorgente di luce. Le immagini sono generate utilizzando rispettivamente 1,2,10,40 raggi di ombra per ogni pixel.

9.4.3. Illuminazione proveniente da più sorgenti. In una scena reale di solito abbiamo più sorgenti luminose nello stesso ambiente. Per il calcolo dell'illuminazione globale possiamo utilizzare semplicemente il metodo appena descritto per il calcolo dell'illuminazione diretta da ogni singola sorgente. Ad ogni punto $\boldsymbol{x}$ si calcola allora il contributo di ogni singola sorgente ed alla fine si sommano i risultati. In questo modo si usano estimatori separati per ciascuna sorgente, ed occorre scegliere con che pesi sommare gli estimatori, e quanti campioni selezionare su ciascuna sorgente, al fine di minimizzare la varianza. La scelta del numero di campioni può essere fatta tramite una

distribuzione secondo l'importanza delle sorgenti, basata sulla diversa potenza delle varie sorgenti e sull'inverso del quadrato della distanza dal punto $\boldsymbol{x}$.

Un altro possibile approccio è quello di considerare le varie superficie luminose come un'unica sorgente composita ed utilizzare il metodo di Monte Carlo scegliendo campioni su questa sorgente complessiva. Sappiamo che l'estimatore di Monte Carlo sarebbe unbiased anche con un unico campione (ma con pochi campioni la varianza sarebbe elevata!). Occorre scegliere un numero adeguato di campioni (ossia raggi d'ombra) sulla sorgente complessiva, ma questa è l'unione di varie superficie che emettono luce e quindi occorre scegliere campioni, in numero appropriato, su ciascuna di esse. Di solito quindi l' algoritmo sceglie i raggio di ombra in base a probabilit` condizionate, ovvero a due passi.

- *Primo passo.* Viene assegnata una funzione di probabilità discreta $p_L(k)$ per la scelta di sorgente luminosa k. Tipicamente questa funzione di probabilità è la stessa per tutte le sorgenti di luce, ma può essere diversa per i vari punti $\boldsymbol{x}$ sui quali vogliamo calcolare l'ombreggiatura.
- *Secondo passo.* In questa fase ogni punto $\boldsymbol{y}_i$ appartenente ad una superficie k viene scelto in base ad una PDF $p(\boldsymbol{y}/k_i)$ condizionale. La natura di questa PDF è quella di una probabilità condizionata in base alla scelta della sorgente. Questo processo è fondamentale per concentrare l'attenzione dei campionamenti sulle sorgenti di luce che illuminano direttamente le zone coinvolte nel rendering ed escludono quelle occluse da un altro oggetto della geometria.

La PDF combinata per ogni punto $\boldsymbol{y}_i$ della superficie che emette la radianza è $p_L(k)\,p(\boldsymbol{y}|k)$. Per N raggi che concorrono al calcolo dell'estimatore avremo quindi:

$$\langle L_{dir}(\boldsymbol{x} \to \boldsymbol{\theta})\rangle = \frac{1}{N}\sum_{i=1}^{N} \frac{L_e(\boldsymbol{y}_i \to \overrightarrow{\boldsymbol{y}_i\boldsymbol{x}})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{x}\boldsymbol{y}_i})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i)}{p_L(k_i)\, p(\boldsymbol{y}_i|k_i)}\,.$$

In ogni caso la scelta della PDF, ovvero la scelta di $p_L(k)$ e da $p(\boldsymbol{y}|k)$, produce una stima non viziata, perché gli estimatori di Monte Carlo sono non viziati. La scelta della distribuzione di probabilità nell'estimatore influisce fortemente sulla varianza, e quindi sul rumore nell'immagine calcolata. Alcune delle scelte più comuni sono:

9.4.3.1. *Campionamento uniforme delle sorgenti e della superficie luminosa.* In questo caso entrambe le PDF sono uniformi, la probabilità di scelta della sorgente è $p_L(k) = \frac{1}{N_L}$ e la probabilità di scelta del punto sulla superficie luminosa è $p(\boldsymbol{y}|k) = \frac{1}{A_{L_k}}$. In questo modo ogni sorgente di luce riceve un egual numero di raggi tramite i quali calcolare la radianza equamente distribuiti sull'area dell'emettitore. Questo metodo è di semplice e rapida implementazione, ma le sorgenti luminose non vengono pesate in base alla loro importanza e ricevono un egual numero di raggi, anche se ci troviamo in area di penombra e la geometria della scena ostruisce alcune sorgenti rispetto al punto esaminato. Sostituendo le PDF nell'equazione 9.4.3 otteniamo il seguente estimatore della radianza emessa in direzione $\boldsymbol{\theta}$:

$$\langle L_{dir}(\boldsymbol{x} \to \boldsymbol{\theta})\rangle$$

$$= \frac{N_L}{N}\sum_{i=1}^{N} A_{L_{k_i}} L_e(\boldsymbol{y}_i \to \overrightarrow{\boldsymbol{y}_i\boldsymbol{x}})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{x}\boldsymbol{y}_i})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i)\,. \quad (9.4.2)$$

9.4.3.2. *Campionamento proporzionale alla potenza di emissione della sorgente con uniforme campionamento della superficie luminosa.* In questo metodo il la distribuzione di probabilità discreta

della scelta delle sorgenti è $p_L(k) = P_k/P_{totale}$ dove P_k è la potenza della sorgente k e P_{totale} è la somma della potenza di tutte le sorgenti della scena. Questo metodo assegna quindi un numero di raggi maggiore alle sorgenti di grande intensità riducendo notevolmente la varianza rispetto al metodo precedente. L'estimatore diventa quindi:

$$\langle L_{dir}(\boldsymbol{x} \to \boldsymbol{\theta})\rangle = \frac{P_{tot}}{N} \sum_{i=1}^{N} \frac{A_{L_{k_i}} L_e(\boldsymbol{y}_i \to \overrightarrow{\boldsymbol{y}_i\boldsymbol{x}})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{x}\boldsymbol{y}_i})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i)}{P_{k_i}}\,. \quad (9.4.3)$$

Nel caso in cui tutte le sorgenti luminose sono diffusive sappiamo da (7.4.1) che $P_k = \pi\, A_k\, L_{e,k}$, e quindi l'estimatore della radianza si semplifica nel modo seguente:

$$\langle L_{dir}(\boldsymbol{x} \to \boldsymbol{\theta})\rangle = \frac{P_{tot}}{\pi N} \sum_{i=1}^{N} f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{x}\boldsymbol{y}_i})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i)\,.$$

Questo metodo produce ottimi risultati eccetto il caso in cui stiamo analizzando una zona che viene ostruita dalla geometria della scena rispetto alle sorgenti più forti e che quindi riceve luce solo da sorgenti deboli. Questo problema può essere risolto solamente applicando un controllo preventivo dell'occlusione del punto rispetto alle sorgenti. Ma per attuare questa strategia dovremmo tracciare raggi a partire dal punto in cui vogliamo calcolare la radianza, però in tal caso non avremmo alcun vantaggio (molti raggi, ossia versori campionati, danno contributo nullo e quindi aumentano la varianza); ma possiamo invece utilizzare più semplicemente uno $z-$buffer per evitare il tracciamento dei raggi (analogamente a quanto fatto nella Sezione **??**).

Un altro problema di questo metodo consiste nel fatto che vengono generati tre indici casuali: un numero stocastico per la selezione delle sorgenti k e due numeri casuali per la selezione del punto $\boldsymbol{y}_i$ relativo alla superficie luminosa. Questo procedimento genera di conseguenza dati con parecchie dimensioni, e la stratificazine è più scomoda da implementare.

9.5. Illuminazione indiretta

Questa sezione affronta il calcolo della stima dell'illuminazione indiretta di una scena. L'illuminazione indiretta in un punto $\boldsymbol{x}$ qualsiasi della scena è un problema di calcolo molto più difficile da affrontare di quella diretta perché il contributo di questa componente, incidente su un punto $\boldsymbol{x}$ può essere lievemente più complesso (occorre stratificare come nel problema delle tre torri).
Il calcolo della componente indiretta interessa l'interazione luminosa che avviene tra il punto esaminato e l'intera scena al più di una interazione tra il punto e le restanti superfici. Questo richiede spesso la maggior parte del tempo di calcolo richiesto da un algoritmo di illuminazione globale, ma è essenziale per ottenere un risultato fotorealistico della scena.

9.5.1. Campionamento uniforme per l'illuminazione indiretta. Per stimare l'illuminazione totale $L(\boldsymbol{x} \to \boldsymbol{\theta})$ emessa da un punto $\boldsymbol{x}$, nella Sottosezione 9.4.1 abbiamo diviso il contributo incidente sul punto nelle componenti diretta ed indiretta. La componente indiretta è rappresentata da:

$$L_{ind}(\boldsymbol{x} \to \boldsymbol{\theta}) = \int_{\Omega_{\boldsymbol{x}}} L_r(r(\boldsymbol{x}, \boldsymbol{\psi}) \to -\boldsymbol{\psi})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle\, d\omega(\boldsymbol{\psi})\,.$$

L'integrando contiene il termine L_r che rappresenta la radianza riflessa da tutti gli altri punti della scena che sono composti a loro volta da una componente diretta ed una indiretta. Il valore

L_r in un ambiente chiuso produce un contributo non nullo per tutte le coppie di punti $(\boldsymbol{x}, \boldsymbol{\psi})$ che compongono la scena. Quindi dobbiamo integrare l'intero emisfero frontale al punto di interesse $\boldsymbol{x}$, ed il calcolo di tale integrale deve essere accurato.

Il metodo Monte Carlo applicato alla stima dell'estimatore per l'illuminazione indiretta, viene utilizzato per generare N direzioni casuali $\boldsymbol{\psi}_i$ con probabilità $p(\boldsymbol{\psi})$ sull'emisfero in oggetto. L'estimatore generato è:

$$\langle L_{ind}(\boldsymbol{x} \to \boldsymbol{\theta}) \rangle$$

$$= \frac{1}{N} \sum_{i=1}^{N} \frac{L_r(r(\boldsymbol{x}, \boldsymbol{\psi}_i) \to -\boldsymbol{\psi}_i)\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi}_i)\, \langle \boldsymbol{\psi}_i, \boldsymbol{n}_{\boldsymbol{x}} \rangle\, d\omega(\boldsymbol{\psi})}{p(\boldsymbol{\psi}_i)}\,. \quad (9.5.1)$$

Per la valutazione di questo estimatore dobbiamo calcolare la BRDF ed il prodotto scalare per ogni direzione generata $\boldsymbol{\psi}_i$, tracciare un raggio dal punto $\boldsymbol{x}$ verso la direzione $\boldsymbol{\psi}_i$ e stimare la radianza riflessa $L_r(r(\boldsymbol{x}, \boldsymbol{\psi}_i) \to -\boldsymbol{\psi}_i)$ nel punto di intersezione più vicino ad $\boldsymbol{x}$ della retta $\boldsymbol{\psi}_i$. Da questa considerazione si evince facilmente la natura ricorsiva dell'algoritmo per l'illuminazione indiretta. Ovviamente il numero di rimbalzi viene arrestato stocasticamente (con il metodo della Roulette Russa) invece che deterministicamente, e quindi l'estimatore ricorsivo rimane privo di bias. Tipicamente il valore della riflessività locale viene utilizzato per stimare un'appropriata probabilità di assorbimento. La Figura 9.5.1 mostra il calcolo per il punto $\boldsymbol{x}$.

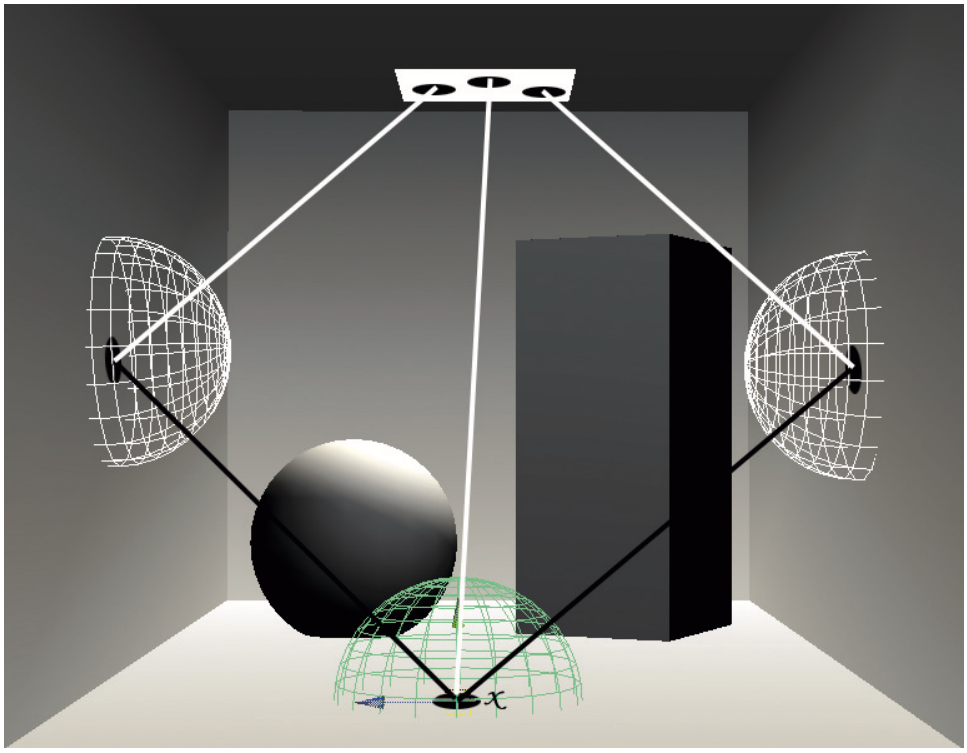


FIGURA 9.5.1. Tracciati generati durante il calcolo dell'illuminazione indiretta. I raggi d'ombra utilizzati per il calcolo dell'illuminazione diretta sono rappresentati dalle linee bianche.

9.5.2. Il campionamento per importanza nell'illuminazione indiretta. La scelta più semplice del fattore $p(\boldsymbol{\psi})$ è quella di prendere una PDF uniforme, con $p(\boldsymbol{\psi}) = \frac{1}{2\pi}$. In questo modo le direzioni vengono campionate in maniera uniforme rispetto all'emisfero che ricopre il punto esaminato. Questo metodo inoltre è di semplice e rapida implementazione, però puo' causare un disturbo piuttosto evidente nell'immagine finale perché la varianza della BRDF, della stima del coseno e della radianza riflessa non vengono ottimizzate.
Per ottenere una riduzione del rumore si possono seguire diversi approcci, dobbiamo costruire un PDF sull'emisfero proporzionale ad ognuno dei seguenti fattori:

- Il prodotto scalare $\langle \boldsymbol{\psi}_i, \boldsymbol{n}_{\boldsymbol{x}} \rangle$.
- La BRDF $f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi}_i)$.
- Il fattore della radianza incidente $L_r(r(\boldsymbol{x}, \boldsymbol{\psi}_i))$.
- Una combinazione di ognuno dei suddetti elementi.

Le direzioni di campionamento proporzionali al lobo del coseno attorno alla normale $\boldsymbol{n}_{\boldsymbol{x}}$ evitano di prendere molti punti di campionamento dell'emisfero vicini all'orizzonte, ovvero dove il prodotto scalare $\langle \boldsymbol{\psi}_i, \boldsymbol{n}_{\boldsymbol{x}} \rangle$ è 0. Ci aspetteremo una riduzione del rumore, perché in questo modo vengono eliminate le componenti che contribuiscono poco o per nulla al valore dell'estimatore, che diventa:

$$p(\boldsymbol{\psi}) = \langle \boldsymbol{\psi}_i, \boldsymbol{n}_{\boldsymbol{x}} \rangle / \pi \, .$$

Se assumiamo che la BRDF f_r è diffusivo nel punto $\boldsymbol{x}$, otterremo il seguente estimatore:

$$\langle L_{ind}(\boldsymbol{x} \to \boldsymbol{\theta}) \rangle = \frac{\pi f_r}{N} \sum_{i=1}^{N} L_r(r(\boldsymbol{x}, \boldsymbol{\psi}_i) \to -\boldsymbol{\psi}_i) \, .$$

In questo estimatore la componente che puó generare disturbo rimane soltanto la variazione della radianza incidente nell'emisfero esaminato.

9.5.3. Campionamento della BRDF. Quando le direzioni di campionamento $\boldsymbol{\psi}$ sono proporzionali al prodotto scalare, possiamo trascurare la duplice natura della BRDF nel punto $\boldsymbol{x}$, teoricamente le direzioni con un alta BRDF verranno campionate con maggiore probabilità. Questo è un ottimo modo per ridurre il rumore quando siamo in presenza di superfici riflettenti o con un alto valore della BRDF.
Per essere maggiormente accurati dovremmo stimare i campionamenti in base al prodotto tra la funzione BRDF e il prodotto scalare. L'esempio illustrato di seguito deriva dal modello di illuminazione di Phong e determina una stima proporzionale alla BRDF, come sopra citato:

$$f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi}) = k_d + k_s \langle \boldsymbol{\psi}_i, \boldsymbol{\theta}_s \rangle^n \, ,$$

dove il termine $\boldsymbol{\theta}_s$ rappresenta la direzione perfettamente speculare a $\boldsymbol{\theta}$ rispetto a $\boldsymbol{n}_{\boldsymbol{x}}$. Questa BRDF ha una parte diffusiva k_d ed una parte speculare $k_s \langle \boldsymbol{\psi}_i, \boldsymbol{\theta}_s \rangle^n$. L'integrale dell'illuminazione indiretta puo' essere ora diviso in due parti e scritto nella maniera seguente:

$$\begin{aligned} L_{ind}(\boldsymbol{x} \to \boldsymbol{\theta}) = & \int_{\Omega_{\boldsymbol{x}}} L_r(r(\boldsymbol{x}, \boldsymbol{\psi}) \to -\boldsymbol{\psi}) k_d \, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \, d\omega(\boldsymbol{\psi}) \\ & + \int_{\Omega_{\boldsymbol{x}}} L_r(r(\boldsymbol{x}, \boldsymbol{\psi}) \to -\boldsymbol{\psi}) k_s \, \langle \boldsymbol{\psi}_i, \boldsymbol{\theta}_s \rangle^n \, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \, d\omega(\boldsymbol{\psi}) \, . \end{aligned}$$

Le due parti consistono di due integrali, di Lambert e di Phong, calcolabili separatamente (ma ricorsivamente) con il metodo di Monte Carlo. A tale scopo è opportuno introdurre prima una

variabile aleatoria a tra stati, il primo per l'evento di applicare l'estimatore di Lambert, il secondo per l'estimatore di Phong, ed l terzo per l'assorbimento via Roulette Russa. La probabilità del terzo evento, ovviamente, si sceglie pari a $1-\rho_{\boldsymbol{x}}$, dove $\rho_{\boldsymbol{x}}$ è il coefficiente di riflettività al punto $\boldsymbol{x}$, il quale, se non fornito in fase di modellazione, possiamo approssimare con k_d+k_s; le probabilità degli altri due eventi si scelgono proporzionali ai rispettivi coefficienti k_d e k_s, e la costante di proporzionalità èscelta in modo che la somma delle tre probabilità valga 1. Una vlta determinato il tipo di estimatore o assorbimento grazie a questa variabile aleatoria preliminare, i campioni da estrarre per gli estimatori sono scelti con i consueti metodi spiegati sopra. In tal modo il campionamento dell'estimatore di $L_{ind}(\boldsymbol{x} \to \boldsymbol{\theta})$ è basato in maniera naturale sul principio della probabilità condizionata.

9.5.4. Campionamento di area. Campionare l'area dell'emisfero è il metodo più diretto per calcolare l'integrale dell'illuminazione indiretta. Per ogni direzione campionata viene inviato un raggio per determinare il punto più vicino d'intersezione con la scena. Questa operazione è computazionalmente costosa, ma saremo sicuri di non incorrere in disturbo provocato dall'interazione con oggetti invisibili dal punto $\boldsymbol{x}$.
Come per l'illuminazione diretta ci sono molti modi per descrivere l'integrale dell'illuminazione indiretta, nel caso in cui utilizziamo il metodo sopra descritto otteniamo:

$$L_{ind}(\boldsymbol{x} \to \boldsymbol{\theta}) = \int_{A_{scena}} L_r(\boldsymbol{y} \to \overrightarrow{\boldsymbol{yx}})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{xy}})\, G(\boldsymbol{x}, \boldsymbol{y})\, V(\boldsymbol{x}, \boldsymbol{y})\, dA_{\boldsymbol{y}}\,.$$

Il corrispondente estimatore utilizzando una PDF $p(\boldsymbol{y})$ è:

$$\langle L_{ind}(\boldsymbol{x} \to \boldsymbol{\theta})\rangle = \frac{1}{N}\sum_{i=1}^{N} \frac{L_r(\boldsymbol{y}_i \to \overrightarrow{\boldsymbol{y}_i\boldsymbol{x}})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \overrightarrow{\boldsymbol{xy}_i})\, G(\boldsymbol{x}, \boldsymbol{y}_i)\, V(\boldsymbol{x}, \boldsymbol{y}_i)}{p(\boldsymbol{y}_i)}\,.$$

9.5.5. Costruzione di un algoritmo completo. A questo punto della trattazione abbiamo tutti gli argomenti necessari per costruire un renderer di illuminazione globale completo utilizzando il tracciamento di raggi stocastico. L'efficienza e l'accuratezza dell'algoritmo generato dipenderanno dai seguenti fattori:

Numero di raggi per pixel. Il numero di raggi sparati attraverso un pixel del piano dell'immagine o, più generalmente, il supporto di $h(p)$ è determinante per la riduzione dei fenomeni di aliasing e di rumore presenti nell'immagine finale.

Illuminazione diretta. Per questo fattore sono necessarie una serie di scelte che determinano l'efficienza e l'accuratezza del metodo:

- Il numero totale dei raggi d'ombra N_d tracciati per ogni punto $\boldsymbol{x}$.
- L'importanza di una singola sorgente di luce rispetto alla somma di tutte le sorgenti della scena.
- La distribuzione dei raggi di ombra sull'area della singola sorgente di luce.

Illuminazione indiretta. Di solito questa tecnica si implementa campionando l'emisfero attorno al punto interessato, le componenti di cui dobbiamo tener conto sono:

- Il numero di raggi N_i di illuminazione indiretta distribuiti sull'emisfero $\Omega_{\boldsymbol{x}}$.
- La distribuzione dei raggi sull'emisfero (coseno, BRDF, ..)
- La probabilità di assorbimento dei raggi effettuata tramite il metodo della roulette russa.

Ovviamente aumentare il numero di raggi per ogni punto, il numero di rimbalzi ed il campionamento dell'emisfero contribuisce alla realizzazione di una immagine priva di disturbo e di altissima accuratezza. Tutto questo però va a discapito della velocità di esecuzione. Di volta in volta si deve trovare il giusto compromesso tra i settaggi del nostro renderer ed i tempi di calcolo.

9.6. Light Tracing

L'algoritmo di Ray Tracing stocastico, descritto nel paragrafo precedente, deriva dall'applicazione del metodo Monte Carlo alla stima dell'equazione del rendering. Il risultato è che l'algoritmo traccia raggi casuali dall'osservatore della scena attraverso il piano dell'immagine e dopo una serie di rimbalzi, i raggi che colpiscono una sorgente di luce, vengono utilizzati per stimare il valore associato al pixel in esame. Questo processo appare del tutto innaturale perchè la luce normalmente parte dalla sorgente emettititrice e dopo una serie di rimbalzi colpisce la retina dell'occhio che percepisce il segnale e ricostruisce l'immagine. L'algoritmo Light Tracing funziona esattamente in questo modo, ossia esamina i raggi uscenti dalla sorgente luminosa. In realtà sfrutta un doppio Ray Tracing combinando i tracciati uscenti dalle luci e dall'occhio e di conseguenza utilizza tutti i metodi di ottimizzazione del calcolo affrontati nei paragrafi precedenti.

9.6.1. Algoritmo Light-Tracing. Questo algoritmo stima il *Potenziale* o *L'equazione di importanza* per ogni singolo pixel. L' equazione di importanza, duale a quella del rendering, è:

$$W(\boldsymbol{x} \to \boldsymbol{\theta}) = W_e(\boldsymbol{x} \to \boldsymbol{\theta}) + \int_{\Omega_{\boldsymbol{x}}} W(\boldsymbol{x} \leftarrow \boldsymbol{\psi})\, f_r(\boldsymbol{x}, \boldsymbol{\theta} \leftrightarrow \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \; d\omega(\boldsymbol{\psi})\,.$$

L'equazione per un singolo pixel è:

$$P = \int_{sorgenti} W(\boldsymbol{x} \to \boldsymbol{\psi})\, L_e(\boldsymbol{x} \to \boldsymbol{\psi})\, \langle \boldsymbol{\psi}, \boldsymbol{n}_{\boldsymbol{x}} \rangle \; dA(\boldsymbol{x})\, d\omega(\boldsymbol{\psi})\,.$$

Il funzionamento di questo algoritmo viene mostrato in Figura 9.6.1. L'equazione 9.6.1 viene stimata utilizzando il metodo di integrazione Monte Carlo. I punti $\boldsymbol{x}_i$ e le direzioni $\boldsymbol{\theta}_i$ sulla sorgente luminosa vengono generati in base al peso derivante dalla stima dell'importanza $W(\boldsymbol{x} \leftarrow \boldsymbol{\theta})$. Questa stima alla fine stabilisce il flusso luminoso attraverso un pixel.

La stima dell'importanza $W(\boldsymbol{x} \leftarrow \boldsymbol{\theta})$ necessita di una integrazione con il metodo Monte Carlo dell'equazione dell'importanza 6.1. Questa stima è molto simile a quella necessaria per l'equazione del rendering nell'algoritmo Ray Tracing stocastico e può essere utilizzato lo stesso schema: emisfero, area, BRDF, e tutti gli altri passi come prima.

Anche i raggi di ombra vengono calcolati in modo analogo al Ray Tracing stocastico, si tiene conto della geometria che occlude la scena a determinati raggi provenienti dalla sorgente con un fattore di occlusione.

L'algoritmo appena descritto è piuttosto inefficiente in quanto il calcolo deve essere implementato per ogni singolo pixel dell'immagine. Però il calcolo puo' essere elaborato in parallelo per i vari pixel: ogni raggio cdi luce verso l'osservatore contribuisce radianza ad uno dei pixel, e basta totalizzare separatamente i risutati ottenuti da ciascun pixel nel corso del tracciamento.

Poiché i pixel, per una immagine di risoluzione tipica, sono parecchi milioni e di solito le sorgenti di luce sono molto meno, questo metodo è comunque molto meno efficiente a causa dell'elevato numero di raggi di ombra da tracciare. Diventa però indispensabile quando si devono visualizzare caustiche, ossia effetti di luce dati dal concentrarsi della luce inviata ad una superficie diffusiva da

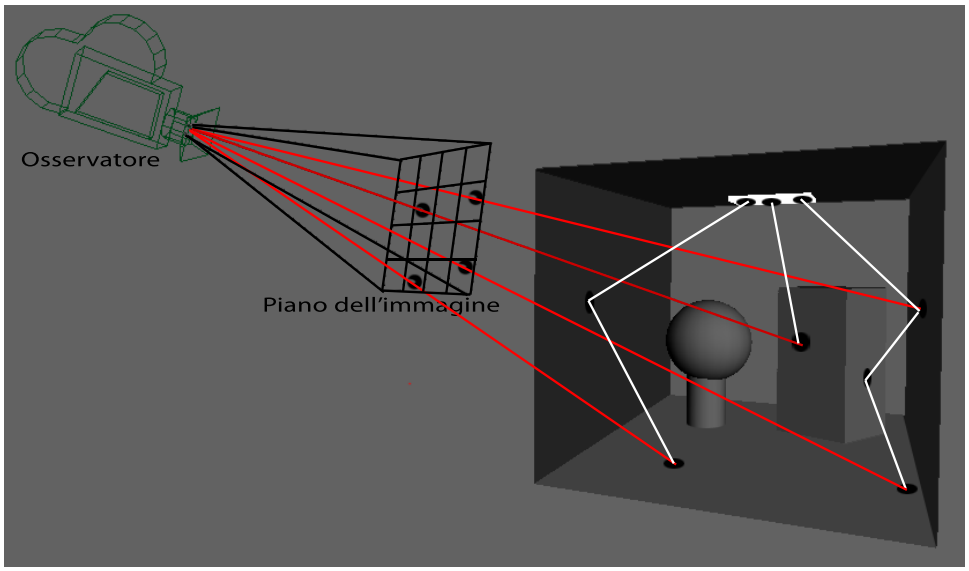


FIGURA 9.6.1. Schema dei tracciati utilizzati dall'algoritmo Light Tracing

uno specchio concavo o da una lente: in prossimità della caustica il contrasto è elevatissimo e servono quindi moltissimi raggi che colpiscano quella zona. Il Ray Tracing distribuisce raggi senza sapere come verranno concentrati da specchi o lenti, ma il Light Tracing li concentra automaticamente perché segue il percorso dei fotoni attraverso proprio quegli specchi e lenti. Quindi, per rendere le caustiche, sono necessari metodi multipass in cui uno dei passi si basi sul Light Tracing, come vedremo nella Sottosezione 11.3.2 dedicata al Photon Mapping.

CAPITOLO 10

Radiosità stocastica

10.1. Revisione della radiosità classica in base all'equazione del rendering

Come è noto il metodo di calcolo della radiosità si basa sulla suddivisione della scena in elementi *(patches)*, cui vengono attribuite luminosità costanti, e che possono a loro volta emettere luce propria. L'algoritmo produce un approssimazione della radiosità della scena a seguito della risoluzione dell'equazione del rendering, discretizzata appositamente, nell'ipotesi che la scena si componga di sole superfici diffusive. Vediamo anzitutto come si possa ottenere una approssimazione lineare discreta dell'equazione integrale del rendering.

Sia N il numero di patches in cui suddividiamo la scena Σ. Indichiamo con b_i la radiosità della patch i-esima, segue dal Capitolo 7 che

$$b_i = \frac{1}{A_i} \int_{S_i} \int_{\Omega(\mathbf{x})} L(\mathbf{x}) \langle \Theta, \mathbf{n}_\mathbf{x} \rangle \, d\omega(\Theta) dA(\mathbf{x}), \tag{10.1.1}$$

essendo $A_i = |S_i|$ l'area della patch i-esima ed $\mathbf{n}_\mathbf{x}$ la normale uscente al punto $\mathbf{x}$. Poiché ci interessiamo al caso di superfici puramente diffusive, la radianza $L(\mathbf{x})$ è indipendente dalla direzione di uscita, precisamente

$$L(\mathbf{x}) = L_e(\mathbf{x}) + \int_{\Omega(\mathbf{x})} f_r(\mathbf{x}) L(\mathbf{x} \leftarrow \Psi) \langle \Psi, \mathbf{n}_\mathbf{x} \rangle \, d\omega(\Psi). \tag{10.1.2}$$

La radianza entrante in $\mathbf{x}$ è ancora funzione dell'angolo di incidenza ma, come già è stato osservato nel Capitolo 7, se $\mathbf{y} = r(\mathbf{x}, -\Psi)$ è il punto visibile da $\mathbf{x}$ in direzione $-\Psi$, allora $L(\mathbf{x} \leftarrow \Psi) = L(\mathbf{y} \rightarrow -\Psi) = L(\mathbf{y})$, dunque l'equazione (10.1.2) può essere riformulata come un integrale sull'intera scena Σ, indipendente dalle direzioni

$$L(\mathbf{x}) = L_e(\mathbf{x}) + f_r(\mathbf{x}) \int_\Sigma L(\mathbf{y}) \mathcal{K}(\mathbf{x}, \mathbf{y}) \, dA(\mathbf{y}) \tag{10.1.3}$$

dove indichiamo con $\mathcal{K}(\mathbf{x}, \mathbf{y}) = G(\mathbf{x}, \mathbf{y}) V(\mathbf{x}, \mathbf{y})$ il prodotto del fattore geometrico e di quello di visibilità. Sia $b(\mathbf{x})$ la radiosità del punto $\mathbf{x}$. Dalla relazione (Capitolo 7) $b(\mathbf{x}) = \int_{\Omega(\mathbf{x})} L(\mathbf{x}) \langle \Theta, \mathbf{n}_\mathbf{x} \rangle \, d\omega(\Theta) = \pi L(\mathbf{x})$ e da (10.1.1) segue che

$$b_i = \frac{1}{A_i} \int_{S_i} L(\mathbf{x}) \int_{\Omega(\mathbf{x})} \langle \Theta, \mathbf{n}_\mathbf{x} \rangle \, d\omega(\Theta) dA(\mathbf{x}) = \frac{1}{A_i} \int_{S_i} b(\mathbf{x}) \, dA(\mathbf{x}). \tag{10.1.4}$$

Introducendo la condizione che la radiosità ed il fattore di riflessione bidirezionale siano costanti sulle singole patch, possiamo modificare l'equazione ottenuta come segue

$$\begin{aligned} b_i &= \frac{1}{A_i} \left(\int_{S_i} b^{(e)}(\mathbf{x}) \, dA(\mathbf{x}) + \int_{S_i} f_r(\mathbf{x}) \int_\Sigma \mathcal{K}(\mathbf{x}.\mathbf{y}) b(\mathbf{y}) \, dA(\mathbf{y}) dA(\mathbf{x}) \right) \\ &= b_i^{(e)} + \sum_{j=1}^N \frac{1}{A_i} \int_{S_i} \int_{S_j} f_r(\mathbf{x}) \mathcal{K}(\mathbf{x}, \mathbf{y}) b(\mathbf{y}) \, dA(\mathbf{y}) dA(\mathbf{x}) = b_i^{(e)} + \rho_i \sum_{j=1}^N F_{ij} b_j \end{aligned} \tag{10.1.5}$$

essendo $\rho_i = f_r(\mathbf{x})$ il fattore di riflessione costante per $\mathbf{x} \in S_i$ ed F_{ij} il *fattore di forma* dalla patch i alla patch j

$$F_{ij} = \frac{1}{A_i} \int_{S_i} \int_{S_j} \mathcal{K}(\mathbf{x}, \mathbf{y})\, dA(\mathbf{y}) dA(\mathbf{x}). \tag{10.1.6}$$

Le approssimazioni fatte ci hanno permesso di formulare l'equazione integrale della radiosità nella forma di un sistema lineare (10.1.5), ovvero di ricondurre il problema al calcolo del vettore $\mathbf{b}$ soluzione di

$$M\mathbf{b} = \mathbf{b}^{(e)}, \quad (M)_{ij} = \delta_{i,j} - \rho_i F_{ij}\,. \tag{10.1.7}$$

L'approccio classico alla realizzazione del rendering di una scena tramite radiosità si compone quindi di quattro macro-operazioni costitutive per l'*algoritmo della radiosità classico*:

- (*i*) Suddividere la scena in un numero finito N di patches, assegnare ad ogni patch un coefficiente di riflettività ρ_i .
- (*ii*) Calcolare F_{ij} per ogni $i = 1, \ldots, N$, $j = 1, \ldots, N$ risolvendo (o approssimando numericamente) l'integrale quadridimensionale (10.1.6).
- (*iii*) Risolvere il sistema lineare della radiosità (10.1.7) numericamente, per mezzo di metodi iterativi quali Jacobi o Gauss–Seidel.
- (*iv*) Visualizzare i risultati ottenuti, ovvero, ad esempio, trasferire i valori di radiosità ottenuti per gli elementi $i = 1, \ldots, N$ a valori per i loro vertici (Sottosezione 6.2.1), poi convertire ogni valore di radiosità in valori di colore, ad esempio tramite coordinaate RGB, poi interpolare secondo Gouraud (Sezione 3.1), ed infine visualizzare sul viewport la scena così illuminata tramite, ad esempio, $z-$buffer o Ray Tracing (Capitolo 1).

Benché non sia del tutto ovvio si intuisce come siano i punti (*ii*) e (*iii*) a dominare il costo computazionale dell'algoritmo. In particolare, forse ancora meno ovvio, la complessità di (*iii*) è spesso trascurabile rispetto a quella richiesta da (*ii*). In effetti, se N è il numero di elementi della scena, ciascuna iterazione nel procedimento al punto (*iii*) si spezza in N^2 moltiplicazioni, dati dall'azione della matrice di iterazione su vettori approssimanti. Infatti l'azione di una matrice su un vettore in $\mathbb{R}^n$ richiede n moltiplicazioni per ciascuna delle n righe). Questi passi devono essere ripetuti iterativamente, e si richiede in media un numero di iterazioni che chiamiamo m, con complessità complessiva $N^2 m$. (ma la velocità di convergenza è esponenziale, e quindi il numero medio m di iterazioni non è elevato). Quindi la complessità in (*iii*) è $\mathcal{O}(N^2 m)$.

Anche il procedimento al punto (*ii*) si spezza in N^2 passi, che consistono, per ogni elemento, nel calcolare il fattore di forma verso ogni altro elemento. Tuttavia, come abbiamo già visto nella relazione di reciprocità (6.2.2), e ridimostreremo fra breve grazie all'equazione del rendering, a meno di un fattore di scala $\alpha_{ij} \in \mathbb{R}$ vale la relazione $F_{ij} = \alpha_{ij} F_{ji}$, dunque dal fattore di forma $i \rightsquigarrow j$ possiamo facilmente ricavare il trasposto $j \rightsquigarrow i$. Ne segue che il numero di passi complessivi di cui abbiamo bisogno è dell'ordine di $\mathcal{O}(N^2/2)$. Ma ciascuno degli $N^2/2$ passi richiede il calcolo numerico dell'integrale quadruplo del fattore di forma. Se questo calcolo fosse approssimato tramite $z-$buffer emicubici, come nella Sottosezione 6.3.2, allora, invece, per ogni elemento occorrerebbe calcolare uno $z-$-buffer di N elementi, ma per il calcolo di questi $z-$buffer occorre proiettare i singoli triangoli della scena e non direttamente gli elementi (che sono superficie, date da unioni di triangoli, e non sapremmo come proiettare). Ogni elemento consiste di una unione di un numero grande di triangoli, diciamo, in media k, e quindi ogni $z-$buffer richiede di proiettare Nk triangoli. Per ciascuno di questi nk triangoli si effettua la scansione di un triangolo e l'interpolazione per aggiornare lo $z-$buffer, operazione a questo punto veloce. Quindi la complessità del procedimento è $\mathcal{O}(N^3 k/2)$.
Ma supponiamo invece, seguendo le linee dello sviluppo della Illuminazione Globale, di voler approssimare gli integrali per il calcolo degli F_{ij} con metodi di Monte Carlo (voler calcolare diversamente tali integrali certamente aumenterebbe la mole di operazioni richieste). Si intuisce da subito come

la complessità che otterremo sarà strettamente dipendente dalla scelta della distribuzione di probabilità per i campioni dell'estimatore di Monte Carlo. Per ogni fattore di forma richiediamo una precisione di τ cifre significative con probabilità $1 - 10^{-\tau}$. Indichiamo con $\mathcal{S}_{ij}^{(n)}$ l'estimatore di F_{ij} con n campioni aleatori. Supponiamo di considerare una distribuzione di probabilità congiunta (di quattro variabili aleatorie) p tale che

$$X = (X_1, X_2, X_3, X_4),\ \Psi_{ij}(X) = \frac{\mathcal{K}(X)}{A_i p(X)} \chi_{(S_i \times S_j)}(X)$$
$$\Longrightarrow \mathrm{Var}\left(\Psi_{ij}(X)\right) \leqslant E\left(\Psi_{ij}(X)^2\right) \approx 1 \quad (10.1.8)$$

Dalla disuguaglianza di Tschebyschev segue che

$$n(\tau) \approx 10^{3\tau} \Longrightarrow \mathbf{P}\left(\left|\mathcal{S}_{ij}^{(n(\tau))} - F_{ij}\right| < 10^{-\tau}\right) \approx 1 - 10^{-\tau} \quad (10.1.9)$$

Tale numero di campioni è necessario per ogni estimatore $\mathcal{S}_{ij}$. Quindi, per ciascuno degli N elementi, occorre calcolare il fattore di forma verso ciascun altro di questi elementi, e la complessità di questo calcolo è $n(\tau)\mathcal{O}(N)$. Pertanto il costo complessivo del punto (ii) è circa $n(\tau)\mathcal{O}(N^3)$, molto maggiore della complessità $\mathcal{O}(N^3)$ del punto (iii).

10.2. Ricalcolo dei fattori di forma e le loro proprietà a partire dall'equazione del rendering

Il calcolo dei fattori di forma è, dunque, un problema importante e complesso (dal punto di vista computazionale), di centrale importanza nell'algoritmo di radiosità .

PROPOSIZIONE 10.2.1. *Siano F_{ij}, $i = 1, \ldots, N$, $j = 1, \ldots, N$ come in (10.1.6), e ricordiamo che $\mathcal{K}(\mathbf{x}, \mathbf{y}) = G(\mathbf{x}, \mathbf{y})\, V(\mathbf{x}, \mathbf{y})$, e*

$$G(\mathbf{x}, \mathbf{y}) = \frac{\langle \overrightarrow{\boldsymbol{xy}},\, \boldsymbol{n_x} \rangle \langle \overrightarrow{\boldsymbol{yx}},\, \boldsymbol{n_y} \rangle}{\pi \|\boldsymbol{x} - \boldsymbol{y}\|^2},$$

dove, come prima, $\overrightarrow{\boldsymbol{xy}}$ è il versore da $\mathbf{x}$ in direzione di $\mathbf{y}$. Se Σ è una scena chiusa, ossia se la proiezione radiale della scena su un emisfero intorno a qualsiasi suo punto è surgettiva, allora valgono le seguenti proprietà :

$$F_{ij} \geqslant 0 \qquad \forall\, (i, j) \in [1, N] \times [1, N], \quad (10.2.1a)$$

$$\sum_{j=1}^{N} F_{ij} = 1 \qquad \forall\, i \in [1, N], \quad (10.2.1b)$$

$$A_i F_{ij} = A_j F_{ji} \qquad \forall\, (i, j) \in [1, N] \times [1, N]. \quad (10.2.1c)$$

DIMOSTRAZIONE. La condizione di non-negatività (10.2.1a) è una conseguenza diretta della definizione di $\mathcal{K}$, infatti certamente $G(\mathbf{x}, \mathbf{y}) > 0$ a meno che $\mathbf{x}$ ed $\mathbf{y}$ non siano complanari, inoltre $V(\mathbf{x}, \mathbf{y}) \in \{0, 1\}$, dunque $\mathcal{K}(\mathbf{x}, \mathbf{y}) \geqslant 0$. Per provare (10.2.1b) si osservi che dal confronto fra (10.1.2) e (10.1.3) (o più direttamente dalle considerazioni fatte nel Capitolo 7) segue che

$$\pi \int_{\Sigma} \mathcal{K}(\mathbf{x}, \mathbf{y})\, dA(\mathbf{y}) = \int_{\Omega(\mathbf{x})} \langle \Psi, \mathbf{n}_{\mathbf{x}} \rangle\ d\omega(\Psi). \quad (10.2.2)$$

Pertanto si ha

$$\sum_{j=1}^{N} F_{ij} = \frac{1}{A_i}\int_{S_i}\sum_{j=1}^{N}\int_{S_j}\mathcal{K}(\mathbf{x},\mathbf{y})dA(\mathbf{y})dA(\mathbf{x}) = \frac{1}{A_i}\int_{S_i}\int_{\Sigma}\mathcal{K}(\mathbf{x},\mathbf{y})dA(\mathbf{x})$$
$$= \frac{1}{A_i}\int_{S_i}\frac{1}{\pi}\int_{\Omega(\mathbf{x})}\langle\Psi,\mathbf{n}_{\mathbf{x}}\rangle\, d\omega(\Psi)dA(\mathbf{x}) = \frac{1}{A_i}\int_{S_i} dA(\mathbf{x}) = 1.$$

Si osservi che, se la scema non è chiusa, la somma dei fattori di forma vale meno di 1. Questa proprietà è fisicamente giustificata dalla conservazione dell'energia o, analogamente, dell'angolo solido: l'energia per unità di area, emessa dalla patch i-esima che raggiunge tutte le altre patches della scena (essendo Σ chiusa) deve essere pari ad 1.

La proprietà (10.2.1c) è conseguenza diretta dell'uguaglianza $\mathcal{K}(\mathbf{x},\mathbf{y}) = \mathcal{K}(\mathbf{y},\mathbf{x})$. □

Osserviamo che il sistema (10.1.7) della radiosità può essere formulato in maniera analoga per le potenze emesse, mantenendo una struttura analoga. Indichiamo con $D(\mathbf{v})$ la matrice diagonale i cui elementi sono gli elementi del vettore $\mathbf{v}$, con $A \in \mathbb{R}^N$ il vettore delle aree delle patches (ovvero $(A)_i = A_i$), con $\boldsymbol{\rho}$ il vettore dei fattori di riflettività (ovvero $(\boldsymbol{\rho})_i = \rho_i = f_r(\mathbf{x})$ per $\in S_i$) e con F la matrice dei fattori di forma $(F)_{ij} = F_{ij}$. Se w_i è la potenza emessa dalla patch i-esima, allora (come abbiamo visto nel Capitolo 7) $w_i = A_i b_i$, pertanto dalla proprietà (10.2.1c) segue che

$$M\mathbf{b} = (I - D(\boldsymbol{\rho})F)\mathbf{b} = \mathbf{b}^{(e)} \Leftrightarrow D(A)(I - D(\boldsymbol{\rho})F)\mathbf{b} = D(A)\mathbf{b}^{(e)}$$
$$\Leftrightarrow (I - D(\boldsymbol{\rho})F^T)D(A)\mathbf{b} = D(A)\mathbf{b}^{(e)} \Leftrightarrow (I - D(\boldsymbol{\rho})F^T)\mathbf{w} = \mathbf{w}^{(e)}.$$

Dunque (10.1.5) è equivalente a

$$w_i = w_i^{(e)} + \sum_{j=1}^{N}\rho_i w_j F_{ji}, \quad i = 1,2,\ldots,N. \tag{10.2.3}$$

Tale sistema di equazioni ha un interessante interpretazione fisica: la potenza w_i emessa dalla patch i-esima consiste della potenza $w_i^{(e)}$, che essa stessa genera, più la percentuale di potenza w_j che riceve riflessa dalle altre $j = 1,\ldots,N$ patches; il fattore di forma F_{ij} indica la frazione di potenza emessa da i che raggiunge j. Pensare ad i fattori di forma in questi termini suggerisce un metodo per darne una semplice stima in termini probabilistici:

Stima dei fattori di forma con raggi locali. Consideriamo i due vettori aleatori $(X_i, Y_i) = X(i)$, uniformemente distribuito su S_i, e $\Theta = (\Theta_1, \Theta_2)$ dipendente da $X(i)$, con densità di distribuzione

$$p_{X(i)}(x_i, y_i) = \frac{1}{A_i}, \quad p_{\Theta|X(i)}(\theta_1,\theta_2) = \frac{1}{\pi}\left\langle(\theta_1,\theta_2),\mathbf{n}(x_i,y_i)\right\rangle, \tag{10.2.4}$$

indichiamo con $\mathbf{x}_i = (x_i, y_i)$ e con $\theta = (\theta_1,\theta_2)$, la densità congiunta di $(X(i),\Theta)$ risulta essere

$$p_{(X(i),\Theta)}(\mathbf{x}_i,\theta) = p_{X(i)}(\mathbf{x}_i)\cdot p_{\Theta|X(i)}(\theta) = \frac{\langle\theta,\mathbf{n}(\mathbf{x}_i)\rangle}{\pi A_i}. \tag{10.2.5}$$

Osservazione 10.2.2. La scelta della distribuzione di Θ è ben posta, infatti sono soddisfatte le due proprietà di *non-negatività* e di *normalizzazione* della densità congiunta ottenuta, più precisamente

$$(i) \qquad p_{(X(i),\Theta)}(\mathbf{x}_i,\theta) \geqslant 0, \quad \forall(\mathbf{x}_i,\theta) \in S_i \times \Omega(\mathbf{x})$$

$$(ii) \qquad \left\|p_{(X(i),\Theta)}\right\|_{L^1(S_i\times\Omega(\mathbf{x}_i))} = \int_{S_i\times\Omega(\mathbf{x}_i)} p_{(X(i),\Theta)}(\mathbf{x}_i,\theta)\, d\omega(\theta)dA(\mathbf{x}_i)$$
$$= \frac{1}{\pi}\int_{S_i}\frac{1}{A_i}\int_{\Omega(\mathbf{x}_i)}\langle\theta,\mathbf{n}(\mathbf{x}_i)\rangle\, d\omega(\theta)dA(\mathbf{x}_i) = 1$$

Vediamo come ricavare una formula per i fattori di forma F_{ij}, facendo uso delle due variabili aleatorie che abbiamo considerato. Definiamo n_i vettori aleatori $Y_1^{(i)}, \ldots, Y_{n_i}^{(i)}$ indipendenti ed uniformemente distribuiti, con densità $p_{Y_k^{(i)}} = p_{(X(i),\Theta)} = p$ per ogni $k = 1, \ldots, n_i$. Sia $f_j(\mathbf{x}_i, \Psi) = \chi_j(\mathbf{x}_i, \Psi)p(\mathbf{x}_i, \Psi)$, dove $\chi_j(\mathbf{x}_i, \Psi)$ vale 1 se un raggio da $\mathbf{x}_i \in S_i$ in direzione Ψ colpisce la patch j, vale zero altrimenti. Sia $Y_k^{(i)} = (X_k^{(i)}, \Theta_k^{(i)})$, consideriamo l'estimatore di Monte Carlo

$$\langle F_{ij} \rangle_{n_i} = \frac{1}{n_i} \sum_{k=1}^{n_i} \frac{f_j\left(Y_k^{(i)}\right)}{p\left(Y_k^{(i)}\right)} = \frac{1}{n_i} \sum_{k=1}^{n_i} \chi_j\left(Y_k^{(i)}\right). \tag{10.2.6}$$

Osservazione 10.2.3. Per ogni n_i risulta che $E\left(\langle F_{ij} \rangle_{n_i}\right) = F_{ij}$, per ogni $i = 1, \ldots, N$, $j = 1, \ldots, N$ (l'estimatore è *unbiased*), inoltre $\mathrm{Var}\left(\langle F_{ij} \rangle_{n_i}\right) = \frac{F_{ij}}{n_i}(1 - F_{ij})$.

*Dimostrazione*Tenendo conto della relazione (10.2.2) si ha

$$\begin{aligned}
E\left(\langle F_{ij} \rangle_{n_i}\right) =& \frac{1}{n_i} \sum_{k=1}^{n_i} E\left(\chi_j(Y_k^{(i)})\right) \\
=& \frac{1}{n_i} \sum_{k=1}^{n_i} \int_{S_i \times \Omega(\mathbf{x})} f_j(\mathbf{x}, \Psi)\, dA(\mathbf{x}) d\omega(\Psi) \\
=& \frac{1}{\pi} \int_{S_i} \frac{1}{A_i} \int_{\Omega(\mathbf{x})} \chi_j(\mathbf{x}, \Psi) \langle \Psi, \mathbf{n}_\mathbf{x} \rangle \, d\omega(\Psi) dA(\mathbf{x}) \\
=& \frac{1}{A_i} \int_{S_i} \int_{\Sigma} \chi_j(\mathbf{x}, \mathbf{y}) \mathcal{K}(\mathbf{x}, \mathbf{y})\, dA(\mathbf{y}) dA(\mathbf{x}) = F_{ij}
\end{aligned}$$

dove $\chi_j(\mathbf{x}, \mathbf{y})$ vale 1 solo se $\mathbf{y} \in S_j$ (vale zero altrimenti), essendo $\mathbf{y} = r(\mathbf{x}, -\Psi)$ il "dirimpettaio" di $\mathbf{x} \in S_i$ in direzione $-\Psi$. Per quanto riguarda la varianza, da quanto osservato nel Capitolo 8, indicata con $Y^{(i)} = (X^{(i)}, \Theta^{(i)})$ una qualsiasi delle n_i variabili $Y_k^{(i)}$, segue che

$$\begin{aligned}
\mathrm{Var}\left(\langle F_{ij} \rangle_{n_i}\right) =& \frac{1}{n_i} \mathrm{Var}\left(\chi_j(Y^{(i)})\right) \\
=& \frac{1}{n_i} \int_{S_i \times \Omega(\mathbf{x})} \left(\chi_j(\mathbf{x}, \Psi) - F_{ij}\right)^2 p\left(\mathbf{x}, \Psi\right) \, d\omega(\Psi) dA(\mathbf{x}) \\
=& \frac{1}{n_i} \int_{S_i \times \Omega(\mathbf{x})} \left(\chi_j^2 \cdot p + F_{ij}^2 \cdot p - 2\chi_j \cdot p\right)(\mathbf{x}, \Psi)\, d\omega(\Psi) dA(\mathbf{x}) \\
=& \frac{F_{ij}(1 - F_{ij})}{n_i}.
\end{aligned}$$

□

Pertanto possiamo interpretare il fattore di forma F_{ij} come la probabilità che un raggio tracciato da un punto casuale $\mathbf{x}_i \in S_i$ nella direzione aleatoria Θ, raggiunga la patch S_j, e possiamo stimare tale probabilità con metodi di Monte Carlo, commettendo un errore aleatorio tanto minore quanti più campioni n_i scegliamo di usare.

Tale metodo, che prende il nome di *local lines sampling*, permette di ridurre drasticamente la complessità computazionale della *macro-operazione* (*ii*) nell'algoritmo di radiosità , migliorando la stima grossolana con ne avevamo dato: Essendo $0 \leqslant F_{ij} < 1$ per ogni i e j, dalla disuguaglianza di

Tchebycheff segue che

$$\mathbf{P}\left(\left|\langle F_{ij}\rangle_{n_i} - F_{ij}\right| < 10^{-\tau}\right) \geqslant 1 - \frac{10^{2\tau} F_{ij}(1-F_{ij})}{n_i} \geqslant 1 - \frac{10^{2\tau}}{4n_i},$$

per ogni $\tau > 0$, dunque certamente per ogni fattore di forma, scegliendo $n_i = n_i(\tau) \geqslant \frac{10^{3\tau}}{4}$, risulta

$$\mathbf{P}\left(\left|\langle F_{ij}\rangle_{n_i} - F_{ij}\right| < 10^{-\tau}\right) \geqslant 1 - 10^{-\tau},$$

il che migliora la stima precedente di un fattore $\frac{1}{4}$. D'altra parte dalle proprietà (10.2.1a) e (10.2.1b) ci aspettiamo che di frequente F_{ij} sia molto minore di 1, ad esempio che mediamente si abbia $F_{ij} \approx \frac{1}{N}$. Da ciò segue che il numero di campioni necessari, in media, per stimare con probabilità $1 - 10^{-\tau}$ un fattore di forma F_{ij}, con precisione $10^{-\tau}$, è approssimativamente $n_i(\tau) \approx \frac{10^{3\tau}}{N}$, essendo $F_{ij}(1 - F_{ij}) \approx 1/N$.

Stima dei fattori di forma con raggi globali. L'algoritmo precedente viene implementato tramite il tracciamento di n_i raggi che partono in direzioni Θ aleatorie da $\mathbf{x}_i \in S_i$, per ogni $i = 1, \ldots, N$ e misurando quanti di questi colpiscono la patch j-esima. Esistono, purtuttavia, alcune varianti di tale metodo che permettono di stimare F_{ij} tracciando un certo numero n di raggi *globali* per tutta la scena, uniformemente distribuiti. Ad esempio si può procedere racchiudendo la scena (di cui a priori conosciamo la geometria) entro una calotta sferica e sparare dei raggi partendo da un punto a caso sulla superficie della sfera. Si può dimostrare che se
- n è il numero complessivo di raggi tracciati,
- A_T è l'area complessiva delle patches (ovvero $A_T = \sum_{i=1}^{N} A_i$),
- $n(i)$ è il numero di raggi che raggiungono la patch i,
- $n(i,j)$ è il numero di raggi che attraversando la patch i raggiungono la patch j, allora risulta

$$n(i) \approx n\frac{A_i}{A_T}, \quad F_{ij} \approx \frac{n(i,j)}{n(i)}\,.$$

Il vantaggio principale di questo procedimento sul procedimento di tracciamento locale è dovuto alla possibilitè di sfruttare la *coerenza* nella geometria della scena per generare più efficentemente le linee globali, tuttavia questo tipo di approccio non permette di scegliere il numero di raggi in maniera adattiva, modificandolo di volta in volta a seconda della patch considerata.

10.3. Radiosità stocastica

Abbiamo osservato come ridurre il costo computazionale dovuto ai fattori di forma, dell'algoritmo di radiosità , che stiamo studiando. Come abbiamo sottolineato più volte il calcolo di tali fattori è molto oneroso e, pertanto, sono stati introdotti metodi probabilistici basati su tecniche di Monte Carlo per darne delle stime con una sostanziale riduzione del costo computazionale. Tuttavia, una difficoltà ulteriore cui fin ora abbiamo sottaciuto è quella dovuta alla quantità di memoria necessaria per poter risolvere il sistema (10.1.7). Basta poco a convincersi che questo è un problema non-irrilevante, nel caso in cui tutti i fattori di forma fossero necessari per definire la matrice M e per risolvere il sistema lineare della radiosità . In questa sezione vedremo come sia possibile, sulla base di quanto visto fin ora, produrre dei metodi per risolvere (10.1.7) senza dover in verità calcolare **nessun** fattore di forma, sfruttando alcune tecniche di teoria della probabilità . I metodi che vedremo nel dettaglio sono metodi di rilassamento stocastico e metodi basati su catene di Markov discrete.

10.3.1. Metodi di rilassamento stocastici. Rientrano in questa classe quei metodi per la risoluzione numerica di sistemi lineari che, basati su schemi iterativi di punto fisso (come Jacobi o Gauss–Seidel), riducono il costo computazionale delle iterazioni (che nel caso classico è $\mathcal{O}(n^2)$) stimando con tecniche di Monte Carlo i prodotti *matrice-vettore* richiesti dal metodo.

Richiamiamo la struttura del sistema lineare cui siamo interessati:

$$M\mathbf{b} = \mathbf{b}^{(e)}, \quad M = I - D(\boldsymbol{\rho})F$$

essendo $\boldsymbol{\rho}$ il vettore dei coefficienti di riflettività ed F la matrice dei fattori di forma. Limitiamoci a considerare lo schema iterativo di Jacobi. Come è noto, se D è la matrice diagonale $(D)_{ii} = (M)_{ii}$ ed $A = D - M$, il metodo di Jacobi consiste nel definire $\Phi(\mathbf{u}) = D^{-1}(\mathbf{b}^{(e)} + A\mathbf{u})$ e costruire la successione di vettori $\mathbf{b}_{k+1} = \Phi(\mathbf{b}_k)$, $k = 0, 1, 2, \ldots$ Osserviamo, in particolare, che $D = I$ e che le iterazioni del metodo assumono la forma

$$\mathbf{b}_0 \in \mathbb{R}^N, \quad \mathbf{b}_{k+1} = \Phi_F(\mathbf{b}_k) = \mathbf{b}^{(e)} + D(\boldsymbol{\rho})F\mathbf{b}_k \quad k = 0, 1, 2, \ldots \tag{10.3.1}$$

Abbiamo già osservato che il sistema lineare della radiosità è equivalente al sistema delle potenze emesse (10.2.3), pertanto possiamo riscrivere le iterazioni di Jacobi nella forma

$$\mathbf{w}_0 \in \mathbb{R}^N, \quad \mathbf{w}_{k+1} = \Phi_{F^T}(\mathbf{w}_k) = \mathbf{w}^{(e)} + D(\boldsymbol{\rho})F^T\mathbf{w}_k \quad k = 0, 1, 2, \ldots \tag{10.3.2}$$

Scegliamo $\mathbf{w}_0 = \mathbf{w}^{(e)}$, ne segue che

$$\mathbf{w}_{k+1} = (\Phi_{F^T} \circ \cdots \circ \Phi_{F^T})(\mathbf{w}^{(e)}) = \Phi^k_{F^T}(\mathbf{w}^{(e)}) = \sum_{\nu=0}^{k} \left(D(\boldsymbol{\rho})F^T\right)^\nu \mathbf{w}^{(e)}$$

Allora possiamo costruire uno schema iterativo equivalente, considerando la potenza incrementale delle patches, ovvero

$$\Delta\mathbf{w}_0 = \mathbf{w}^{(e)}, \quad \Delta\mathbf{w}_{k+1} = D(\boldsymbol{\rho})F^T\Delta\mathbf{w}_k, \quad k = 0, 1, 2, \ldots \tag{10.3.3}$$

osservando che, così facendo, il vettore delle potenze calcolato con Jacobi al passo m-esimo coincide esattamente con la somma dei primi m incrementi $\Delta\mathbf{w}_j$, ovvero $\mathbf{w}_m = \sum_{j=0}^{m} \Delta\mathbf{w}_j$.

Dal punto di vista del metodo iterativo puramente deterministico non vi sono differenze (in termini di complessità per iterazione) fra gli schemi (10.3.1), (10.3.2) e (10.3.3), mentre questa ultima formulazione è ottimale se si usano tecniche di Monte Carlo per stimare le approssimanti k-esime. Abbiamo osservato, nel metodo delle *linee locali*, che il fattore di forma F_{ij} può essere interpretato come la probabilità che un raggio tracciato in un direzione casuale Θ da un punto a caso della patch i, atterri sulla patch j. Scegliamo con probabilità $p_k(j)$ la patch j al passo k-esimo, dove

$$p_k(j) = \frac{(\Delta\mathbf{w}_k)_j}{\sum_{i=1}^{N}(\Delta\mathbf{w}_k)_i} = \frac{\Delta w_j^{(k)}}{\Delta w_T^{(k)}} \tag{10.3.4}$$

e sia F_{ij} la distribuzione di probabilità condizionale di colpire j dato che è stata scelta la patch i. La probabilità complessiva di selezionare una coppia di patch (i, j) in tale modo è

$$p_k(i, j) = p_k(i) \cdot F_{ij} = \frac{F_{ij}\Delta w_i^{(k)}}{\Delta w_T^{(k)}}. \tag{10.3.5}$$

È chiaro che l'idea degli estimatori di Monte Carlo che abbiamo sviluppato per approssimare un integrale, può essere estesa alle sommatorie, nel modo seguente: consideriamo l'insieme limitato $J \subset \mathbb{Z}^d$, supponiamo di voler approssimare la somma S di una certa sequenza $u : J \longrightarrow \mathbb{R}$

$$S = \sum_{\underline{i} \in J} u(\underline{i}), \quad \underline{i} = (i^{(1)}, \ldots, i^{(d)})$$

Scegliamo una distribuzione discreta di probabilità π ed un insieme di variabili aleatorie $\{\underline{i}_\ell\}$ i.i.d. con distribuzione π. Si osserva, in modo analogo al caso continuo, che per ogni $h > 0$, la seguente

$$\langle S \rangle_h = \frac{1}{h} \sum_{\ell=1}^{h} \frac{u(\underline{i}_\ell)}{\pi(\underline{i}_\ell)}, \quad \underline{i}_\ell \in J, \, \forall \ell \, : \, 1 \leqslant \ell \leqslant h$$

definisce un estimatore *unbiased* per S, ovvero tale che $E(\langle S \rangle_h) = S$.

Riscriviamo (10.3.3) scalarmente, con la notazione introdotta in (10.3.4), introducendo un secondo indice nella sommatoria, per motivi che chiariremo a breve

$$\Delta w_i^{(k+1)} = \sum_{j,r=1}^{N} \Delta w_j^{(k)} F_{jr} \rho_r \delta(r,i) \quad i = 1, \dots, N \tag{10.3.6}$$

essendo δ la delta di Kronecker. Vogliamo produrre un estimatore di Monte Carlo per (10.3.6) facendo uso della distribuzione di probabilità p che abbiamo introdotto in (10.3.5) e ragionando con *local lines*. Selezioniamo a caso $m_i^{(k)}$ patches $\{(i_\nu, j_\nu)\}_\nu$ con distribuzione p, come abbiamo appena richiamato, l'estimatore per (10.3.6) assume la forma

$$\left\langle \Delta w_i^{(k+1)} \right\rangle_{m_i^{(k)}} = \frac{1}{m_i^{(k)}} \sum_{\nu=1}^{m_i^{(k)}} \frac{\Delta w_{j_\nu}^{(k)} F_{j_\nu r_\nu} \rho_{r_\nu} \delta(r_\nu, i)}{p_k(j_\nu, r_\nu)} = \rho_i \frac{n_i^{(k)}}{m_i^{(k)}} \Delta w_T^{(k)} \tag{10.3.7}$$

dove è stato posto $n_i^{(k)} = \sum_{\nu=1}^{m_i^{(k)}} \delta(r_\nu, i)$, la frazione delle $m_i^{(k)}$ patches selezionate che coincide con i.

Dunque possiamo riscrivere (10.3.3) con una formulazione "approssimata", in termini scalari ed in modo che risulti sensibilmente più agevole dal punto di vista computazionale, non essendo **mai** richiesto il calcolo (ne lo storage) dei vari fattori di forma:

$$\begin{cases} \Delta w_i^{(0)} = w_i^{(e)} \\ \Delta w_i^{(k+1)} = \left\langle \Delta w_i^{(k+1)} \right\rangle_{m_i^{(k)}} = \rho_i \frac{n_i^{(k)}}{m_i^{(k)}} \left(\sum_{i=1}^{N} \Delta w_i^{(k)} \right) \quad , \quad i = 1, \dots, N \\ w_i^{(k)} = \sum_{j=0}^{k} \Delta w_i^{(j)} \end{cases} \tag{10.3.8}$$

Concludendo, facciamo notare che un metodo analogo può essere implementato facendo uso di linee globali, si pone $p_k(i,j) = \frac{A_i}{A_T} F_{ij}$, assegnando ad ogni coppia (i,j) la probabilità con la quale una stessa linea globale interseca una coppia di patch i, j mutuamente visibili. Infine va sottolineato che lo schema (10.3.8) può essere ulteriormente esemplificato scegliendo il numero di *campioni* indipendentemente da i e da k (ovvero si sceglie $m_i^{(k)} = m$ per ogni i e k), calcolando contemporaneamente tutte le approssimanti del vettore $\mathbf{\Delta w}_k$, tuttavia è comunque necessario, elemento per elemento, valutare la quantità n_i.

10.3.2. Catene di Markov discrete. La seconda classe di metodi per la risoluzione di (10.1.7), si basa sul concetto di *passeggiata aleatoria su uno spazio degli stati discreto* o, analogamente, su quello di *catena di Markov discreta, finita ed omogenea*. Ci limiteremo a presentare le idee alla base di tali metodi, che non miglioreranno i risultati ottenuti nel caso di "Jacobi stocastico", ma sostanzialmente ne eguaglieranno le prestazioni.

10.4. Catene di Markov finite

Introduciamo preliminarmente il concetto di Catena di Markov finita, discreta ed omogenea, presentandone le proprietà principali.

Sia $\{X_1, \dots, X_n, \dots\} = \mathcal{CM}$ un insieme numerabile di variabili aleatorie discrete che assumo un numero finito di valori $\mathcal{S} = \{1, 2, \dots, n\}$ (detto *spazio degli stati*). Diremo che $\mathcal{CM}$ è omogeneo se ogni variabile aleatoria $X_j \in \mathcal{CM}$ dipende solo dalla variabile $X_{j-1} \in \mathcal{S} \setminus \{X_j\}$ mentre è indipendente

dalle altre $n-2$ variabili $X_i \in \mathcal{CM} \setminus \{X_j, X_{j-1}\}$. Un insieme di variabili aleatorie $\mathcal{CM}$ con tali proprietà e tale che la probabilità con cui X_{j+1} dipende da X_j è la stessa per ogni $j = 1, 2, \dots,$ si dice *Catena di Markov discreta, omogenea, con un numero finito di stati (*o semplicemente *finita).*

Definizione 10.4.1 (Probabilità di transizione). Data una catena di Markov finita, discreta ed omogenea, si chiama probabilità di transizione ad un passo da i a j, la probabilità

$$p_{ij} = \mathbf{P}\left(X_{n+1} = j \mid X_n = i, \dots, X_0 = i_0\right) = \mathbf{P}\left(X_{n+1} = j \mid X_n = i\right),$$

analogamente si dice probabilità di transizione ad m passi da i a j la probabilità

$$p_{ij}^{(m)} = \mathbf{P}(X_m = j \mid X_0 = i).$$

Poiché ci interesseremo solo di catene di Markov finite, discrete ed omogenee, sottintederemo che ogni catena nel seguito considerata abbia tali proprietà .

Definizione 10.4.2 (Matrice di transizione). Sia $\mathcal{CM} = \{X_1, X_2, X_3, \dots\}$ una catena di Markov con spazio degli stati $\mathcal{S} = \{1, 2, \dots, n\}$ e probabilità di transizione da i a j ad un passo p_{ij}. La matrice $P \in \mathbb{R}^{n\times n}$ tale che

$$P = \begin{bmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{bmatrix}$$

si dice *matrice di transizione* della catena.

Osservazione 10.4.3. Ogni matrice di transizione P di una $\mathcal{CM}$ è tale che
(1) P è stocastica per righe;
(2) $(P)_{ij} \geqslant 0$, per ogni $i, j \in \mathcal{S}$.
(3) $P\mathbf{e} = \mathbf{e}$, essendo $\mathbf{e}^T = (1, \dots, 1)$.

*Dimostrazione*La proprietà (2) è ovvia. Mostriamo la proprietà (1) che implica la (3).

$$\sum_{j=1}^{n} p_{ij} = \sum_{j=1}^{n} \mathbf{P}\left(X_{n+1} = j \mid X_n = i\right) = \mathbf{P}\left(\cup_{j\in\mathcal{S}} \{X_{n+1} = j\} \mid X_n = i\right)$$
$$= \mathbf{P}\left(X_{n+1} \in \mathcal{S} \mid X_n = i\right) = 1$$

□

Vale la seguente formula per la probabilità di transizione ad m passi, di fondamentale importanza

Proposizione 10.4.4 (Chapman-Kolmogorov).

$$p_{ij}^{(m)} = \sum_{\nu=1}^{n} p_{i\nu}^{(m-1)} p_{\nu j}$$

da cui segue che la probabilità di transizione da i a j ad m passi è data dall'elemento (i, j) della potenza m-esima della matrice di transizione P, più precisamente $p_{ij}^{(m)} = (P^m)_{ij}$.

*Dimostrazione*Dalla proprietà $\mathbf{P}(A \mid B) = \mathbf{P}(A \cap B)/\mathbf{P}(B)$ segue che

$$\begin{aligned} p_{ij}^{(m)} &= \mathbf{P}\left(X_m = j \mid X_0 = i\right) = \frac{\mathbf{P}\left(X_m = j, X_0 = i\right)}{\mathbf{P}\left(X_0 = i\right)} \\ &= \sum_{\nu=1}^{n} \frac{\mathbf{P}\left(X_m = j, X_{m-1} = \nu, X_0 = i\right)}{\mathbf{P}\left(X_{n-1} = \nu, X_0 = i\right)} \cdot \frac{\mathbf{P}\left(X_{n-1} = \nu, X_0 = i\right)}{\mathbf{P}\left(X_0 = i\right)} \\ &= \sum_{\nu=1}^{n} \mathbf{P}\left(X_n = j \mid X_{m-1} = \nu, X_0 = i\right) \mathbf{P}\left(X_{m-1} = \nu \mid X_0 = i\right) \\ &= \sum_{\nu=1}^{n} p_{i\nu}^{(m-1)} p_{\nu j} \end{aligned}$$

□

DEFINIZIONE 10.4.5. Siano $i, j \in \mathcal{S}$ due indici nello spazio degli stati della catena $\mathcal{S}$, diremo che i *comunica* con j se esiste un $m > 0$ tale che $p_{ij}^{(m)} > 0$. Sia $\mathcal{C} \subseteq \mathcal{S}$ un sottoinsieme dello spazio degli stati. $\mathcal{C}$ si dice una classe chiusa di $\mathcal{S}$ se per ogni $i, j \in \mathcal{S}$ con $i \in \mathcal{C}$ e $j \in \mathcal{C}^{\complement}$ si ha $p_{ij}^{(m)} = 0$ per ogni $m > 0$. Una classe chiusa è dunque un sottoinsieme di stati che "non comunicano mai con l'esterno".

Pertanto se per un certo m si verifica $X_m \in \mathcal{C}$, con $\mathcal{C}$ classe chiusa di $\mathcal{S}$, allora $X_q \in \mathcal{C}$ per ogni $q \geqslant m$. Data la classe chiusa $\mathcal{C}$, introduciamo la seguente variabile aleatoria

$$\tau_i(\mathcal{C}) = \inf_{n \in \mathbb{N}} \{X_n \in \mathcal{C} \mid X_0 = i,\, i \notin \mathcal{C}\} \tag{10.4.1}$$

ovvero il primo "istante" in cui la catena visita $\mathcal{C}$ essendo partita da $i \notin \mathcal{C}$, detta *tempo di prima visita della classe chiusa* $\mathcal{C}$. Vale la seguente

PROPOSIZIONE 10.4.6. *Sia $\mathcal{C}$ una classe chiusa di $\mathcal{S}$ e $T \subset \mathcal{S}$ il sottoinsieme degli stati di $\mathcal{S}$ che non sono in nessuna classe chiusa (detti transitori). Sia $\zeta_i(\mathcal{C}) = E\left(\tau_i(\mathcal{C})\right)$, allora $\zeta_i(\mathcal{C})$ è soluzione della seguente*

$$\zeta_i(\mathcal{C}) = 1 + \sum_{j \in T} p_{ij} \zeta_j(\mathcal{C}). \tag{10.4.2}$$

Si osservi che l'insieme $\nu_i(h)$ delle volte che la catena visita lo stato i-esimo

$$\nu_i(h) = \{m \in [1, h] : X_m = i\}$$

è anch'essa una variabile aleatoria. Indichiamo con $\boldsymbol{\pi}$ il vettore delle distribuzioni iniziali di probabilità , ovvero il vettore tale che
1. $\pi_i \geqslant 0$ per ogni $i \in \mathcal{S}$,
2. $\|\boldsymbol{\pi}\|_1 = 1$,
3. $\pi_i = \mathbf{P}\left(X_0 = i\right)$.
Se $\gamma_i(h)$ è il numero medio di volte che la catena visita i, $\gamma_i(h) = E(\nu_i(h))$, allora $\gamma_i(h)$ è soluzione dell'equazione

$$\gamma_i(h) = h\pi_i + \sum_{j \in \mathcal{S}} (P^T)_{ij}\, \gamma_j(h) = h\pi_i + \sum_{j \in \mathcal{S}} p_{ji}\, \gamma_j(h) \tag{10.4.3}$$

per ogni $i = 1, \ldots, n$. Studiamo, infine, la situazione in cui si volesse sapere con che probabilità partendo dallo stato i-esimo, la catena, dopo un certo numero di passi, vi faccia ritorno. Siamo dunque interessati alla distribuzione di probabilità

$$\chi_i = \mathbf{P}\left(\exists\, m > 0 : X_m = i \mid X_0 = i\right). \tag{10.4.4}$$

che per ogni $i \in \mathcal{S}$, χ_i soddisfa l'equazione

$$\chi_i = \pi_i + \sum_{j\in\mathcal{S}} (P^T)_{ij}\,\chi_j = \pi_i + \sum_{j\in\mathcal{S}} p_{ji}\,\chi_j, \quad i = 1,\dots,n. \tag{10.4.5}$$

Vediamo come sia possibile applicare i concetti qui introdotti per il problema della radiosità (10.1.7).

10.4.1. Metodi di *Shooting* per la radiosità. Abbiamo già osservato in (10.3.2) che se il vettore delle radiosità della scena, $\mathbf{b}$, è soluzione di (10.1.7) allora il vettore delle potenze $\mathbf{w}$ è punto fisso della mappa $\Phi_{F^T} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, ovvero risulta

$$\mathbf{w} = \Phi_{F^T}(\mathbf{w}) = \mathbf{w}^{(e)} + D(\boldsymbol{\rho})F^T\mathbf{w}.$$

Indichiamo con $\mathbf{z} = \frac{\mathbf{w}}{\|\mathbf{w}^{(e)}\|_1}$ e con $\mathbf{z}^{(e)} = \frac{\mathbf{w}^{(e)}}{\|\mathbf{w}^{(e)}\|_1}$, osserviamo che, di conseguenza, $\|\mathbf{z}^{(e)}\|_1 = 1$ e che $\mathbf{z}$ è soluzione del sistema

$$\mathbf{z} = \mathbf{z}^{(e)} + (FD(\boldsymbol{\rho}))^T\,\mathbf{z} \Longrightarrow z_i = z_i^{(e)} + \rho_i \sum_{j=1}^{n} F_{ji} z_j\,. \tag{10.4.6}$$

Pertanto ogni singola componente z_i di $\mathbf{z}$ è soluzione di un equazione del tipo (10.4.5) e può essere pensata come la distribuzione di probabilità (10.4.4) associata ad una catena di Markov. Se generiamo con probabilità $z_i^{(e)}$ dei raggi che partono dalla sorgente i e, facendo uso del metodo delle *local lines*, delle probabilità di transizione $p_{ij} = \rho_j F_{ij}$, possono essere utilizzati i seguenti tre metodi di stima per le potenze emesse w_i, $i = 1,\dots,n$:

(1) (*Survival*) Facendo uso della probabilità di transizione p_{ij} viene fatto un test di "accettazione" per ogni altra patch della scena usando per ogni $j = 1,\dots,N$, ρ_j come probabilità di sporavvivenza. Vengono simulate h variabili aleatorie, costruendole in questo modo, e si registra il numero γ_i di volte che viene visitata la patch i-esima. Allora si può stimare w_i facendo uso della seguente

$$w_i \approx \frac{\gamma_i}{h} \cdot \left\|\mathbf{w}^{(e)}\right\|_1$$

(2) (*Collition*) Il metodo *survival* implica che vengano sparati dei raggi anche se poi non sempre questi saranno usati per il calcolo di $\mathbf{w}$, perché potrebbero non superare il test di sopravvivenza (per esempio succede di frequente se ρ_j è molto piccolo). Il metodo *collision* prevede di contare sempre ogni patch che viene colpita da un raggio (senza il test di accettazione), si ottiene così la stima seguente per w_i

$$w_i \approx \rho_i \frac{\tilde{\gamma}_i}{h} \cdot \left\|\mathbf{w}^{(e)}\right\|_1$$

dove $\tilde{\gamma}_i$ indica il numero totale di raggi che colpiscono i, il cui valore atteso è $\tilde{\gamma}_i\rho_i \approx \gamma_i$.

(3) (*Absorption*) Quest'ultimo metodo tiene conto solo dei raggi che vengono assorbiti da i (quelli che con *survival* sono scartati), risulta di conseguenza

$$w_i \approx \frac{\bar{\gamma}_i}{h} \cdot \frac{\rho_i}{1-\rho_i} \cdot \left\|\mathbf{w}^{(e)}\right\|_1$$

dove $\bar{\gamma}_i$ è il numero di raggi assorbiti da i, tale che $\tilde{\gamma}_i = \gamma_i + \bar{\gamma}_i$, $(1-\rho_i)\tilde{\gamma}_i \approx \bar{\gamma}_i$

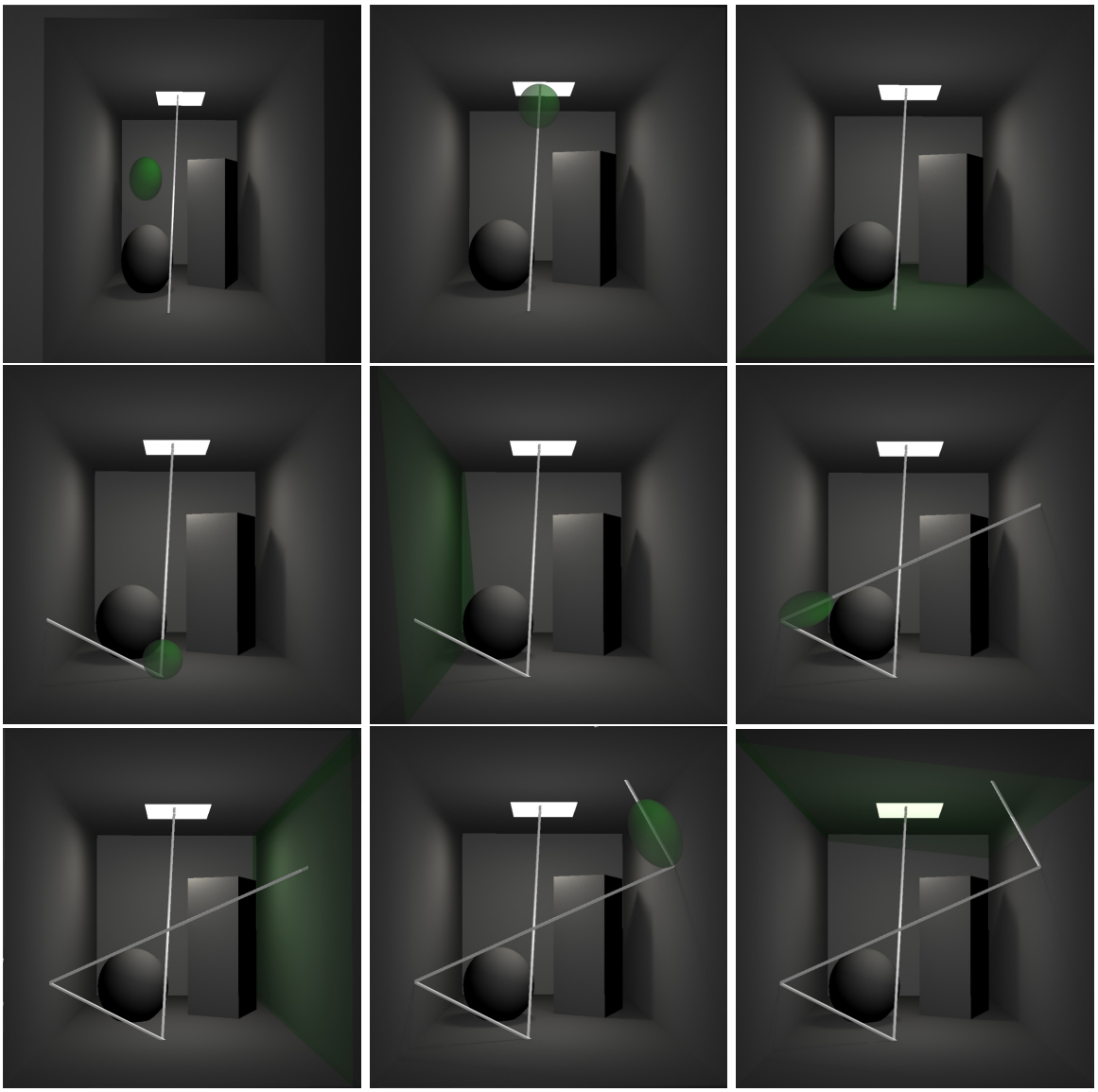


FIGURA 10.4.1. Simulazione del trasporto di un fotone. Selezionata una posizione iniziale, seguiamo il raggio fino a quando non viene assorbito.

10.4.1.1. *Rilassamento stocastico o Random walk?* Si può osservare che entrambe le tre tipologie poc'anzi presentate (*Survival*, *Collision*, *Absorption*) danno luogo ad una varianza approssimativamente pari a

$$\frac{\mathrm{Var}(\mathbf{b_{rw}})}{h_{\mathbf{rw}}} \approx \frac{\rho_k}{h_{\mathbf{rw}}\, A_k} \left\| \mathbf{w}^{(e)} \right\|_1 \left(\mathbf{b}_k - \mathbf{b}_k^{(e)} \right). \tag{10.4.7}$$

Dove $\mathbf{b_{rw}}$ è la radiosità approssimata con tali metodi ed $h_{\mathbf{rw}}$ è il numero di variabili aleatorie che vengono simulate con metodi di *local lines* (o, analogamente, con linee globali) per emulare il processo di Markov. Una stima analoga si può dare per i metodi di rilassamento stocastico (in particolare per quello di Jacobi, presentato nei dettagli nella sezione 10.3.1). Facendo uso della formula (10.3.7) risulta che la varianza della radiosità stimata stocasticamente è

$$\frac{\mathrm{Var}(\mathbf{b_{js}})}{m_{\mathbf{js}}} \approx \frac{\rho_k}{m_{\mathbf{js}}\, A_k} \left\| \mathbf{w} \right\|_1 \left(\mathbf{b}_k - \mathbf{b}_k^{(e)} \right). \tag{10.4.8}$$

Anche in questo caso intendiamo con $\mathrm{Var}(\mathbf{b_{js}})$ la varianza della radiosità ottenuta e con $m_{\mathbf{js}}$ il numero di campioni usati (indipendentemente dalle patches) per l'estimatore (10.3.7). Si può mostrare che per simulare un processo di Markov con $h_{\mathbf{rw}}$ variabili sono necessari, mediamente, $m = h_{\mathbf{rw}} \frac{\|\mathbf{w}\|_1}{\|\mathbf{w}^{(e)}\|_1}$ raggi. Pertanto se sostituiamo tale espressione in (10.4.8), ovvero poniamo $m_{\mathbf{js}} = m$, si ottiene

$$\frac{\mathrm{Var}(\mathbf{b_{js}})}{m_{\mathbf{js}}} \approx \frac{\mathrm{Var}(\mathbf{b_{rw}})}{h_{\mathbf{rw}}} \tag{10.4.9}$$

ovvero che, per lo stesso numero di raggi, **i metodi di stima con catene di Markov discrete e con rilassamento stocastico** (di Jacobi, in particolare), **sono approssimativamente efficienti in egual misura**.

10.5. Metodi di stima di densità di fotoni

L'idea di questi metodi è quella di risolvere stocasticamente (tramite processi di Markov con spazio degli stati continuo, anziché discreto) l'equazione integrale della radiosità (10.1.3), piuttosto che il sistema lineare che la approssima (10.1.5). Il beneficio maggiore che apporta questo tipo di approccio è la possibilità di tener conto della luce emessa non diffusivamente. Ricondurremo il problema del calcolo della radiosità ad un problema di *statistica* del tutto equivalente, presentando alcuni possibili approcci per la sua risoluzione. Teniamo, infine, a sottolineare che tali metodi acquistano maggiore importanza in un ottica generale, in quanto aprono la strada per ulteriori, più recenti, tipi di rappresentazione dell'illuminazione della scena (differenti dall'idea della radiosità), più sofisiticati ed allo stesso tempo più realistici (come ad esempio la mappa fotonica).

Cominciamo descrivendo un procedimento per la simulazione stocastica della traiettoria dei fotoni, in accordo con le leggi fisiche cui soggiacciono. Sia X_0 una variabile aleatoria con cui indichiamo il punto della scena scelto inizialmente. Costruiamo tale variabile con densità proporzionale alla radiosità emessa

$$p_{X_0}(\mathbf{x}) = \frac{b^{(e)}(\mathbf{x})}{\left\|\mathbf{w}^{(e)}\right\|_1}, \tag{10.5.1}$$

detta *birth density*. Successivamente si sceglie una direzione aleatoria Θ_0, dando rilevanza proporzionale al coseno dell'angolo con la normale in X_0. Dunque si pone la densità di $\Theta_0 \mid X_0$ pari ad

$$p_{(\Theta_0|X_0)}(\boldsymbol{\theta}, \mathbf{x}) = \frac{1}{\pi} \langle \boldsymbol{\theta}, \mathbf{n_x} \rangle . \tag{10.5.2}$$

Se con X_1 indichiamo la variabile aleatoria che descrive un secondo punto della scena che viene raggiunto da un raggio sparato da X_0 in direzione Θ_0, allora $(X_1 \mid X_0, \Theta_0)$ dipende da come la superficie su cui il raggio atterra è orientata (rispetto alla direzione Θ_0), quanto il punto dista da X_0 e dalla mutua visibilità tra i due. Pertanto

$$p_{(X_1|X_0,\Theta_0)}(\mathbf{x}_1, \mathbf{x}_0, \boldsymbol{\theta}) = \frac{\langle -\boldsymbol{\theta}, \mathbf{n}(\mathbf{x}_1) \rangle}{\|\mathbf{x}_0 - \mathbf{x}_1\|_2^2} V(\mathbf{x}_0, \mathbf{x}_1), \tag{10.5.3}$$

dove $V(\mathbf{x}, \mathbf{y})$ è il fattore di visibilità fra $\mathbf{x}$ ed $\mathbf{y}$ (si noti che sia $\boldsymbol{\theta}$ che $\mathbf{x}$ sono vettori di due componenti). Osserviamo che, facendo uso della proprietà $\mathbf{P}(A, B) = \mathbf{P}(A \mid B)\mathbf{P}(B)$, si ottiene la formula

$$\mathbf{P}(C \mid A, B)\mathbf{P}(A, B) = \mathbf{P}(C, B, A) = \mathbf{P}(C, B)\mathbf{P}(A)$$

da cui, dividendo per $\mathbf{P}(A, B)$,

$$\mathbf{P}(C \mid A, B) = \frac{\mathbf{P}(C, B)}{\mathbf{P}(B \mid A)}.$$

Pertanto la densità della variabile aleatoria $(X_1 \mid \Theta_0)$ risulta essere

$$p_{(X_1|\Theta_0)}(\mathbf{x}_1, \mathbf{x}_0) = p_{(X_1|X_0,\Theta_0)}(\mathbf{x}_1, \mathbf{x}_0, \boldsymbol{\theta}) p_{(\Theta_0|X_0)}(\boldsymbol{\theta}, \mathbf{x}) = \mathcal{K}(\mathbf{x}_0, \mathbf{x}_1), \tag{10.5.4}$$

dove ricordiamo che

$$\mathcal{K}(\mathbf{x},\mathbf{y}) = G(\mathbf{x},\mathbf{y})V(\mathbf{x},\mathbf{y}) = \frac{\langle v(\mathbf{x},\mathbf{y}), \mathbf{n}_{\mathbf{x}}\rangle \langle -v(\mathbf{x},\mathbf{y}), \mathbf{n}(\mathbf{y})\rangle}{\pi \|\mathbf{x}-\mathbf{y}\|_2^2} V(\mathbf{x},\mathbf{y}),$$

essendo $v(\mathbf{x},\mathbf{y})$ il versore della direzione da $\mathbf{x}$ a $\mathbf{y}$. Successivamente viene fatto un test di "sopravvivenza" sul punto X_1 così ottenuto. Bisogna decidere se il raggio viene assorbito o riflesso. Indichiamo con $\sigma(X_1)$ la probabilità che avvenga una riflessione, scegliamo tale densità pari alla frazione (albedo) di luce $\rho(X_1, -\Theta_0)$ che, provenendo dalla direzione di X_0, viene riflessa in X_1. Se la superficie di X_1 è diffusiva l'albedo coincide con il coefficiente di riflettività $f_r(X_1) = \rho(X_1)$, pertanto la probabilità di transizione da X_0 ad X_1 è

$$p_{(X_1|X_0)} = p_{(X_1|\Theta_0)} \cdot \rho(X_1) = \mathcal{K}(\mathbf{x}_0,\mathbf{x}_1)\rho(\mathbf{x}_1). \tag{10.5.5}$$

Si procede iterando questo schema per alcuni altri punti della scena. Consideriamo il valore atteso χ_X di raggi che si addensano in una zona in prossimità di X, al seguito di tale simulazione, per unità di area. Tale densità consiste di due contributi: la densità di probabilità S_X di raggi che sono inizialmente fatti partire da X e la densità di raggi che raggiungono nuovamente X dopo aver visitato altri punti Y della scena . Tale valore soddisfa l'equazione integrale

$$\chi_X(\mathbf{x}) = S_X(\mathbf{x}) + \int_\Sigma \chi_Y(\mathbf{y}) p_{(X|Y)}(\mathbf{x},\mathbf{y}) dA(\mathbf{y}),$$

in particolare per lo schema diffusivo che abbiamo considerato, risulta

$$\chi_X(\mathbf{x}) = \frac{b^{(e)}(\mathbf{x})}{\left\|\mathbf{w}^{(e)}\right\|_1} + \int_\Sigma \chi_Y(\mathbf{y})\mathcal{K}(\mathbf{y},\mathbf{x})\rho(\mathbf{x})dA(\mathbf{y}),$$

pertanto da (10.1.3) segue che

$$\chi_X(\mathbf{x}) = \frac{b(\mathbf{x})}{\left\|\mathbf{w}^{(e)}\right\|_1}. \tag{10.5.6}$$

Abbiamo ottenuto, dunque, che il numero medio di raggi che per unità di area colpiscono la superficie in prossimità di X è direttamente proporzionale alla radiosità $b(\mathbf{x})$ in tale punto. **Il problema della radiosità è stato ricondotto al problema di stimare la densità di raggi che colpiscono un dato punto x della scena .**

Il problema di stimare una tale densità è stato ampiamente studiato in statistica. Ci proponiamo di concludere presentando l'idea alla base di alcuni dei principali metodi di stima statistica per tale quantità .

Il metodo dell'istogramma. Questo è il metodo, probabilmente, più usato per la stima di densità . L'idea alla base è piuttosto semplice (e questo è il suo pregio principale e ciò che lo rende molto popolare): la scena viene suddivisa in N elementi $E_1, \ldots, E_N$ di dimensione ridotta A_i (nel nostro caso le patches) e vengono contati il numero n_i di raggi generati, che cade in ogni elemento, per $i = 1, \ldots, N$. Il rapporto $\frac{n_i}{A_i}$ risulta essere un approssimazione della densità di raggi che colpisce ogni elemento, dunque ogni punto $\mathbf{x} \in E_i$.

Il metodo delle basi ortogonali. Tale metodo si basa, come il precedente, nel suddividere la scena in N elementi $\{E_i\}_{i=1,\ldots,N}$. L'idea è quella di approssimare il valore della radiosità $b_i(\mathbf{x})$ della patch i-esima, in un sottospazio di Hilbert finito dimensionale di L^2, in modo da generalizzare il metodo precedente, ottenendo un approssimazione migliore. Consideriamo il sottospazio

$$\mathbb{V} = \text{Span}\{\phi_1, \phi_2, \ldots, \phi_\nu\} \subset L^2(\Sigma) \tag{10.5.7}$$

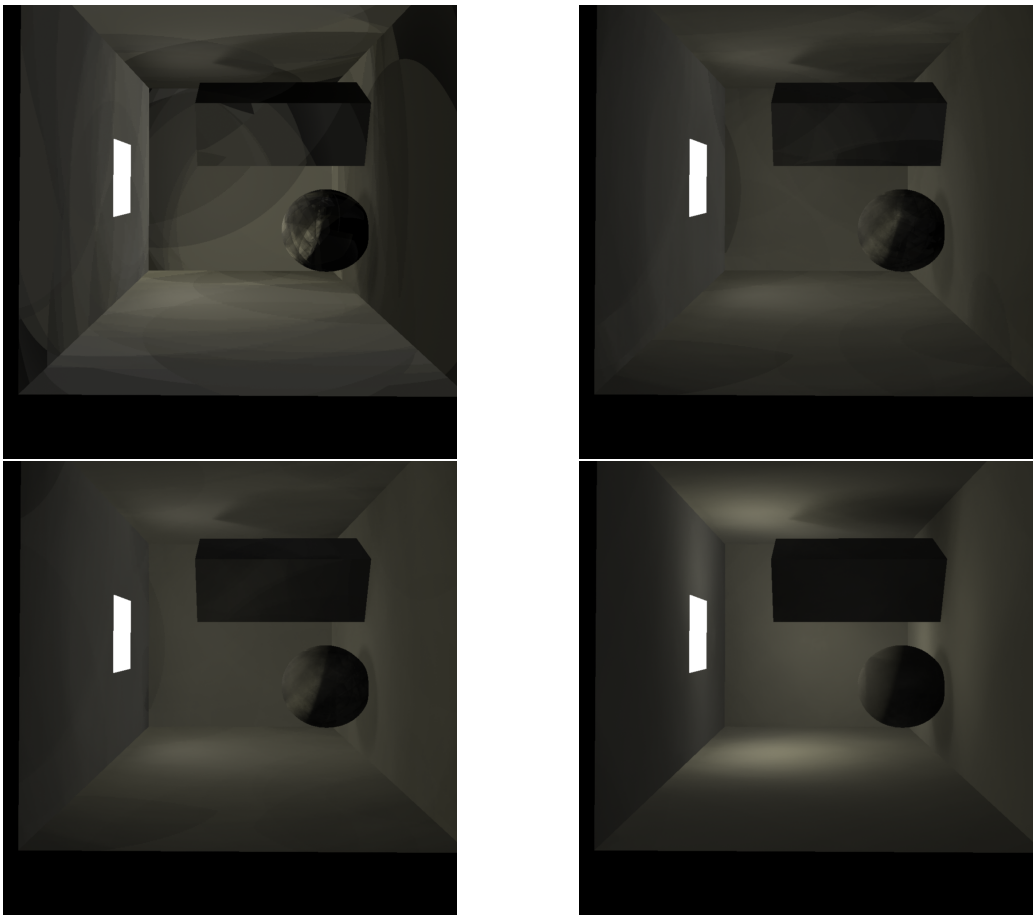


FIGURA 10.5.1. Le immagini mostrano la stessa scena *renderizzata* con densità di raggi per patch pari a 100, 500, 1000, 10000, rispettivamente.

essendo $\{\phi_1, \phi_2, \ldots, \phi_\nu\}$ un insieme di funzioni linearmente indipendenti di L^2. Data $f \in L^2(\Sigma)$, indichiamo con $f_{\mathbb{V}}$ la funzione di $\mathbb{V}$ che meglio approssima f, ovvero tale che

$$\|f - f_{\mathbb{V}}\|_{L^2(\Sigma)} = \min_{v \in \mathbb{V}} \|f - v\|_{L^2(\Sigma)} .$$

Indichiamo con $\boldsymbol{\alpha}_f$ il vettore dei coefficienti di $f_{\mathbb{V}}$, ovvero

$$f_{\mathbb{V}}(\mathbf{x}) = \sum_{i=1}^{\nu} (\boldsymbol{\alpha}_f)_i \, \phi_i(\mathbf{x})$$

Sia $\langle \cdot, \cdot \rangle$ il prodotto scalare $L^2(\Sigma)$, dal noto teorema di proiezione di Hilbert segue che $f_{\mathbb{V}}$ è l'unica funzione di $\mathbb{V}$ per cui risulti

$$\langle f_{\mathbb{V}} - f, v \rangle = 0 \quad \forall v \in \mathbb{V} \Longleftrightarrow \sum_{j=1}^{\nu} (\boldsymbol{\alpha}_f)_j \, \langle \phi_i, \phi_j \rangle = \langle f, \phi_i \rangle \quad \forall 1 \leqslant i \leqslant \nu$$

ovvero si possono caratterizzare i coefficienti di $f_{\mathbb{V}}$ come la soluzione di un sistema lineare di ν equazioni in ν incognite. Si osserva facilmente che se $\{\phi_i\}_{i=1}^{\nu}$ è una base ortogonale per $\mathbb{V}$, allora si

può dare la seguente formula esplicita per l'approssimante $f_{\mathbb{V}}$

$$f_{\mathbb{V}}(\mathbf{x}) = \sum_{j=1}^{\nu} \frac{\langle f, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j(\mathbf{x}). \tag{10.5.8}$$

L'idea del metodo che qui discutiamo è quella di scegliere opportuni sottospazi $\mathbb{V}$, considerare opportune basi ortogonali per tali sottospazi ed approssimare la radiosità su ogni elemento considerando $b_{\mathbb{V}}(\mathbf{x})$ al posto di $b(\mathbf{x})$. La scelta tipica per il sottospazio $\mathbb{V}$ è quella dei polinomi di grado minore o uguale a $k > 0$, che indichiamo con $\mathbb{P}_k$, oppure, molto in voga di recente, quella delle B-Spline (quasi sempre con nodi equi spaziati), di grado massimo $k > 0$, che indichiamo con $\mathbb{S}_k$. Si noti, ad esempio, che se si sceglie come base del sottospazio $\mathbb{V} = \mathbb{S}_1$ l'insieme delle funzioni caratteristiche dei domini E_i (ovvero $\phi_i = \chi_{E_i}$), allora si riottiene il metodo dell'istogramma.

Il metodo di *Nearest Neighbor*. Questo metodo prende spunto da un idea di Jensen e dall'algoritmo di *Photon Mapping* che egli ha sviluppato. L'idea è simile a quella del metodo dell'istogramma, tuttavia si lavora "cambiando punto di vista": invece di fissare un certo elemento E_i e contare il numero di raggi n_i che lo colpiscono, si fissa un numero n e si cerca un elemento che abbia ricevuto un tale numero di raggi. Se un tale numero n viene trovato su un elemento E_i di area piccola, allora la densità $n/A(E_i)$ sarà elevata, e viceversa. Questo tipo di approccio permette di limitare la quantità di memoria da occupare, infatti necessita solo di tener memoria dei punti colpiti dai raggi.

Parte 3

Sviluppo di un render fotorealistico di Illuminazione Globale

I contenuti sono stati ottenuti nella tesi di laurea di Federico Forti [**14**]

CAPITOLO 11

Algoritmi per il rendering fotorealistico

Il renderer fotorealistico che d'ora in poi viene spiegato in questo libro è stato sviluppato in [**14**] (per renderers ancora più sofisticati si veda [**33**]). Il codice, interamente presentato e commentato in questo e nei prossimi tre Capitoli, con spiegazioni sugli aspetti matematici ed algoritmici, è scritto in C++ (scelta piú abituale per applicativi che si propongono di gestire finestre di rendering e di interfaccia), ma per semplicità viene usato un sottoinsieme delle strutture di C++ sostanzialmente identico al linguaggio C (in effetti, quasi tutte le classi, con modifiche banali, possono essere esguite in C). Una versione differente del codice, scritta usando tutte le strutture tipiche di C++, ma algoritmicamente più limitata (ad esempio perché non utilizza procedure di rendering della radiosità, Photon Mapping e Final Gathering) è presentata nel Capitolo 18.

Nel nostro programma ci sono diverse possibili scelte per il rendering di una scena: le scelte vengono determinate da *flag*[1] inseriti nel «`main.h`». Bisogna decidere tra uno solo di essi ed impostarlo su `True`, assicurandosi che tutti gli altri siano impostati su `False`.

```
<<main.h>>
```

```
/* elenco dei flag utilizzati per il rendering.
! solamente uno di loro può essere impostato su true, a tutti gli altri bisogna
   assegnare false ! */
// \‘{e} attivo il raytracing stocastico?
bool stoc=true;
//raytracing stocastico nella sola formulazione emisferica?
bool stocE=false;
//\‘{e} attivo il metodo di Jacobi stocastico?
bool jacob=false;
//\‘{e} attivo il photon mapping?
bool photonMap=false;
//\‘{e} attivo il final gathering?
bool Fg= false;
//\‘{e} attivo il photon mapping multipass?
bool multiPassPhotonMap=false;
```

Il calcolo dell'illuminazione inizia con una fase di ray tracing, nella quale per ogni pixel del sensore viene creato un raggio dall'osservatore al pixel. Intersecando questo raggio con gli oggetti della scena otteniamo il punto $\mathbf{x}$ in cui dobbiamo effettuare il calcolo. Questo valore sarà poi assegnato al pixel per la creazione dell'immagine finale. Di seguito sono presentati gli algoritmi usati per la risoluzione dell'equazione del rendering.

[1] parametri che possono essere solamente `True` o `False`.

11.1. Codice per il Ray Tracing stocastico

Abbiamo visto nel Capitolo 9 che il Ray tracing stocastico (o path tracing) scompone la radianza in tre componenti: radianza emessa, radianza proveniente direttamente dalle luci e radianza indiretta, proveniente dalle interriflessioni nella scena. La prima tra queste è ricavata direttamente dal valore L_e del materiale, e di solito supera di molto il range di luminositàdell'immagine, ma, come tutti i valori di radianza che vengono calcolati esso sarà, alla fine del processo viene rinormalizzato al range $[0, 1]$.

```
<<srt.cpp>>
```

```
//radianza emessa dall'oggetto o in direzione r:
float3 Le(Ray& r,Obj* o){
    //carico l'indice del material
    int mId=o->matId;
    //il valore restituito \'{e} uguale al valore Le del materiale
    return material[mId].Le;
}
```

Le altre due componenti sono state rispettivamente analizzate nelle Sezioni 9.4 e 9.5. Esse vengono sommate in modo da ottenere la risoluzione dell'equazione del rendering. Questo algoritmo permette di ottenere una rappresentazione fedele della scena, anche per materiali con BRDF particolari, purtroppo però le immagini hanno sempre un elevato grado di rumore (quindi un alto valore di varianza), sopratutto nelle zone in ombra dove l'illuminazione indiretta diventa importante: infatti sono pochi i campioni che continuano le riflessioni multiple poiché non sono assorbiti nel corso della Roulette Russa. Se si utilizza la formulazione emisferica anche per materiali con BRDF particolari, e non l'importance sampling, si rischia, inoltre, di aumentare il livello del rumore, in quanto verrebbero considerati anche i campioni che danno poco o nessun contributo al rendering.

```
<<srt.cpp>>
```

```
//Stochastic Ray Tracing:
float3 radianceSTOC(Ray &viewRay,Obj* o,int x, int y, int& n){
    float3 radianceOutput;
    //illuminazione generata:
    radianceOutput=radianceOutput+Le(viewRay,o);
    //illuminazione diretta:
   radianceOutput=radianceOutput+directIllumination(viewRay,o,x,y);
    //illuminazione indiretta:
    //si controlla se \'{e} stato superato il massimo numero dei raggi che possono
        essere riflessi
   if(n<MAX_DEPTH){
   radianceOutput=radianceOutput+indirectIllumination(viewRay,o,x,y,n); }
    return radianceOutput;
}
```

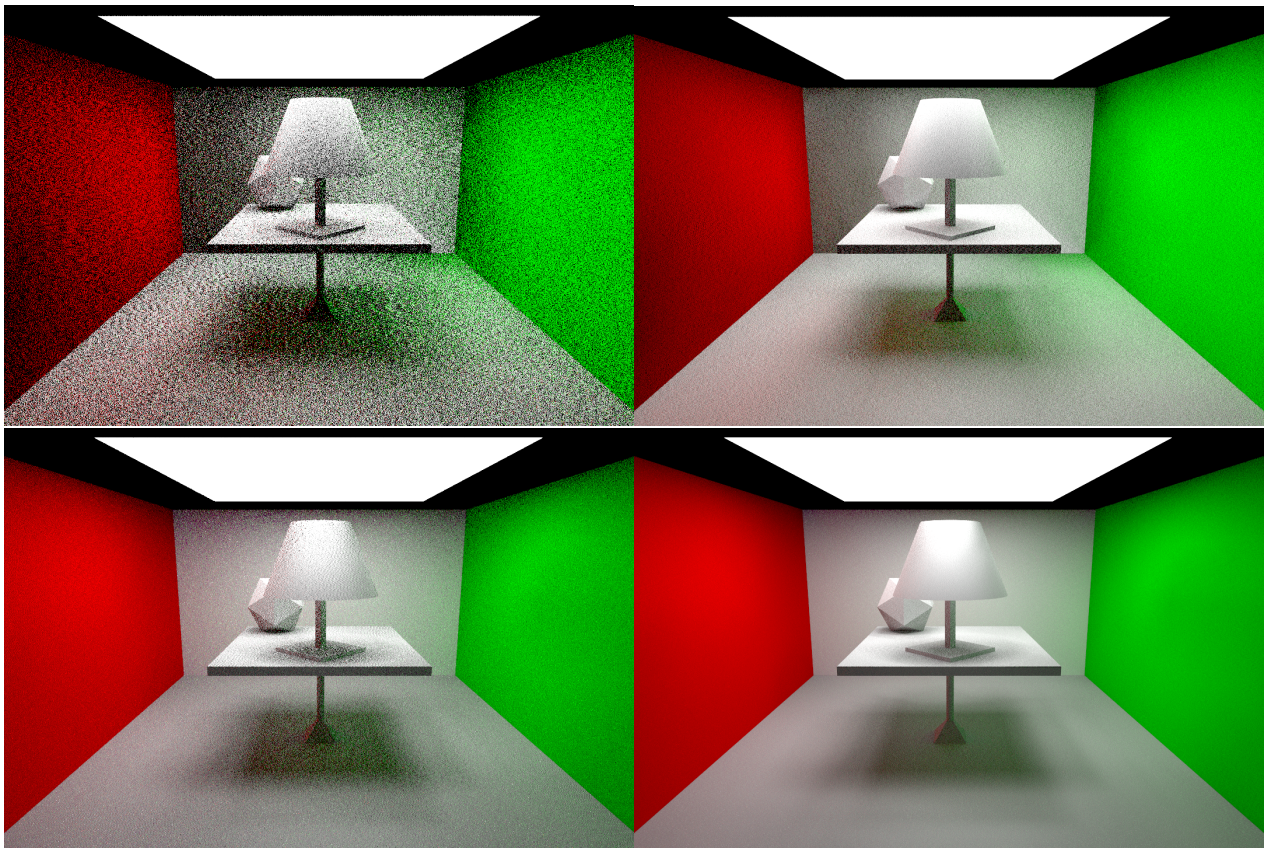


FIGURA 11.1.1. **Confronto tra formulazione emisferica e formulazione ad area**: nella prima riga il ray tracing stocastico è stato effettuato utilizzando la formulazione emisferica, nella seconda riga invece è stata divisa la parte di illuminazione indiretta da quella diretta e per quest'ultima parte è stata utilizzata la formulazione ad area. Infine nella colonna a sinistra si sono utilizzati 10 sample mentre nella colonna a destra 100.

Utilizzando questo metodo non si riesce a riprodurre il fenomeno delle *caustiche*[2], in quanto non si possono creare dei raggi d'ombra che attraversino l'oggetto, siano rifratti e raggiungano la luce. Si potrebbe utilizzare la sola formulazione emisferica dell'equazione del rendering, ma in questo caso diventerebbe ancora più difficile eliminare il rumore, poiché le *caustiche* producono piccole zone ad alto contrasto che hanno l'effetto di aumentare la varianza.

[2]in questo fenomeno i raggi riflessi o rifratti provenienti da un oggetto speculare convergono e si concentrano su una superficie diffusiva.

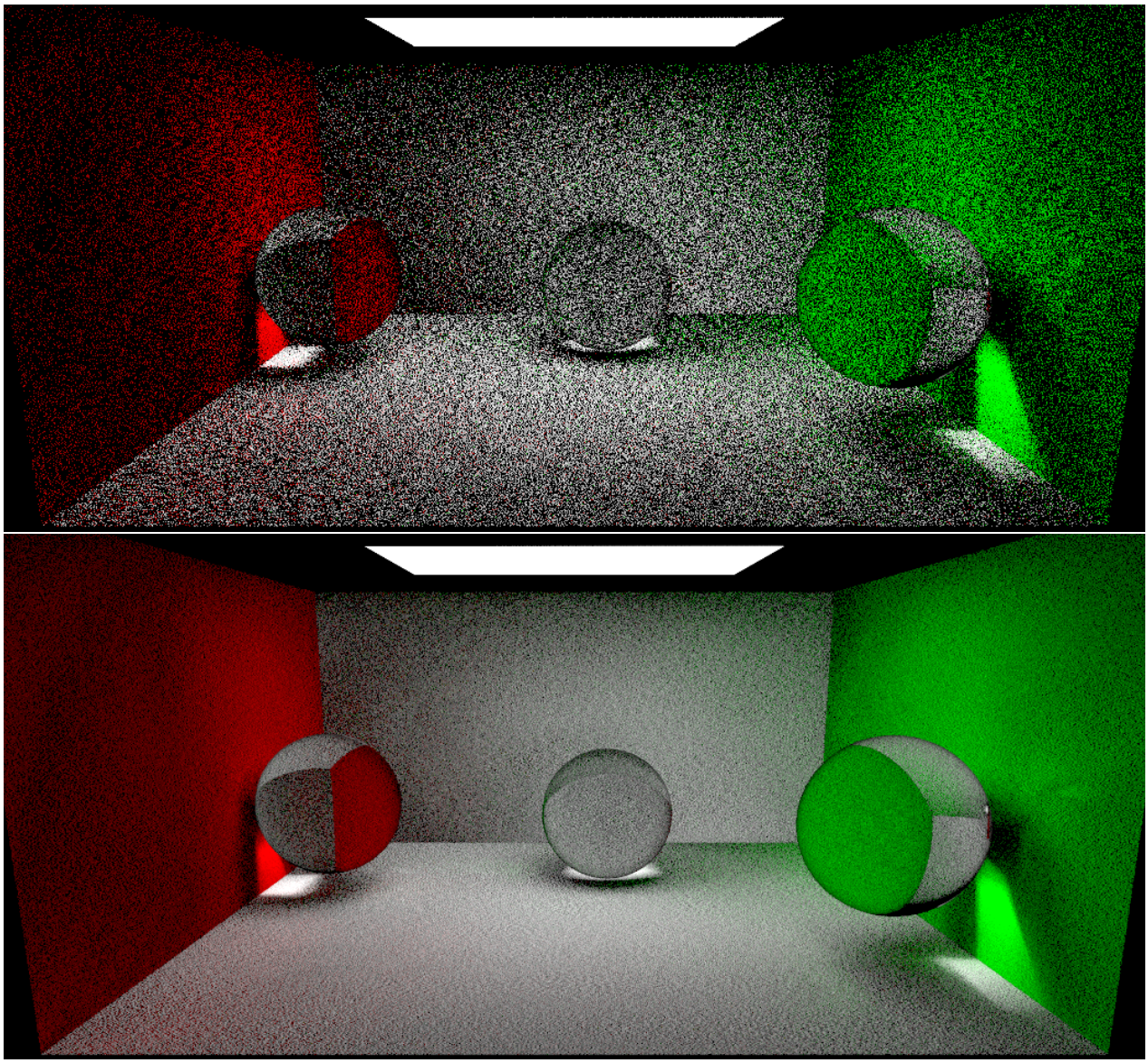


FIGURA 11.1.2. **Caustiche tramite il ray tracing stocastico nella formulazione solo emisferica**: Il materiale utilizzato per le sfere ha indice di rifrazione (1.55), ed è stato impostato $K_r = (0, 0, 0)$ e $K_g = (1, 1, 1)$. In alto sono stati usati 10 sample mentre in basso 100.

11.2. Codice per scene diffusive: metodi di rilassamento stocastici, calcolo tramite Monte Carlo dei fattori di forma

11.2.1. Radiosità. Ci concentriamo ora su scene i cui elementi hanno una BRDF diffusiva, ovvero costante rispetto all'angolo di uscita. Grazie a questa condizione iniziale possiamo limitarci ad osservare la *radiosità* della scena piuttosto che la *radianza* (si veda il Capitolo 10, particolarmente la Sezione 10.1). L'analisi è stata già presentata nel Capitolo 10. Qui presentiamo il codice relativo al metodo di Jacobi incrementale.

11.2.2. Metodo di Jacobi stocastico incrementale (Stochastic Incremental Shooting of Power). Per prima cosa, rammentiamo qui il metodo di Jacobi stocastico incrementale, spiegato

nella Sottosezione 10.3.1.

Il metodo che implementiamo riguarda il sistema delle potenze, ovvero

$$MP = P_e, \quad (M)_{ij} = \delta_{ij} - \rho_i F_{ji} \qquad i = 1, \ldots, n,\ j = 1, \ldots, n$$

Vogliamo però considerare, anziché la potenza P, solamente la potenza residua ΔP, che deve essere ancora propagata nella scena. Possiamo infatti ricavare l'una dall'altra sommando tutti i contributi $\Delta P^{(l)}$ ottenuti nelle k iterazioni del processo

$$\Delta P_i^{(0)} = P_{ei} \tag{11.2.1}$$

$$\Delta P_i^{(k+1)} = \rho_i \sum_{j=1}^{n} \Delta P_j^{(k)} F_{ji} \tag{11.2.2}$$

$$P_i^{(k)} = \sum_{l=0}^{k} \Delta P_i^{(l)}. \tag{11.2.3}$$

Trasformiamo ora la (11.2.2) in una doppia sommatoria che potremo stimare stocasticamente scegliendo opportunamente le patch j e l. Per far questo utilizziamo la delta di Kronecker $\delta_{li} = 1$ se $l = i$ e 0 negli altri casi.

$$\Delta P_i^{(k+1)} = \sum_{j=1,\, l=1}^{n} \Delta P_j^k \, F_{jl} \, \rho_l \, \delta_{li} \tag{11.2.4}$$

Si scelgono quindi le due patch (j, l) nel seguente modo:

(1) Si seleziona una "sorgente" (la patch j) con probabilità proporzionale alla potenza che deve essere ancora rilasciata:

$$p_j = \frac{\Delta P_j^{(k)}}{\Delta P_{\mathbb{T}}^{(k)}} \tag{11.2.5}$$

con $P_{\mathbb{T}}^{(k)} = \sum_{j=1}^{n} \Delta P_j^{(k)}$.

(2) Si seleziona una "destinazione" (la patch l) con probabilità condizionale

$$p_{l|j} = F_{jl}\,, \tag{11.2.6}$$

utilizzando il *local line sampling*. Si crea quindi un raggio dalla patch i distribuito uniformemente sull'emisfero frontale alla patch. La probabilità che questo raggio colpisca la patch l è esattamente uguale al suo fattore di forma.

Unendo le due probabilità in (11.2.5) e (11.2.6) otteniamo la probabilità congiunta:

$$p_{jl} = \frac{\Delta P_j^{(k)} F_{jl}}{\Delta P_{\mathbb{T}}^{(k)}}$$

Otteniamo quindi N coppie $(i, j)_p$ con $p = 1, \ldots, N$ con cui possiamo costruire un estimatore di Monte Carlo per la doppia sommatoria (11.2.4). Useremo qui per semplicità $(i, j) = (i, j)_p$

$$< \Delta P_i^{(k+1)} >_N = \frac{1}{N} \sum_{p=1}^{N} \frac{\Delta P_j^k \, F_{jl} \, \rho_l \, \delta_{li}}{p_{jl}}$$

$$= \frac{1}{N} \sum_{p=1}^{N} \Delta P_{\mathbb{T}}^k \, \rho_l \, \delta_{li}$$

Questa stima deve essere effettuata per ogni lunghezza d'onda RGB, di conseguenza il numero dei campioni utilizzato per ciascuna di esse viene preliminarmente suddiviso in base alla potenza rimanente; quindi sono utilizzati più campioni per le lunghezze d'onda con maggiore potenza residua.

La scelta della patch l, su cui rilasciare l'energia, avviene in base al fattore di forma F_{jl}, che ci dà informazioni su quanto angolo solido copre la patch l rispetto alla patch j, così facendo si ha maggiore rumore sulle patch con area più piccola. Queste patch, infatti, hanno meno probabilità di essere scelte nel processo, in quanto coprono un angolo solido minore rispetto alle patch più grandi. Nel programma viene data la possibilità di aumentare il numero di triangoli con cui è composta la scena tramite il parametro `sceneDepth`, esso rappresenta il numero di iterazioni svolte nella suddivisione dei triangoli. Ad ogni iterazione ogni triangolo della scena viene diviso in due triangoli più piccoli, viene quindi cercato il punto intermedio nel lato più lungo del triangolo e tracciata una linea di divisione per i due nuovi triangoli (si veda il Capitolo 13). Qui presentiamo il codice relativo al metodo di Jacobi incrementale:

<<srt.cpp>>

```
//metodo incrementale stocastico di Jacobi :
void JacobiStoc(Obj** objects,int nObj){

    //potenza residua totale:
    float3 Prtot=float3(0);
    //potenza di ogni patch della scena:
    float3* P=new float3[nObj];
    //potenza residua di ogni patch della scena:
    float3* Pr=new float3[nObj];
    //numero di step raggiunti dal processo:
    int steps=0;
    //stima dell'errore raggiunto dal processo:
    float err=0;
    Obj** objX=new Obj*();

    //vengono caricati i valori iniziali di Luminosità Emessa per ogni patch della
        scena (Pe)
    for(int i=0;i<nObj;i++){
        //viene caricata l'area dell'oggetto
        float area=objects[i]->areaObj;
        //se l'area \‘{e} più piccola della precisione di calcolo allora impostiamo
            l'area a 0
        area=area>EPS?area:0;

        //potenza emessa dalla patch i
        float3 LP=material[objects[i]->matId].Le*M_PIf*area;
        //viene calcolata la potenza totale iniziale
        Prtot=Prtot+LP;
       //viene salvata la potenza dell'elemento i
        P[i].copy(LP);
        //viene salvata la potenza residua dell'elemento i
        Pr[i].copy(LP);
        //viene calcolato l'errore di approssimazione con cui \‘{e} possibile
            fermare il metodo
        err=err+pow(LP.media(),2);
    }
```

```
//viene calcolata la radice così da ottenere l'errore
err=sqrtf(err);
//iterazioni del metodo di Jacobi:

//Si continuano le iterazioni finché l'energia rilasciata non diventa minore di
   un certo valore (in base alla variabile err) o gli steps superano gli steps
   massimi (in base alla variabile maxsteps)
while((err>maxerr)&&(steps<maxsteps)){

    //viene inizializzato un seme casuale
    unsigned int s=round(rand());

    //si inizializza per ognuna delle tre componenti RGB (che chiamiamo qui
       (x,y,z)), una variabile per il numero di sample utilizzati nella patch e
       una per il numero di sample utilizzati fino ad ora.
    int NprevX=0;
    int NX=0;
    int NprevY=0;
    int NY=0;
    int NprevZ=0;
    int NZ=0;

    //potenza residua totale nelle tre componenti:
    float Prt= Prtot.x+Prtot.y+Prtot.z;
    //sample distribuiti in base alla potenza totale di ciascuna componente RGB:
    float3 samps=float3(Jacobisamps*Prtot.x/Prt,
    Jacobisamps*Prtot.y/Prt,Jacobisamps*Prtot.z/Prt);

    float3 q=0;
    //per ogni patch della scena
    for(int i=0;i<nObj;i++){
Obj* o=objects[i];
        //probabilità con cui viene scelta la patch i:
        float3 pi=(Pr[i].div(Prtot));
        //parametro che ci permette di utilizzare tutti i sample che erano
           stati previsti:
        q=q+pi;

       //componente rossa:

        //si calcola il numero di samples utilizzati per la patch i in base
           alla sua potenza (e in base a quanti sample sono già stati usati)
        NX=round((q.x*samps.x)+NprevX*(-1));
        //per ogni samples dell'elemento i
        for (int j=0; j<NX; j++){
            float3 dir;
            float rndX=0.0f;
            float rndY=0.0f;
            float rndZ=0;
```

```
//scelta randomica
if(aST==SOB){
    rndX=generateRandom(aoSId[0],3,aST);
    rndY=generateRandom(aoSId[0],4,aST);
    rndZ=generateRandom(aoSId[0],4,aST);
}
else{
    rndX=generateRandom(s,j+1,aST);
    rndY=generateRandom(s,j+1,aST);
    rndZ=generateRandom(s,j+1,aST);
}

    //si utilizza l'Inverse Cumulative Distribuction Function del
       coseno, così da non dover calcolare i fattori di forma
                    float rndPhi=2*M_PIf*(rndX);
    float rndTeta=acosf(sqrtf(rndY));

    //punto scelto uniformemente nella patch i
    float3 rndPoint= o->randomPoint(rndX, rndY,rndZ);

    // creazione della base ortonormale
    float3 u,v,w;
    w=o->normal(rndPoint);
    float3 up(0.0015f,1.0f,0.021f);
    v=w%up;
    v.norm();
    u=v%w;
    dir=u*(cos(rndPhi)*sinf(rndTeta))
    +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
    dir.norm();

    //si crea il raggio per scegliere la patch j con probabilità
       uguale al fattore di forma tra la patch i e quella j
    Ray ffRay(rndPoint,dir);
    float t=inf;
    *objX=NULL;
   //l'oggetto j intersecato \'{e} la patch su cui sarà rilasciata
      la potenza totale della componente rossa
    if(intersect(ffRay,t,objX))
    {
        //si salva la potenza rilasciata all'interno della struttura
           dell'oggetto
        //essa si sommerà con la potenza residua parziale che
           l'oggetto ha raggiunto finora
        //solo alla fine del processo infatti si avrà la potenza
           residua totale della patch
        (*objX)->P.x=(*objX)->
        P.x+material[(*objX)->matId].Kd.x
```

```
                *M_PIf*(Prtot.x)/((float)samps.x);
            }
        }
    //si aggiornano il numero di sample usati per questa componente
    NprevX=NprevX+NX;

//componente verde: (stesso procedimento della componente rossa)

NY=round((q.y*samps.y)+NprevY*(-1));
for (int j=0; j<NY; j++){
    float3 dir;
    float rndX=0.0f;
    float rndY=0.0f;
    float rndZ=0;

    if(aST==SOB){
        rndX=generateRandom(aoSId[0],5,aST);
        rndY=generateRandom(aoSId[0],6,aST);
        rndZ=generateRandom(aoSId[0],7,aST);
    }
    else{
        rndX=generateRandom(s,j+1,aST);
        rndY=generateRandom(s,j+1,aST);
        rndZ=generateRandom(s,j+1,aST);
    }

    float rndPhi=2*M_PIf*(rndX);
    float rndTeta=acosf(sqrtf(rndY));
    //punto scelto uniformemente nella patch i:
    float3 rndPoint= o->randomPoint(rndX, rndY,rndZ);

    float3 u,v,w;
    w=o->normal(rndPoint);
    float3 up(0.0015f,1.0f,0.021f);
    v=w%up;
    v.norm();
    u=v%w;

    dir=u*(cos(rndPhi)*sinf(rndTeta))
    +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
    dir.norm();

    //si crea il raggio per scegliere la patch j con probabilità uguale al
        fattore di forma tra la patch i e quella j
    Ray ffRay(rndPoint,dir);
    float t=inf;
    *objX=NULL;

    if(intersect(ffRay,t,objX)){
```

```
            (*objX)->P.y=(*objX)->P.y+material[(*objX)->matId].Kd.y
            *M_PIf*(Prtot.y)/((float)samps.y);

        }
    }
    NprevY=NprevY+NY;

    //componente blu:(stesso procedimento della componente rossa)

        NZ=round((q.z*samps.z)+NprevZ*(-1));

        //per ogni samples sull'elemento i
        for (int j=0; j<NZ; j++){

            float3 dir;
            float rndX=0.0f;
            float rndY=0.0f;
            float rndZ=0;

            if(aST==SOB){
                rndX=generateRandom(aoSId[0],7,aST);
                rndY=generateRandom(aoSId[0],8,aST);
                rndZ=generateRandom(aoSId[0],9,aST);
            }
            else{
                rndX=generateRandom(s,j+1,aST);
                rndY=generateRandom(s,j+1,aST);
                rndZ=generateRandom(s,j+1,aST);
            }

            float rndPhi=2*M_PIf*(rndX);
            float rndTeta=acosf(sqrtf(rndY));

            float3 rndPoint= o->randomPoint(rndX, rndY,rndZ);

            float3 u,v,w;
            w=o->normal(rndPoint);
            float3 up(0.0015f,1.0f,0.021f);
            v=w%up;
            v.norm();
            u=v%w;
            dir=u*(cos(rndPhi)*sinf(rndTeta))
            +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
            dir.norm();
            Ray ffRay(rndPoint,dir);
            float t=inf;
            *objX=NULL;
```

```
                if(intersect(ffRay,t,objX)){
                    (*objX)->P.z=(*objX)->P.z+material[(*objX)->matId].Kd.z
                    *M_PIf*(Prtot.z)/((float)samps.z);
                }
            }
            NprevZ=NprevZ+NZ;
        }

    //una volta terminata la fase di shooting si aggiornano le componenti di
        Potenza residua, Pr, Potenza P, e si azzerano le potenze residue parziali
        che erano state salvate negli oggetti
   //azzeramento della potenza totale:
    Prtot=float3(0);

    for(int i=0;i<nObj;i++)
    {
      //aggiornamento delle Potenze (vengono aggiunte le potenze residue totali
          immagazzinate dalle patch durante il processo)
      P[i]=P[i]+objects[i]->P;
       //aggiornamento delle potenze residue totali
        Pr[i].copy(objects[i]->P);
        //calcolo dell'errore
        err=err+pow(Pr[i].media(),2);
        //calcolo dell'energia residua totale
        Prtot=Prtot+Pr[i];
        //azzeramento della potenza residua parziale contenuta nella patch i
        objects[i]->P.copy(float3(0));
    }
    err=sqrtf(err);
    //viene aumentato il numero di step
    steps++;
}
//finite le iterazioni:

//I valori ottenuti vengono salvati su ciascuna patch nella variabile P (in cui
    durante il processo veniva salvata la potenza residua parziale)
    for(int i=0; i<nObj; i++)
    {
       objects[i]->P.copy(P[i]);
    }
 //si libera la memoria allocata
    delete[] Pr;
    delete[] P;
    delete objX;
}
```

Per visualizzare il risultato dividiamo per l'area della patch e per π, ricavandoci così la radianza nel punto osservato. Nel programma non è stato implementato nessun metodo di interpolazione dei risultati, si visualizza, perciò, un colore costante su ogni patch della scena.

```
<<srt.cpp>>
<<radiance>>
```

```
        //metodo di Jacobi stocastico:
        if(jacob){
        //calcolo della radiosità nel punto osservato
            float3 L=(*o)->P*(1/(*o)->areaObj);
            return L+radianceRefr+radianceRefl;
       }
```

11.3. Metodi multi-pass: Final Gathering e Photon map

I metodi che qui illustreremo sono degli algoritmi ibridi, che utilizzano al meglio gli elementi del ray tracing stocastico e della radianza stocastica che abbiamo visto nelle sezioni precedenti. Questi metodi sono suddivisi in step di ordine prefissato; i primi forniscono delle informazioni preliminari sulla scena, che consentono ai successivi di ottenere il risultato in modo più efficiente. Entrambi i metodi qui presentati hanno però delle limitazioni: il primo presenta delle difficoltà nel rendere scene con BRDF e geometrie particolari; il secondo, pur non avendo questo limite, fornisce un risultato viziato (*biased*), seppur consistente.

11.3.1. Final Gathering. Questo metodo si basa sull'utilizzo di una soluzione precalcolata di radiosità, tramite la quale evitare la ricorsività del ray tracing stocastico (sezione 11.1). La resa della scena viene divisa in due passi:

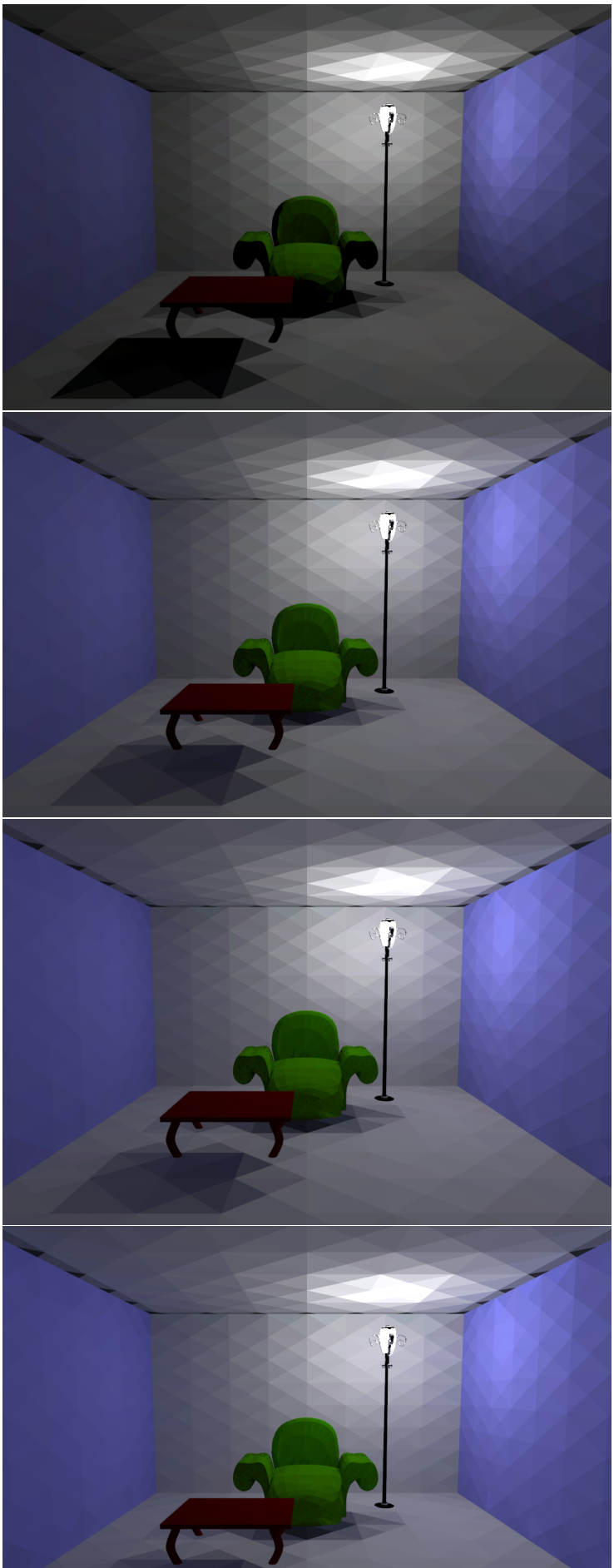
(1) calcolare la radiosità della scena;
(2) utilizzare il ray tracing stocastico, ma servendosi del risultato dello step precedente per il calcolo dell'illuminazione indiretta, quindi evitando il processo ricorsivo che altrimenti l'illuminazione indiretta richiederebbe. La soluzione di radiosità dello step precedente viene pensata come una distribuzione di luce sulla scena: quindi tutte le parti della scena diventano equivalenti a lampade la cui illuminazione è data dalla radiosità precalcolata, ed il ray tracing stochastico si limite a raccogliere i contributi della loro illuminazione diretta.

Più in dettaglio: la parte ricorsiva dell'equazione del rendering è l'illuminazione indiretta; per fermare questo processo, nel ray tracing stocastico, si utilizza la Roulette Russa, responsabile dell'elevata varianza del metodo. Il Final Gathering, invece, si serve di una sola iterazione del ray tracing stocastico, nella quale si raccolgono (*gathering*) le informazioni ottenute dalla soluzione precalcolata di radiosità, contenente le interriflessioni della scena. Per utilizzare il Final Gathering nel programma bisogna impostare i parametri `Fg` e `jacob` su TRUE. Nel programma è inoltre possibile utilizzare il Final Gathering con il photon mapping (cfr. 11.3.2), al posto del metodo di Jacobi e quindi del calcolo della radiosità.

```
<<srt.cpp>>
```

```
float3 FinalGathering(Ray &viewRay,int x,int y,Obj* o){

    float3 radianceOutput=float3(0);
    //illuminazione generata
    radianceOutput=radianceOutput+Le(viewRay,o);
    //illuminazione diretta
```



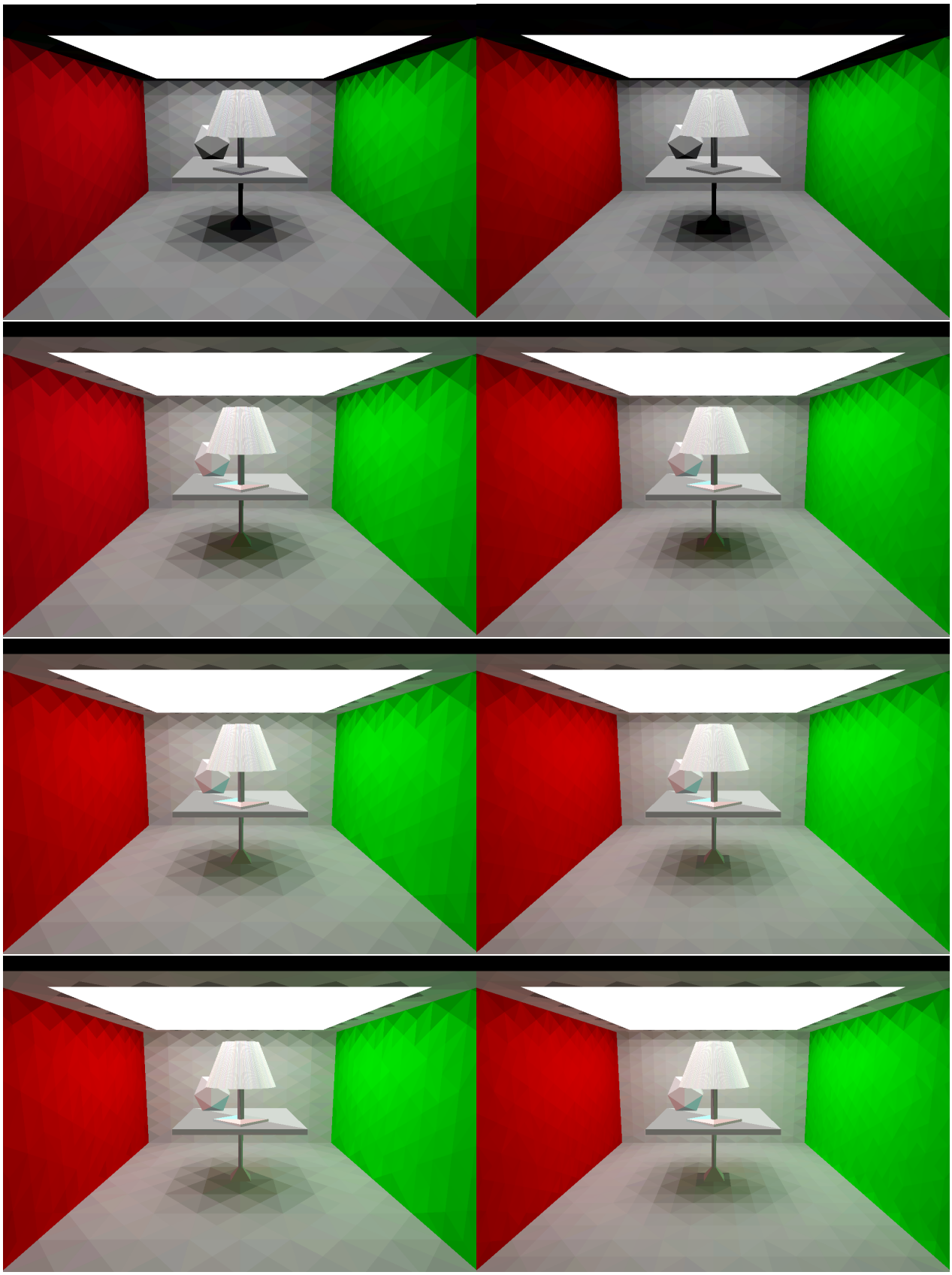


FIGURA 11.2.2. **Stochastic Incremental Shooting of Power**: ogni riga rappresenta un iterazione del metodo di Jacobi, nella colonna di sinistra sono stati utilizzati 5000000 campioni e un valore di `SceneDepth`= 7, nella colonna di destra sono stati utilizzati 15000000 campioni e un valore di `SceneDepth`= 8.

```
    radianceOutput=radianceOutput+directIllumination(viewRay,o,x,y);
    //illuminazione indiretta
    radianceOutput=radianceOutput+FinalIndirect(viewRay,o,x,y);
    return radianceOutput;

}
```

<<srt.cpp>>

```
//funzione che si occupa di calcolare l'illuminazione indiretta nel Final
    Gathering:
float3 FinalIndirect(Ray &r,Obj* o,int x,int y){

    //valore che viene restituito alla fine del processo
    float3 radianceOutput=float3(0);
    //si carica la normale normale dell'oggetto
    float3 n1=o->normal(r.o);
    //si carica l'identificativo del materiale
    int mId=o->matId;

    //per ogni sample di illuminazione indiretta:
    for(int s=0; s<aosamps; s++){
        //inizializzazione variabili:
        float3 dir;
        float rndX=0.0f;
        float rndY=0.0f;

        //si genera un numero uniformemente distribuito tra 0 e 1
        if(aST==SOB){
            rndX=generateRandom(aoSId[0],3,aST);
            rndY=generateRandom(aoSId[0],4,aST);
        }
        else{
            rndX=generateRandom(aoSamplesX[x+y*w],s+1,aST);
            rndY=generateRandom(aoSamplesY[x+y*w],s+1,aST);
        }
        //si utilizza la distribuzione rispetto al coseno di deviazione con la
            normale:
        float rndPhi=2*M_PIf*(rndX);
        float rndTeta=acosf(sqrtf(rndY));

        // creazione base ortonormale:
        float3 u,v,w;
        w=n1;
        float3 up(0.0015f,1.0f,0.021f);
        v=w%up;
        v.norm();
        u=v%w;
```

```
        //direzione del raggio:
        dir=u*(cos(rndPhi)*sinf(rndTeta))
        +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
        dir.norm();

        //creazione del raggio di illuminazione indiretta:
        Ray reflRay(r.o,dir);
        float t=inf;
        Obj** objX =new Obj*();
        *objX=NULL;
        // il metodo và avanti solo se si interseca un oggetto e se questo oggetto
            non \‘{e} una luce ( poich\‘{e} la luminosità diretta l’abbiamo già
            considerata):
        if((intersect(reflRay,t,objX))
        &&(material[(*objX)->matId].Le.max()==0)){
            //se \‘{e} stato utilizzato il metodo di Jacobi
            if(jacob){
            //si calcola la radianza nel punto incontrato
            float _area=1/(*objX)->area();
            radianceOutput=radianceOutput+((*objX)->P)
            .mult(material[mId].Kd)*_area*M_1_PIf;
            }
            //se \‘{e} stato utilizzato il photon mapping
            if(photonMap){
                //calcolo del punto di intersezione:
                float3 iP=reflRay.o+reflRay.d*t;
                //creazione del raggio di entrata nel punto:
                Ray r2(iP,reflRay.d*(-1));
                //calcolo della BRDF:
                float3 BRDF= material[mId].C_T_BRDF(reflRay,r,n1);
                //calcolo della radianza attraverso il photon mapping:
                radianceOutput=radianceOutput
                +photonRadiance(r2,*objX,KdTree,photond_2,nPhotonSearch)
                .mult(BRDF)*M_PIf;
            }
        }
        delete objX;
    }
    //si divide in base al numero di campioni che sono stati utilizzati
    radianceOutput=radianceOutput/aosamps;
    return radianceOutput;
}
```

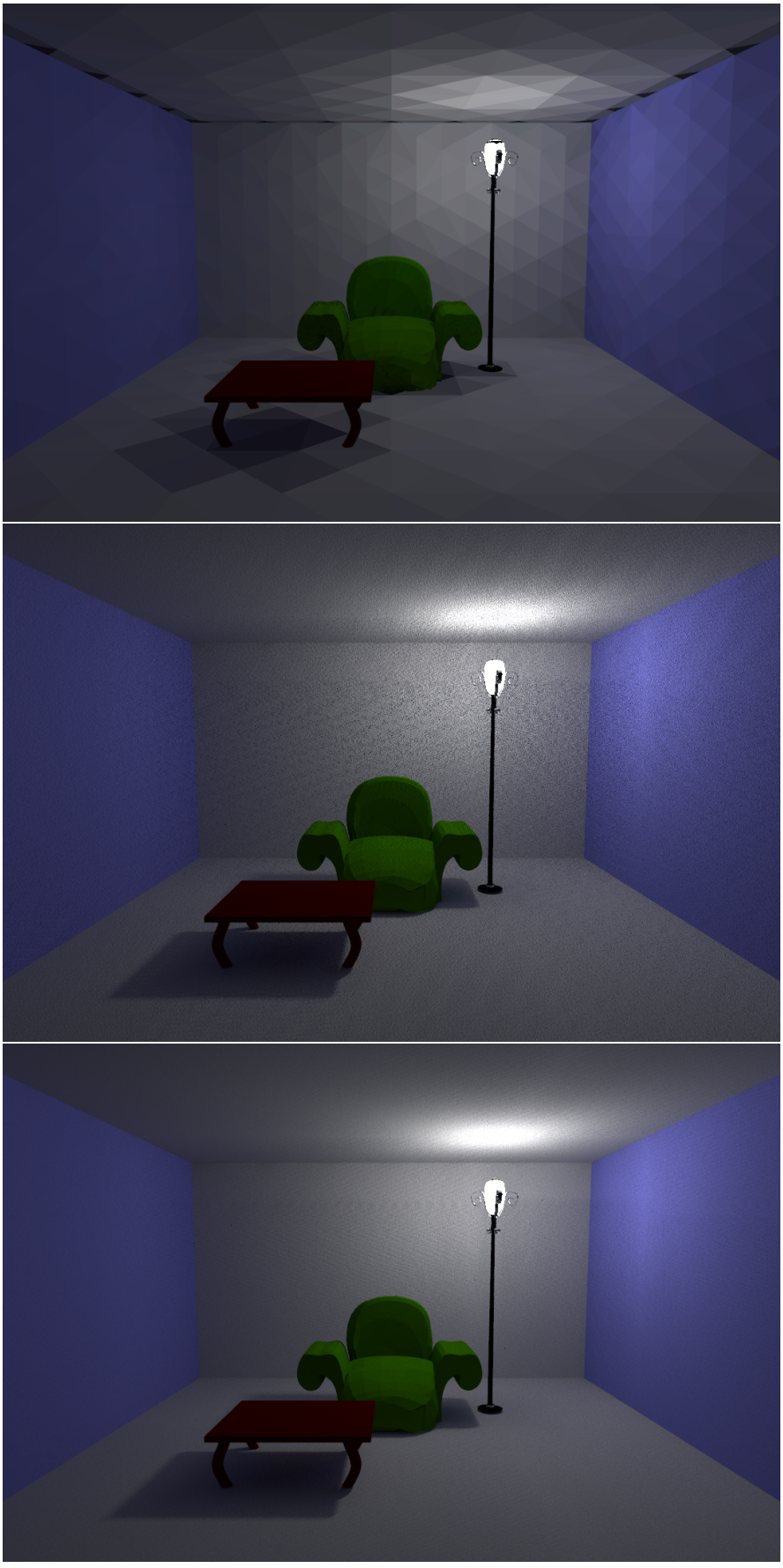


FIGURA 11.3.1. **Final Gathering**: la seconda e la terza immagine sono state create a partire dalla soluzione di radiosità visualizzata nella prima immagine (15m 35sec), nella quale

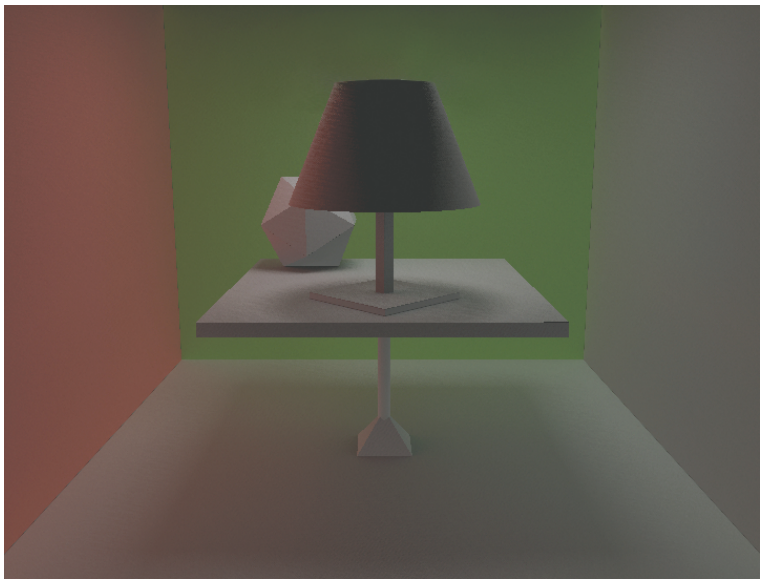


FIGURA 11.3.2. **Color bleeding nel Final Gathering**: in questa immagine si visualizza il mescolamento dei colori dovuto alla vicinanza tra le superficie.

11.3.2. Photon mapping. Lo scopo di questo algoritmo è quello di ottenere in breve tempo immagini realistiche, con poco rumore, di scene costruite con qualsiasi tipo di materiale e con qualsiasi tipo di geometria: in particolare, il metodo è indipendente dalla modellazione a maglie. Questo metodo basa inoltre i suoi calcoli su percorsi che partono direttamente dalla fonte di luce, come avviene nella realtà fisica, e non più solo dall'osservatore. Qui ora è cambiato il metodo di calcolo della soluzione precalcolata: non più adirezionale come la radiosità, ma fotorealistica grazie al seguire i raggi di luce. In particolare, ne segue la possibilità e l'opportunità di campionare percorsi diversi della luce a densità di campionamento (ovvero numero di campioni) differenti a seconda delle esigenze di contrasto e di riduzione del rumore. Queste caratteristiche costituiscono la differenza rispetto ai metodi ad elementi finiti come l'algoritmo di Radiosity, il quale, a confronto, presenta i seguenti svantaggi:

- è utilizzabile solo su superfici diffusive (infatti una delle sue semplificazioni è proprio considerare la BRDF costante).
- produce artefatti nell'illuminazione, perché essa viene considerata costante all'interno di ciascuna singola patch

Il photon mapping si basa sul presupposto che è raro trovare rapide variazioni di illuminazione in larghe parti della scena. Di conseguenza si approssima la luce che arriva in un punto con la luce che arriva nei punti adiacenti ad esso. A questo scopo, però, occorre trattare separatamente i percorsi della luce che provocano forti e rapide variazioni del contrasto di illuminazione, ossia le situazioni di luce concentrata su piccole aree diffusive come effetto di collimazione dovuta a lenti o specchi concavi (*caustiche*). Quindi i percorsi di luce che generano le caustiche vengono trattati ex novo in una delle fasi del metodo, che segue i fotoni a partire dalle sorgenti di luce e si accorge di come essi si concentrino su aree diffusive dopo aver attraversato lenti o specchi (*caustic map*). Possiamo quindi suddividere l'algoritmo in tre passi:

(1) **emissione dei "fotoni" dalle fonti di luci.** Questi "fotoni", che trasportano una parte del flusso della fonte originaria, si propagano e rimbalzano in tutta la scena, lasciando parte della loro energia ogni volta che incontrano una superficie diffusiva o glossy. Sono quindi

salvati all'interno di una mappa (la *photon map*) che conserva la posizione, la potenza e la direzione di provenienza del fotone. Se nella scena sono presenti superfici speculari saranno inoltre emessi dei "fotoni" verso di esse così da costituire la *caustic map*.

(2) la **photon map** viene utilizzata per stimare la radianza in ogni punto della scena attraverso la stima della densità di "fotoni".

(3) la **caustic map** viene utilizzata per stimare la radianza nei punti della scena che contengono caustiche.

NOTA 11.3.1.
- *Nella photon map la varianza si identifica con rumore in modo diverso da quello visto finora: questo metodo infatti trasforma un errore ad alta frequenza*[3]*, il rumore, in un errore a bassa frequenza* [**24**]*. Infatti, una varianza insufficiente è dovuta ad un numero troppo esiguo di fotoni, i quali quindi, quando vengono assorbiti, depositano la propria energia in punti troppo radi e sparsi. Pertanto la stima della densità di energia, ossia la media per area su dischi piccoli del flusso dei fotoni assorbiti, produce chiazze di luce.*
- *Se da una parte il photon map ci permette di creare immagini fotorealistiche in tempi molto brevi, dall'altra introduce un errore di bias. Infatti, certi modi di trasporto della luce sono contati più di una volta, nalla photon map completa e poi nella mappa delle caustiche. Il metodo è però consistente: aumentando il numero di fotoni l'errore tende a scomparire (ma di conseguenza aumenta anche il tempo di esecuzione).*

Per questo motivo il photon mapping viene raramente utilizzato in modo diretto, ma piuttosto attraverso il Final Gathering. Si divide quindi l'illuminazione diretta, la quale presenta forti variazioni tra le zone in ombra e quelle in luce, da quella indiretta, nella quale tramite raggi distribuiti uniformemente lungo un emisfero si raccolgono le informazioni presenti nella *photon map*. Nel programma offre inoltre un'altra possibilità [`?pharr`], in cui il calcolo dell'illuminazione diretta avviene tramite ray tracing stocastico mentre quello dell'illuminazione indiretta avviene tramite lettura diretta della photon map. A questo scopo la photon map deve essere creata in modo da non contenere i fotoni di illuminazione diretta ma solo quelli creati attraverso riflessioni successive nella scena. Il metodo in questione viene chiamato col nome di `multiPassPhotonMap` ed è attivabile impostando il parametro booleano corrispondente. Vediamo ora più nel dettaglio come si struttura il photon mapping.

11.3.2.1. *Passo 1: tracciamento dei fotoni.* Nel primo passo i "fotoni" originati dalle fonti di illuminazione sono propagati nella scena attraverso raggi, così come avviene nel ray tracing. Utilizzando la tecnica della Roulette russa si decide se essi sono riflessi, trasmessi o assorbiti. Ogni qual volta un fotone si deposita su una superficie non speculare esso viene conservato all'interno della photon map. In base al materiale incontrato viene associata al fotone una probabilità di riflessione diffusiva, speculare, di trasmissione o assorbimento. Nei materiali completamente diffusivi ad esempio la probabilità di riflessione speculare o trasmissione è 0 mentre la probabilità di assorbimento dipende dall'indice di riflettività del materiale (un oggetto nero ha infatti probabilità di assorbimento 1 in quanto non riflette luce). L'uso di una struttura leggera (basata su punti) per la conservazione dei fotoni è la chiave dell'efficienza di questo metodo. Essa deve contenere informazioni riguardo: la potenza, la direzione di provenienza (per il calcolo della BRDF) e la posizione del fotone.

```
<<geometry.h>>
```

```
//struttura per la conservazione dei fotoni nel photon mapping:
struct photon{
```

[3] considerando la variazione di colore tra i pixel in un immagine come un segnale discreto a due dimensioni.

FIGURA 11.3.3. **visualizzazione diretta della mappa fotonica:** nell'immagine è possibile notare come il numero di fotoni usati per approssimare l'illuminazione diretta non è sufficiente; il risultato è un immagine a chiazze, le quali evidenziano i punti con un minore numero di fotoni. Se si fosse utilizzata la mappa fotonica solamente per l'illuminazione indiretta questo problema non sarebbe visibile.

```
    //posizione del fotone nella scena:
    float3 position;
    //potenza del fotone:
    float3 power;
    //direzione di provenienza del fotone:
    float3 direction;
    //costruttore:
    photon(float3 ip,float3 d, float3 p){
        position.copy(ip);
        power.copy(p);
        direction.copy(d);
    }
};
```

FIGURA 11.3.4. **visualizzazione diretta di una photon map con 1072447973 fotoni, di cui 1500000 emessi direttamente dalle luci:** nell'immagine la presenza di una BRDF di Cook e Torrance con valore di `slope` = 0.1 obbliga l'uso di molti fotoni. In questo caso, infatti, i fotoni che sono lontani dal punto osservato possono causare un grave errore, poiché si prendono in considerazione effetti di *specular* che non sarebbero presenti nella realtà.

Consideriamo dapprima l'emissione e il tracciamento di fotoni dalla sorgente di luce i, ognuno di essi trasporta una frazione della potenza totale di i, che è

$$P_e^{(i)} = L_e^{(i)} \pi \, A_l^{(i)} \, ,$$

dove con $A_l^{(i)}$ si indica l'area della luce i.
Considerando un numero n_{ph} di fotoni emessi, la potenza di ciascun fotone, emesso dalla sorgente i, diventa:

$$P_{ph}^{(i)} = \frac{P_e^{(i)} \cos\theta}{n_{ph}} \, .$$

Qui il termine $\cos\theta$ ha il compito di proiettare l'area considerata lungo la direzione di uscita del fotone θ. Ma poiché anche in questo caso utilizzeremo la distribuzione coseno (vedi capitolo 8) per l'emissione dei fotoni, abbiamo

$$P_{ph}^{(i)} = \frac{P_e^{(i)} \pi}{n_{ph}} \, .$$

dove la potenza del fotone viene moltiplicata per π, normalizzazione della distribuzione coseno dell'angolo di deviazione dalla normale. Per poter gestire scene con un numero elevato di luci, l'emissione

del fotone avviene da una luce scelta con probabilità uniforme tra tutte quelle presenti nella scena. Dividendo anche per questa PDF (discreta) otteniamo quindi:

$$P_{ph}^{(i)} = \frac{P_e^{(i)}\,\pi\,N_l}{n_{ph}}\,,$$

dove N_l è il numero di luci nella scena. Inseriamo qui il codice relativo all'emissione diffusiva dei fotoni:

```
<<srt.cpp>>
```

```
//funzione che permette l'emissione diffusiva dei fotoni dalle luci della scena.
void emitPhotons(std::vector<photon*>& photons){
    //inizializzazione:
    unsigned int rnd1=rand();
    unsigned int rnd2=rand();
    unsigned int rnd3=rand();
    unsigned int rndL=rand();
    float random1;
    float random2;
    float random3;
    float randomL;
    Obj** objX=new Obj*();
    int s=0;

    //per ogni fotone:
    while(s<nPhoton){
        //si aumenta il numero dei fotoni utilizzati
        s++;

        //generazione di un numero random uniforme in [0,nLight-1] per la scelta
            della luce:
        if(aST==SOB){
            randomL=generateRandom(aoSId[0],3,aST)*(nLight-1);
        }
        else{
            randomL= generateRandom(rndL,s+1,aST)*(nLight-1);
        }

        //il numero generato viene reso discreto e utilizzato per accedere
            all'array delle luci:
        unsigned long i= (unsigned long)floorf(randomL);

        //si carica l'area della luce
        float area= luci[i]->areaObj;

        //si caricano i dati relativi alla luce
        int lid= luci[i]->matId;
```

```
//calcoliamo la potenza trasportata da un singolo fotone (uguale per tutti
   gli nPhoton fotoni)
float3 P= material[lid].Le*M_PIf*M_PIf*area*nLight/(nPhoton);

    //si campiona la luce uniformemente lungo la sua area
    if(aST==SOB){
        random1=generateRandom(aoSId[0],5,aST);
        random2=generateRandom(aoSId[0],6,aST);
        random3=generateRandom(aoSId[0],7,aST);
    }
    else{
        random1= generateRandom(rnd1,s+1,aST);
        random2= generateRandom(rnd2,s+1,aST);
        random3=generateRandom(rnd3,s+1,aST);
    }


    // punto scelto uniformemente nella patch i;
    float3 p= luci[i]->randomPoint(random1,random2,random3);

    float rndPhi=2*M_PIf*(random1);
    float rndTeta=acosf(sqrtf(random2));

    //creazione della base ortonormale:
    float3 u,v,w;
    w=luci[i]->normal(p);
    float3 up(0.0015f,1.0f,0.021f);
    v=w%up;
    v.norm();
    u=v%w;

    float3 dir=u*(cos(rndPhi)*sinf(rndTeta))
    +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
    //raggio di emissione del fotone:
    Ray photonRay(p,dir);
    float t= inf;
    *objX=NULL;
    int n=0;
    //il raggio viene intersecato con la scena:
    if( intersect(photonRay,t,objX)){

        //punto di intersezione:
        float3 iP=photonRay.o+photonRay.d*t;

        //si crea il nuovo fotone nel punto di intersezione
        photon* p2=new photon(iP,dir*(-1),P);
        //se il fotone si \‘{e} depositato su una superficie non speculare
            allora viene salvato dentro al vettore dei fotoni
        if((!multiPassPhotonMap)&&
```

```
                ((material[(*objX)->matId].Kd.max()>0)||
                (material[(*objX)->matId].slope!=0))){
                    photons.push_back(p2);
                }
                //il fotone continua il suo percorso all'interno della scena:
                photonScatter(photons,(*objX),n,p2);
            }
    }
    delete objX;
}
```

I fotoni scontrandosi con gli oggetti diffusivi della scena sono salvati all'interno di una mappa tridimensionale (la *photon map*) dopodiché in base alla BRDF dell'oggetto e al numero di riflessioni che sono state effettuate sono assorbiti, trasmessi o riflessi. Nel programma è proposto il metodo della Roulette Russa, modificato in modo che sia l'utente a determinare quando il metodo debba fermarsi con certezza. Questo è necessario poiché il numero di riflessioni potrebbe superare la memoria del supporto sul quale il programma è stato lanciato. Nel file «`utilities.h`» viene quindi definito il valore `MAX_DEPTH_PHOTON`, da questo ricaviamo il `peso` che moltiplicato per la probabilità di assorbimento del fotone costituisce il valore da utilizzare nella Roulette Russa. Con l'aumentare delle riflessioni nella scena, n, il `peso` diminuisce e il metodo si ferma con certezza solo quando è raggiunto il numero $M =$ `MAX_DEPTH_PHOTON`.

$$\texttt{peso} = \frac{M - n}{M}$$

Sono ora assegnate tre probabilità, per la riflessione, la diffusione o la trasmissione del fotone, che ricaviamo dai rispettivi coefficienti del materiale: K_r, K_d, K_g. Poiché questi tre coefficienti sono diversi in base alla lunghezza d'onda del fotone, si prende in considerazione la media delle tre componenti RGB del coefficiente. La probabilità che il fotone continui il suo percorso è data dalla somma delle tre, bisogna quindi assicurarsi che la loro somma sia minore di uno, scalando opportunamente i coefficienti.

```
<<srt.cpp>>
```

```
//funzione per la propagazione dei fotoni nella scena
void photonScatter(std::vector<photon*>& photons,Obj* objX, int n_,photon* p){

    //si carica il numero massimo di rimbalzi previsti per un fotone
    float MAX=MAX_DEPTH_PHOTON;
    //ci si ricava il peso da utilizzare nella Roulette Russa
    float peso=(MAX-n_)/MAX;

    //si carica l'ID del materiale
    int mId= objX->matId;

    //si carica la normale dell'oggetto
    float3 n= objX->normal(p->position);

    //si ricava dai coefficenti del materiale, diffusione , rifrazione del fotone
    float P_refl= material[mId].Kr.media();
    float P_diff= material[mId].Kd.media();
```

```
float P_glass= material[mId].Kg.media();

//ci si assicura che la somma di questi valori sia minore di 1 altrimenti essi
    sono scalati
float Ptot= P_refl+P_diff+P_glass;
if(Ptot>1){
P_refl/=Ptot;
P_diff/=Ptot;
P_glass/=Ptot;
}

//si caricano le probabilità in unico array di 3 elementi moltiplicando per il
    parametro peso, il quale assicura una probabilità di assorbimento non nulla.
float P[3];
P[0]=P_refl*peso;
P[1]=P[0]+P_diff*peso;
P[2]=P[1]+P_glass*peso;

//inizializzazione dell'oggetto da colpire
Obj** objY=new Obj*();
//si crea il raggio di entrata del fotone:
Ray entryRay= Ray(p->position,p->direction);

//si genera un numero random da utilizzare per la Roulette Russa
float rnd;
if(aST==SOB){
    rnd=generateRandom(aoSId[0],3,aST);
}
else{
    rnd=rand()/RAND_MAX;
}
//metodo della Roulette russa:

//probabilità materiali riflettenti:
//(il fotone viene riflesso perfettamente)
if(rnd<P[0]){
    //si calcola la direzione di riflessione
    float3 refl= reflect(p->direction,n);
    //si crea il raggio riflesso
    Ray reflRay;
    reflRay.o=p->position;
    reflRay.d=refl;
    float t2=inf;
    //si interseca il raggio con la scena
    if(intersect(reflRay,t2,objY)){
        //aumento del numero di iterazioni:
        n_++;
        //si calcola il punto di intersezione
        float3 iP=reflRay.o+reflRay.d*t2;
```

```
            //si carica la potenza del fotone
            float3 oldP=p->power;
            //si calcola il coseno di deviazione con la normale
            float cos_i=p->direction.dot(n);
            //si calcola il coefficenti di Fresnel
            float3 Fresn=material[mId].getFresn(cos_i);
            //si carica la BRDF dei materiali riflettenti
            float3 BRDF = material[mId].S_BRDF(Fresn);
            //si calcola la potenza del nuovo fotone riflesso dividendo per la
                probabilità della Roulette Russa
            float3 newP= oldP.mult(BRDF)*(1/(P_refl*peso));
            //si crea il nuovo fotone
            photon* p2=new photon(iP,refl*(-1),newP);

            //se l'oggetto incontrato risponde al modello di Cook e Torrance allora
                il fotone viene immagazzinato nella photon map
            if((material[(*objY)->matId].Kd.max()>0)
            ||(material[(*objY)->matId].slope!=0)){
                photons.push_back(p2);
            }
            //il fotone continua il suo percorso
            photonScatter(photons,*objY,n_,p2);
        }
    }

    //probabilità materiali diffusivi:
    //(il fotone viene riflesso diffusivamente)
    if((rnd<P[1])&&(rnd>P[0])){

        float rndls;
        float rndlt;
       //generazione randomica per la scelta della direzione del nuovo fotone
          riflesso
        if(aST==SOB){
            rndls=generateRandom(aoSId[0],3,aST);
            rndlt=generateRandom(aoSId[0],4,aST);
        }
        else{
            rndls= rand()/RAND_MAX;
            rndlt= rand()/RAND_MAX;
        }

        float rndPhi=2*M_PIf*(rndls);
        float rndTeta=acosf(sqrtf(rndlt));

        //si crea una base ortonormale nel punto di intersezione
        float3 u,v,w;
        w=objX->normal(p->position);
        float3 up(0.0015f,1.0f,0.021f);
```

```
        v=w%up;
        v.norm();
        u=v%w;

        float3 dir=u*(cos(rndPhi)*sinf(rndTeta))
        +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
        dir.norm();

        //creazione del raggio riflesso
        Ray reflRay(p->position,dir);

        float t2=inf;
        if(intersect(reflRay,t2,objY)){
            n_++;
            float3 iP=reflRay.o+reflRay.d*t2;
            float3 oldP=p->power;
            //si carica la BRDF del materiale
            float3 BRDF = material[mId].C_T_BRDF(entryRay,reflRay,n);
            float3 newP= oldP.mult(BRDF)*(1/(P_diff*peso));
            photon* p2=new photon(iP,dir*(-1),newP);

            //se l'oggetto incontrato risponde al modello di Cook e Torrance allora
                il fotone viene immagazzinato nella photon map
            if((material[(*objY)->matId].Kd.max()>0)
            ||(material[(*objY)->matId].slope!=0)){
                photons.push_back(p2);
            }
            photonScatter(photons,*objY,n_,p2);
        }
    }

//probabilità materiali trasparenti:
//(il fotone viene rifratto)
if((rnd<P[2])&&(rnd>P[1])){

    //si verifica che l'indice di rifrazione sia uguale per tutte le componenti RGB
    //in questo caso il calcolo per la rifrazione sarà semplificato:
    if((material[mId].ior.x==material[mId].ior.y)
    &&(material[mId].ior.x==material[mId].ior.z)){
        //direzione del raggio rifratto:
        float3 dir;
        //calcolo della direzione per il raggio rifratto:
        if(refract(dir,p->direction,n,material[mId].ior.x)){

            //viene creato il raggio rifratto
            Ray refrRay(p->position,dir);

            float t=inf;
            Obj** objX=new Obj*();
```

```
            *objX=NULL;

            if(intersect(refrRay, t, objX)){
                n_++;
                float3 iP=refrRay.o+refrRay.d*t;
                float3 oldP=p->power;
                float cos_i=p->direction.dot(n);
                //calcolo del coefficiente di Fresnel e della BRDF per materiali
                    trasparenti
                float3 Fresn=material[mId].getFresn(cos_i);
                float3 BRDF = material[mId].T_BRDF(Fresn);
                //potenza del nuovo fotone
                float3 newP= oldP.mult(BRDF)*(1/(P_glass*peso));
                photon* p2=new photon(iP,refrRay.d*(-1),newP);

                //se l'oggetto incontrato risponde al modello di Cook e Torrance
                    allora il fotone viene immagazzinato nella photon map
                if((material[(*objY)->matId].Kd.max()>0)
                ||(material[(*objY)->matId].slope!=0)){
                    photons.push_back(p2);
                }
                photonScatter(photons,*objY,n_,p2);
            }
            delete objX;
        }
    }else{
        //se l'indice di rifrazione \‘{e} diverso per ciascuna componente RGB:
        //si carica un array di 3 raggi corrispondenti alle 3 lunghezza d'onda di
            base RGB
        Ray* refrRay= new Ray[3];
        refrRay[0].o=p->position;
        refrRay[1].o=p->position;
        refrRay[2].o=p->position;
        //calcolo delle direzioni dei raggi rifratti
        refract(refrRay,p->direction,n,material[mId].ior);

        float t2=inf;
        float3 oldP=p->power;
        n_++;
        //componente rossa:
        //si verifica che non ci sia riflessione totale interna
        if(refrRay[0].depth!=0){
                *objY=NULL;
                if(intersect(refrRay[0],t2,objY)){
                    float3 iP=refrRay[0].o+refrRay[0].d*t2;
                    float cos_i=p->direction.dot(n);
                    //calcolo del coefficiente di Fresnel e della BRDF per materiali
                        trasparenti
                    float3 Fresn=material[mId].getFresn(cos_i);
```

```
            float3 BRDF = material[mId].T_BRDF(Fresn);
            //potenza del fotone che trasporta la componente rossa
            float3 newP= float3(oldP.x*BRDF.x*(1/(P_glass*peso)),0,0);
            photon* p2=new photon(iP,refrRay[0].d*(-1),newP);

            //se l'oggetto incontrato risponde al modello di Cook e Torrance
                allora il fotone viene immagazzinato nella photon map
            if((material[(*objY)->matId].Kd.max()>0)
            ||(material[(*objY)->matId].slope!=0)){
                photons.push_back(p2);
            }
            photonScatter(photons,*objY,n_,p2);
        }
    }
//componente verde:
if(refrRay[1].depth!=0){

        t2=inf;
        *objY=NULL;
        if(intersect(refrRay[1],t2,objY)){

            float3 iP=refrRay[1].o+refrRay[2].d*t2;
            float cos_i=p->direction.dot(n);
            //calcolo del coefficiente di Fresnel e della BRDF per materiali
                trasparenti:
            float3 Fresn=material[mId].getFresn(cos_i);
            float3 BRDF = material[mId].T_BRDF(Fresn);

            //potenza del fotone che trasporta la componente verde:
            float3 newP= float3(0,oldP.y*BRDF.y*(1/(P_glass*peso)),0);
            photon* p2=new photon(iP,refrRay[1].d*(-1),newP);

            //se l'oggetto incontrato risponde al modello di Cook e Torrance
                allora il fotone viene immagazzinato nella photon map
            if((material[(*objY)->matId].Kd.max()>0)
            ||(material[(*objY)->matId].slope!=0)){
                photons.push_back(p2);
            }
            photonScatter(photons,*objY,n_,p2);
        }
    }
//componente blu:
if(refrRay[2].depth!=0){

        t2=inf;
        *objY=NULL;
        if(intersect(refrRay[2],t2,objY)){

            float3 iP=refrRay[2].o+refrRay[2].d*t2;
```

```
                    float cos_i=p->direction.dot(n);
                    //calcolo del coefficiente di Fresnel e della BRDF per materiali
                        trasparenti:
                    float3 Fresn=material[mId].getFresn(cos_i);
                    float3 BRDF = material[mId].T_BRDF(Fresn);
                    //potenza del fotone che trasporta la componente blu:
                    float3 newP= float3(0,0,oldP.z*BRDF.z*(1/(P_glass*peso)));
                    photon* p2=new photon(iP,refrRay[2].d*(-1),newP);

                    //se l'oggetto incontrato risponde al modello di Cook e Torrance
                        allora il fotone viene immagazzinato nella photon map
                    if((material[(*objY)->matId].Kd.max()>0)
                    ||(material[(*objY)->matId].slope!=0)){
                        photons.push_back(p2);
                    }
                    photonScatter(photons,*objY,n_,p2);
                }
            }
        delete[] refrRay;
    }
}
   delete objY;
}
```

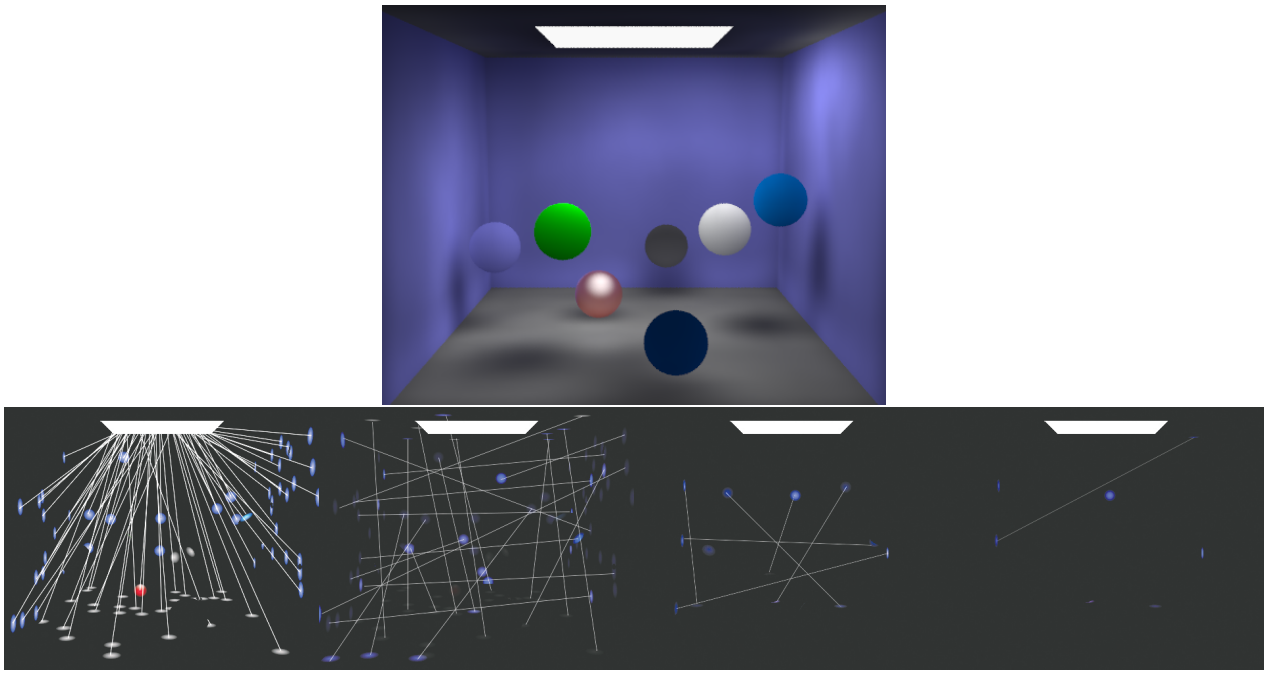


FIGURA 11.3.5. **photon scattering**: sono qui mostrate 4 riflessioni in una scena con 100 fotoni emessi dalla sorgente di luce. Il parametro `MAX_DEPTH_PHOTON` è pari a 1000000000. In alto mostriamo la scena attraverso il photon mapping utilizzando 60000 fotoni e un accuratezza pari a 0.7 .

11.3.2.2. *Passo 2: stima della radianza.* Ogni fotone della *photon map* indica che la zona in cui si trova sta ricevendo un flusso, $d\Phi$, proveniente da un'illuminazione diretta o indiretta. Con un solo fotone però non possiamo stabilire niente a proposito dell'illuminazione, per questo motivo dobbiamo affidarci alla stima della *densità di fotoni*, $d\Phi/d\sigma$. La quantità che ci interessa è infatti la radianza riflessa L_r, che è definita in (9.3.1) come:

$$L_r(x \to \mathbf{\Theta}) = \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) L(x \leftarrow \mathbf{\Psi}) \langle N_x, \mathbf{\Psi} \rangle \, d\omega(\mathbf{\Psi})$$

Per calcolare questo integrale abbiamo bisogno di informazioni riguardo alla radianza entrante $L(x \leftarrow \mathbf{\Psi})$, ma la photon map ci dà solo informazioni riguardo al flusso entrante. Dobbiamo quindi utilizzare la relazione (7.1.1) tra flusso e radianza, che per comodità riscriviamo qui:

$$L(x \leftarrow \mathbf{\Psi}) = \frac{d^2\Phi(x \leftarrow \mathbf{\Psi})}{d\omega \; d\sigma \, \langle N_x, \mathbf{\Psi} \rangle} \, .$$

Riscriviamo quindi l'integrale in questo modo:

$$\begin{aligned} L_r(x \to \mathbf{\Theta}) &= \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) \frac{d^2\Phi(x \leftarrow \mathbf{\Psi})}{d\omega \; d\sigma \, \langle N_x, \mathbf{\Psi} \rangle} \langle N_x, \mathbf{\Psi} \rangle \, d\omega(\mathbf{\Psi}) \\ &= \int_{\Omega_x} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) \frac{d^2\Phi(x \leftarrow \mathbf{\Psi})}{d\sigma} \, . \end{aligned}$$

Ora possiamo utilizzare la photon map per approssimare il flusso in entrata su una superficie centrata in x, ΔA, attraverso la ricerca dei fotoni su di essa depositati. Ogni fotone, p, ha una diversa potenza $\Delta\Phi_p(x \leftarrow \mathbf{\Psi})$; la densità di fotoni è calcolata sommando questi contributi e dividendo per l'area di ricerca.

$$L_r(x \to \mathbf{\Theta}) \approx \sum_{p=1}^{n} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) \frac{\Delta\Phi_p(x \leftarrow \mathbf{\Psi})}{\Delta A}$$

In ogni scena sono presenti zone in cui il numero di fotoni è alto, che chiamiamo ad alta densità, e zone in cui il numero di fotoni è basso (come ad esempio le zone in ombra), quindi a bassa densità. In quest'ultime è necessario utilizzare una vasta area di ricerca, che comprenda un numero di fotoni sufficiente ad ottenere una stima accurata. In questo modo però avremo una bassa variazione nell'illuminazione della scena. Scegliendo una vasta area di ricerca, infatti, si prendono in considerazione anche i fotoni che sono lontani dal punto osservato. Spostare quest'ultimo ha quindi minor rilevanza nel calcolo, poiché i fotoni presi in considerazione rimangono sempre gli stessi. Nelle zone ad alta densità questo non sarebbe necessario e si potrebbe utilizzare un'area di ricerca più piccola, così da ottenere una maggiore variazione. Se stabilissimo, preliminarmente, il valore ΔA per tutti i punti della scena non si potrebbero visualizzare in modo adeguato entrambe le zone. Questo problema si risolve facendo variare l'area di ricerca in base al numero di fotoni che si vogliono utilizzare per la stima. Si localizzano quindi gli n fotoni più vicini, espandendo una sfera centrata in x. Se assumiamo che la superficie è piatta attorno al punto x, l'area considerata nella ricerca è la proiezione della sfera su questa superficie ovvero:

$$\Delta A = \pi r^2 \, .$$

Otteniamo quindi la seguente approssimazione:

$$L_r(x \to \mathbf{\Theta}) \approx \frac{1}{\pi r^2} \sum_{p=1}^{n} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) \Delta\Phi_p(x \leftarrow \mathbf{\Psi}) \, .$$

È chiaro da quanto detto che *l'accuratezza* dipende dal numero di fotoni usati per la ricerca rispetto al totale dei fotoni presenti nella photon map. Se questa proporzione è piccola otteniamo un'immagine ben definita ma con maggior rumore (poiché la stima è effettuata con pochi fotoni).

Se la proporzione invece si avvicina ad 1 allora l'immagine sarà priva di rumore ma meno definita. Aumentando il numero di fotoni queste differenze tendono a diminuire, in quanto anche nelle aree piccole riscontriamo un numero di fotoni sufficienti per la stima. Abbiamo quindi questo interessante risultato che garantisce la consistenza del metodo:

$$L_r(x \to \mathbf{\Theta}) = \lim_{N\to\infty} \frac{1}{\pi r^2} \sum_{p=1}^{N^\alpha} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) \Delta\Phi_p(x \leftarrow \mathbf{\Psi}) \qquad \forall \alpha \in]0,1[\,,$$

dove N è il numero di fotoni della photon map. All'interno del programma il valore α, che gestisce l'accuratezza del calcolo, è definito col nome di `P_accuracy` per la photon map globale e col nome di `C_accuracy` per la mappa delle caustiche. Si utilizza questo parametro solo se il numero di fotoni per la ricerca, `nPhotonSearch` e `nCausticSearch`, è inizializzato a 0. Per la conservazione degli n fotoni più vicini viene utilizzato il contenitore associativo `std::map`, i cui elementi sono formati dalla combinazione di un valore chiave, corrispondente alla distanza dal punto osservato, e un fotone. Il valore chiave viene utilizzato per il riordinamento e l'identificazione dei fotoni all'interno della mappa; in questo modo i fotoni saranno ordinati rispetto alla distanza nello stesso momento in cui sono caricati dentro alla mappa. La funzione `locate_photons` si occupa della ricerca degli n fotoni più vicini; per velocizzare questa ricerca generalmente si caricano i fotoni all'interno di un Kd-Tree, struttura ad albero binario. Questo sarà analizzato più nel dettaglio nel prossimo capitolo insieme alla funzione `locate_photons` (vedi 12.2 a pagina 478). Una possibile fonte di errore è individuabile nel **metodo di ricerca dei fotoni**. Infatti se utilizzassimo per la localizzazione una sfera, rischieremmo di includere anche i fotoni che non colpiscono la superficie, come accade negli angoli e negli oggetti spigolosi; per evitare questo problema si potrebbe **limitare la ricerca ad un disco** che si espande nella superficie. Se il numero di fotoni è basso questa soluzione ha l'effetto di aumentare il rumore nelle patch piccole della scena, le quali hanno meno probabilità di ricevere fotoni, così come avveniva per il metodo di Jacobi. Nel programma si è deciso di inserire un parametro `sphericalSearch` compreso tra 0 e 1, che corrisponde al coseno dell'angolo di deviazione del fotone trovato dalla normale. Se il valore è 1 avremo quindi la ricerca su una sfera mentre se è 0 la ricerca è sul disco.

11.3.2.3. *Filtraggio.* Si incorre in un altro errore quando la ricerca avviene sul **bordo di una patch**, in questi casi, infatti, l'area di ricerca non comprende tutto il disco, ma solo l'intersezione di questo con la superficie in esame. Una semplice strategia per diminuire l'errore è data dal **filtraggio**. Questo metodo permette di incrementare il peso dei fotoni molto vicini al punto di interesse x. Inoltre utilizzando il filtraggio si possono diminuire gli effetti di sfocatura, tipici del photon mapping, nelle caustiche e nelle ombre. Nel programma si adotta il filtro a cono **[24]**, nel quale si assegna un peso ad ogni fotone, ω_{pc}, calcolato in base alla distanza, d_p, tra il punto osservato x e il fotone p. Il peso è:

$$\omega_{pc} = 1 - \frac{d_p}{k\, r}$$

dove $k \geqslant 1$ è una costante che caratterizza il filtro e r è la distanza massima di ricerca. La normalizzazione del filtro è $1 - \frac{2}{3k}$ e la stima della radianza filtrata diventa:

$$L_r(x \to \mathbf{\Theta}) \approx \frac{1}{\pi r^2} \sum_{p=1}^{n} f_r(x, \mathbf{\Psi} \leftrightarrow \mathbf{\Theta}) \Delta\Phi_p(x \leftarrow \mathbf{\Psi}) \frac{\omega_{pc}}{(1 - \frac{2}{3k})}$$

Nel programma si utilizzano due variabili che definiscono il valore k: la prima per le caustiche, `Cfilter`, e la seconda per la photon map globale, `Pfilter`.

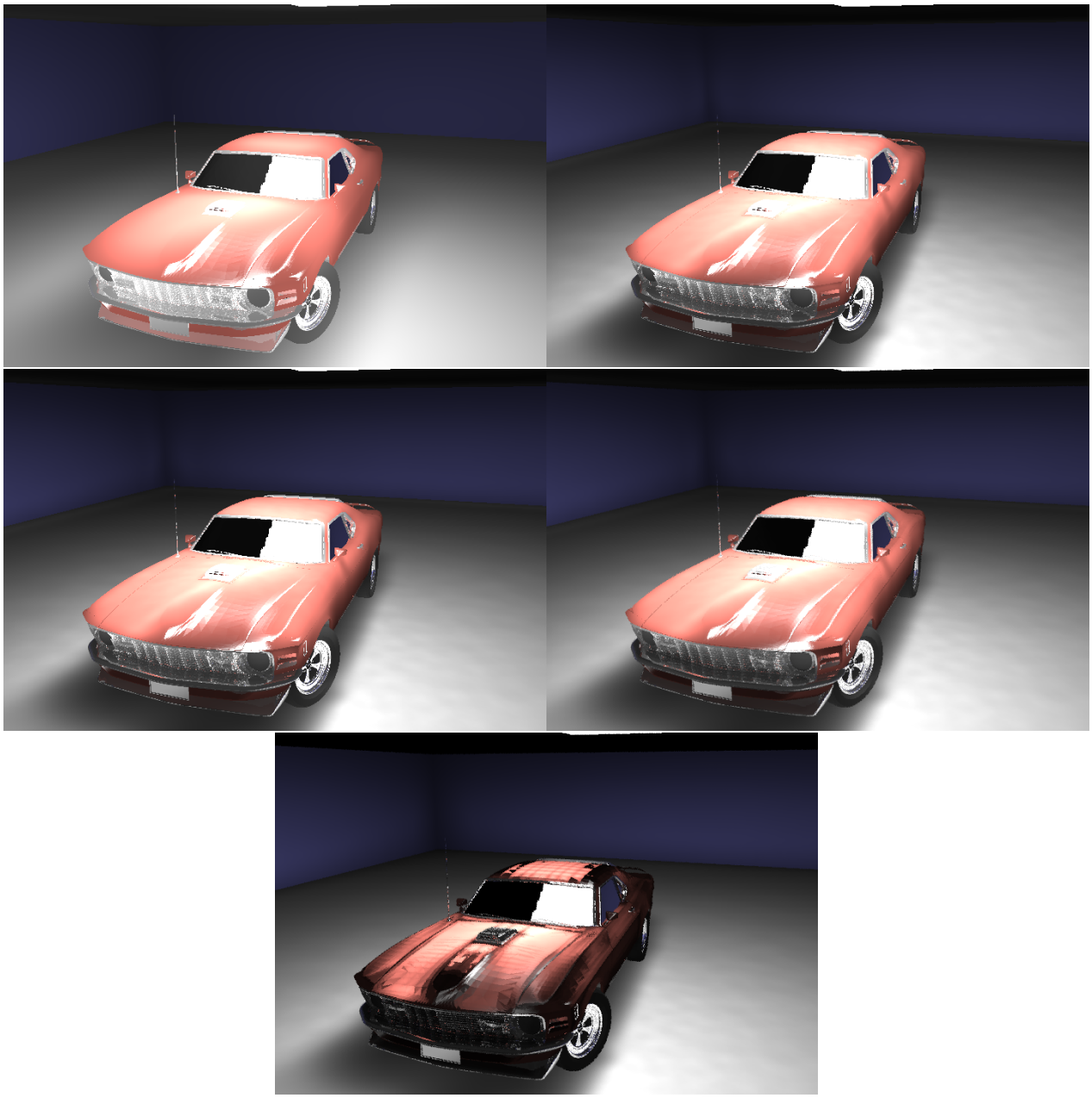


FIGURA 11.3.6. **utilizzo del parametro** `sphericalSearch`: le immagini sono state realizzate con 500000 fotoni ed impostando il parametro `sphericalSearch` come: 1, $\sqrt{3}/3$, $\sqrt{(2)}/2$, 1/2, 0. Se si utilizzano pochi fotoni la ricerca sferica permette di considerare fotoni vicini anche se non sullo stesso piano del triangolo, effettuando una ricerca planare invece ci si assicura che il fotone appartenga allo stesso piano del triangolo, di conseguenza il rendering risulterà più corretto.

Inseriamo qui il codice relativo alla stima della radianza:

```
<<srt.cpp>>
```

```
//funzione che deriva il valore della radianza dalla photon map conservata dentro
    alla variabile Tree:
float3 photonRadiance(Ray& r,Obj* objX,photonBox** Tree,float photond_2,int
    nph,float k){
float3 radianceOutput;

    //viene caricato l'ID del materiale
    int matId=objX->matId;
    //viene caricata la normale dell'oggetto
    float3 n1= (objX)->normal(r.o);
    //si inizializza la mappa che conserverà gli nph fotoni più vicini al punto r.o
        insieme alla loro distanza da tale punto
    std::map< float ,photon* > nearPh;
    //si inizializza l'iteratore per scorrere la mappa
    std::map< float ,photon* > ::iterator it;
    //si inizializza la distanza massima di ricerca
    float d_2=photond_2;
    //si localizzano gli nph fotoni più vicini a r.o scorrendo il Kd-Tree
    locate_photons(nearPh,r.o,1,objX,Tree,d_2,nph);

    //per ogni fotone it trovato:
    for (it=nearPh.begin(); it!=nearPh.end(); ++it){

        //si carica il raggio di entrata del fotone
        Ray psi= Ray(r.o,it->second->direction);
        //calcolo della BRDF:
        float3 BRDF=material[matId].C_T_BRDF(psi,r,n1);
        //si ricava la distanza del fotone dal punto:
        float dist=it->first;
        //si calcola il peso per il filtro a cono:
        float W=1-(sqrtf(dist)/(k*sqrtf(d_2)));
        //stima della radianza nel punto r.o :
        radianceOutput=radianceOutput+BRDF.mult(it->second->power)
        *(M_1_PIf)*(1/d_2)*(W/(1-(2/(3*k))));
        }
//si restituisce il valore di radianza
return radianceOutput;
}
```

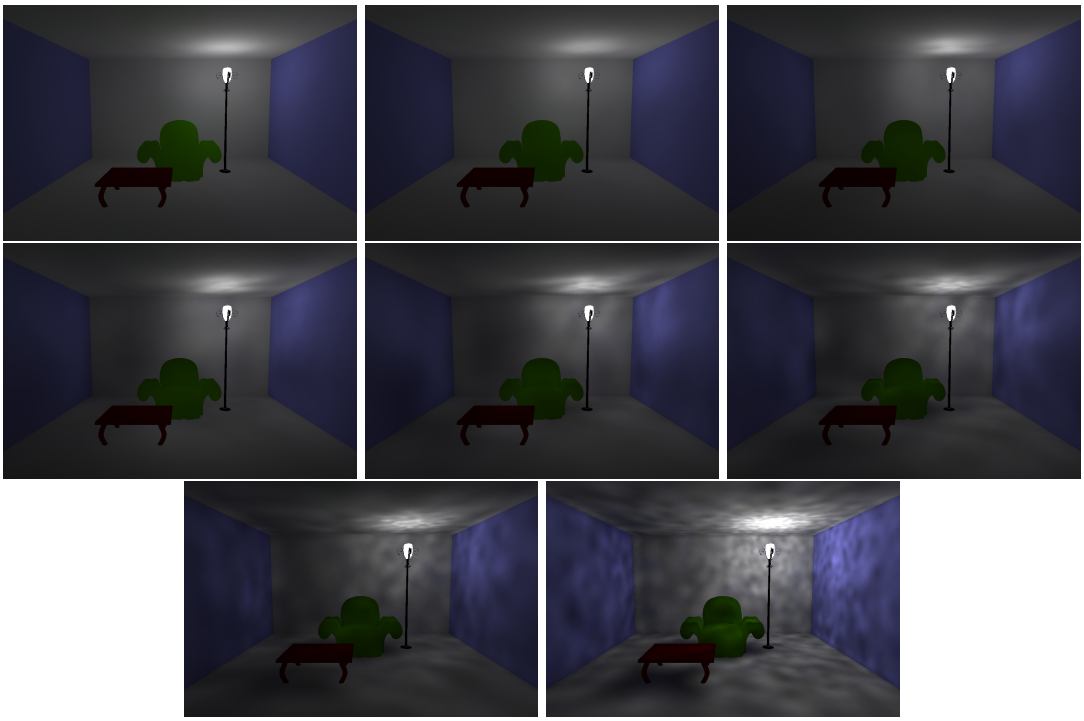


FIGURA 11.3.7. **aumento dei fotoni nel Photon mapping**: in queste immagini si sono utilizzati 80 fotoni per la ricerca e $200 \cdot 2^n$, dove n è il numero dell'immagine, fotoni emessi.

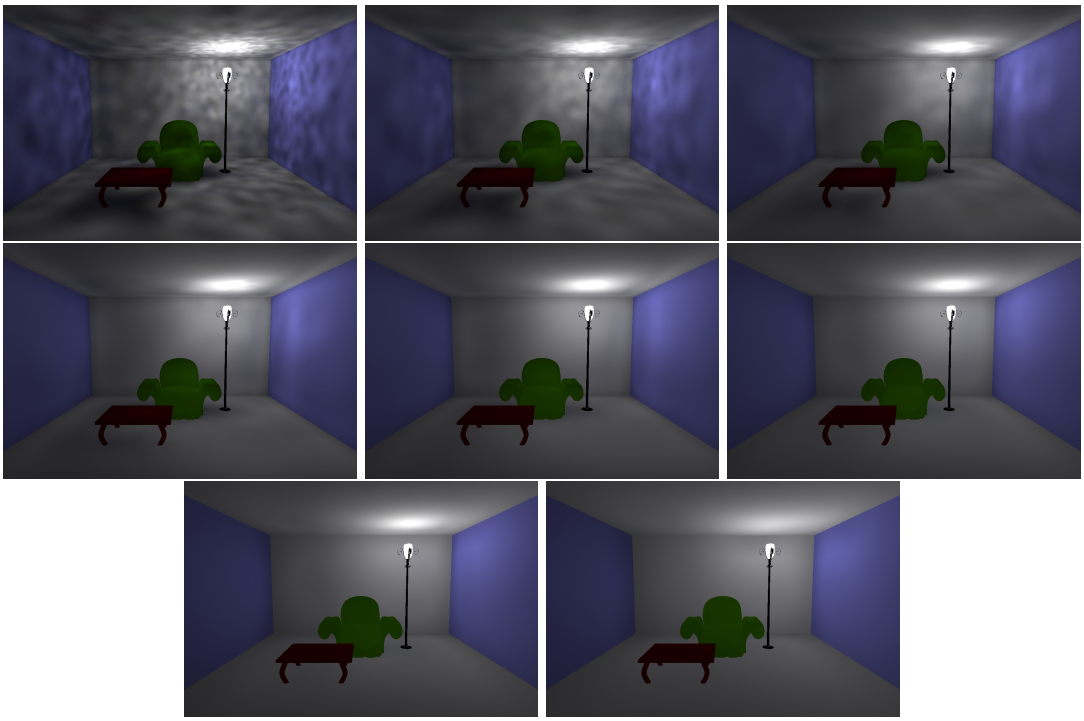


FIGURA 11.3.8. **aumento del numero di fotoni da ricercare**: per ognuna di queste immagini, che numeriamo con $n = 0, 1, \ldots$, sono stati emessi 25600 fotoni, di cui $80 \cdot 2^n$ sono utilizzati per la ricerca.

11.3.2.4. *Passo 3: La mappa delle caustiche.* La mappa delle caustiche contiene i fotoni che sono riflessi o trasmessi da una superficie speculare, prima di colpire una superficie diffusiva. Aumentando il numero di fotoni emessi verso queste zone, otteniamo una minore sfocatura ed una migliore definizione delle caustiche. Questa concentrazione di fotoni emessi avviene tramite una proiezione della scena rispetto alla luce. La luce, infatti, per scegliere la direzione di emissione deve poter visualizzare l'intera scena, così da identificare le zone contenenti oggetti speculari. Nel programma le luci sono emettitori diffusivi ad area, quindi l'emissione di fotoni avviene scegliendo prima un punto all'interno della superficie e poi una direzione rispetto all'emisfero frontale. Per questo motivo si è scelto di effettuare una proiezione emisferica per ogni campione della luce. Così facendo l'emissione dei fotoni rallenta; per velocizzare il processo è stato aggiunto un nuovo parametro, `aoCausticPhoton`, che gestisce il numero di fotoni usati per ciascun campione della superficie, quindi per ciascuna proiezione. L'utilizzo di questo parametro potrebbe comportare degli errori, se la superficie della luce è vasta, e ad una caustica con una forma errata. In questo caso il problema si risolve impostando un valore `aoCausticPhoton` pari ad 1. La funzione `Projection` si occupa di costruire una mappa emisferica della scena. Questa mappa ha una risoluzione definita dal parametro `ProjectionResolution`, in base al quale l'emisfero viene diviso in settori sferici di area uguale. Utilizzando questi settori si cerca di campionare uniformemente la poligonale sferica corrispondente agli oggetti speculari proiettati sull'emisfero.

PROPOSIZIONE 11.3.2 (Campionamento uniforme di una poligonale sferica). *La seguente procedura dà luogo ad un insieme equidistribuito di campioni su una poligonale sferica. Si comincia con lo scindere la poligonale P come unione finita di settori $S_{\Delta\Theta_i,\Delta\Psi_j}$, dove $i = 0, 1, \ldots n_r$ e $j = 0, 1, \ldots n_r$, con area uguale, ovvero*

$$m(S_{ij}) = m(S)\,.$$

Vengono quindi create due successioni nelle variabili date dall'angolo θ di deviazione dalla normale e dall'angolo ϕ di longitudine, e si considerano gli intervalli così generati:

$$\theta_i \leqslant \theta \leqslant \theta_{i+1}, \quad \phi_j \leqslant \phi \leqslant \phi_{j+1}\,.$$

Poiché P è misurabile, questo può essere fatto in modo da avere una esaustione dell'area di P:

$$m\left(\bigcup_{i,j} S_{\Delta\Theta_i,\Delta\Psi_j} - P\right) \leq \epsilon\,.$$

Infine, per ciascun $S_{\Delta\Theta_i,\Delta\Psi_j} = S_{ij}$ si generano campioni "uniformi" così:

$$dp(\boldsymbol{v})_{ij} = dp(\boldsymbol{v}_{\theta,\phi}) = \frac{1}{m(S_{ij})}\sin\theta\, d\theta\, d\phi\,.$$

Dimostrazione. Abbiamo visto che per una PDF uniforme su S_{ij} vale

$$\begin{aligned}\int\int_{S_{ij}} dp(\boldsymbol{v})_{ij} = & \\ &= \int\int_{S_{ij}} \frac{1}{m(S)} \\ &= \frac{1}{m(S)}\int_{\phi_j}^{\phi_{j+1}}\int_{\theta_i}^{\theta_{i+1}} \sin\theta\, d\theta\, d\phi \\ &= 1\end{aligned}$$

poiché $\sin\theta$ è lo Jacobiano della trasformazione in coordinate sferiche. Basta ora calcolare qual'è la probabilità di colpire P, dove

$$P = \bigcup_{i,j=1}^{\infty} S_{i,j}.$$

P è misurabile poiché unione di infiniti settori sferici disgiunti, e quindi, se i settori $S_{i,j}$ si scelgono di area sufficientemente piccola, $\forall\epsilon\,\exists N < M$ tali che i settori sferici $S_{i,j}$ con i e $j < N$ sono contenuti in P, e

$$\bigcup_{i,j=1}^{M} S_{i,j} \subset P \subset \bigcup_{i,j=1}^{N} S_{i,j} \tag{11.3.1}$$

e

$$m\left(\bigcup_{i,j=1}^{M} S_{i,j} - \bigcup_{i,j=1}^{N} S_{i,j}\right) < \epsilon.$$

La differenza $E = \bigcup_{i,j=1}^{M} S_{i,j} - \bigcup_{i,j=1}^{N} S_{i,j}$ è unione di settori sferici che denominiamo "di cornice". La probabilità di campionare un insieme misurabile E piccolo rispetto alla partizione fatta, ovvero con $\frac{m(E)}{m(S)} < \mu$, è piccola:

$$Prob(E) = \int\int_E dp(\boldsymbol{v}) = \frac{1}{m(S)}\int\int_E \sin\theta d\theta d\phi = \frac{m(E)}{m(S)} < \mu\,.$$

Di conseguenza il limite all'infittirsi della partizione della probabilità di campionare i settori sferici dentro P è la probabilità di colpire P:

$$Prob(P) = \sum_{i,j}^{\infty} Prob(S_{ij})\,.$$

□

Al fine di ottenere una partizione equidistribuita lungo l'emisfero sono costruite le seguenti due successioni:

$$\theta_i = \arccos\left(1 - \frac{i}{n_r}\right) \qquad e \qquad \phi_j = j\frac{2\pi}{n_r}\,.$$

Verifichiamo allora che tramite queste due successioni l'area dei settori sferici è costante

$$\begin{aligned} m(S) = & \\ &= \int_{\phi_j}^{\phi_{j+1}} \int_{\theta_i}^{\theta_{i+1}} \sin\theta \, d\phi \, d\theta = \\ &= \frac{2\pi}{n_r}(\cos\theta_i - \cos\theta_{i+1}) = \\ &= \frac{2\pi}{n_r}(1 - \frac{i}{n_r} - 1 + \frac{i+1}{n_r}) = \\ &= \frac{2\pi}{n_r^2} \end{aligned}$$

Nel programma si caricano gli angoli, corrispondenti alla direzione degli oggetti speculari, all'interno dell'oggetto `std:Vector`, i cui elementi sono `float2`, cioè strutture contenenti due float: x e y.

<<srt.cpp>>

```
//funzione che effettua la mappatura della scena di risoluzione
   ProjectionResolution tramite proiezione su un emisfero
void Projection(float3 p,float3 n,std::vector<float2>& ProjectionMap){
       //base ortonormale
       float3 u,v,w;
       w=n;
       float3 up(0.0015f,1.0f,0.021f);
       v=w%up;
       v.norm();
       u=v%w;
       Obj** objY=new Obj*();
       //creazione della mappa
       for(int i=0; i<ProjectionResolution; i++){
           //angolo di latitudine
           float Theta=acosf(1-(i/ProjectionResolution));
           for(int j=0; j<ProjectionResolution; j++){
              //angolo di longitudine
              float Phi=j*(2*M_PIf)/ProjectionResolution;
               //si crea il raggio col compito di verificare la presenza di un
                  materiale speculare
               float3 dir=u*(cos(Phi)*sinf(Theta))
               +v*(sin(Phi)*sinf(Theta))+w*(cosf(Theta));
               Ray pRay= Ray(p,dir);
               float t=inf;
               *objY=NULL;
               if(intersect(pRay, t, objY)){

                   //si verifica la presenza di un materiale speculare
```

```
                if( (material[(*objY)->matId].Kg.max()>0)||(
                    (material[(*objY)->matId].Kr.max()>0)
                &&(ReflectedCaustic))){
                    //l'angolo viene conservato all'interno della mappa di
                        proiezione
                    float2 angle=float2(Phi,Theta);
                    ProjectionMap.push_back(angle);
                }
            }
        }
    }
    delete objY;
}
```

Per la generazione di fotoni diretti verso le caustiche si sceglie una delle luci, con probabilità uniforme, tra tutte quelle presenti nella scena. Anche la superficie della luce si campiona con probabilità uniforme e si crea la mappa di proiezione a partire dal punto trovato. Dai concetti visti nel capitolo 8 sappiamo che la potenza del fotone dovrà essere moltiplicata per l'area della luce, proiettata rispetto all'angolo di uscita del fotone, e il numero delle luci. Dalla funzione `Projection` ricaviamo il vettore contenente gli angoli da campionare, a cui corrispondono un egual numero di settori sferici. Questi settori definiscono un angolo solido pari a $2\pi/n_r^2$; moltiplicando per il numero di angoli risultanti dal processo, `nAngle`, otteniamo l'angolo solido totale della proiezione. L'emissione dei fotoni avviene però sull'intero emisfero quindi la potenza del fotone deve essere moltiplicata per il rapporto tra l'angolo solido ricoperto dalla poligonale e quello totale, che su un emisfero è 2π. Otteniamo quindi la potenza P_{ph} del fotone:

$$P_{ph} = \frac{n_l\, L_e \pi}{n_{ph}}\, m(l) \cos\theta\, \frac{n_s 2\pi}{2\pi n_r^2} = \frac{n_l\, L_e \pi}{n_{ph}}\, m(l) \cos\theta\, \frac{n_s}{n_r^2}\,.$$

Qui $m(l)\cos(\theta)$ rappresenta l'area della luce proiettata rispetto all'angolo di uscita θ, n_l è il numero di luci nella scena, n_{ph} è il numero di fotoni usati e n_s è il numero di settori che formano la poligonale. All'interno di ogni settore circolare il campionamento deve avvenire in base alla densità di probabilità: $dp(v_{\theta,\phi})_{ij} = \frac{1}{m(S)} \sin\theta d\theta d\phi$. Costruiamo la funzione di ripartizione:

$$\begin{aligned} F(\theta,\phi) = & \\ &= \frac{1}{m(S)} \int_{\phi_j}^{\phi} \int_{\theta_i}^{\theta} \sin\theta d\theta d\phi = \\ &= \frac{1}{m(S)} (\phi - \phi_j)\,(\cos\theta_i - \cos\theta) = \\ &= \frac{{n_r}^2}{2\pi} (\phi - \phi_j)\,(\cos\theta_i - \cos\theta) \end{aligned}$$

L'Inverse Cumulative Distribution Function è composta da due funzioni una per ϕ e una per θ:

$$\phi = \phi_j + \frac{2\pi}{n_r} u_1 \qquad e \qquad \theta = \arccos\left(\cos\theta_i - \frac{u_2}{n_r}\right)$$

dove u_1 e u_2 sono due variabili equidistribuite in $[0,1]$. Inseriamo il codice relativo all'emissione dei fotoni per le caustiche qui in basso:

```
<<srt.cpp>>
```

```
//emissione dei fotoni per la creazione della mappa delle caustiche:
void caustic(std::vector<photon*>& caustics){
    //si inizializza il seme per la generazione randomica di tipo LCG
    unsigned int rn=rand();
    //variabili per la scelta del punto nella superficie della luce:
    float rnd1;
    float rnd2;
    float rnd3;
    Obj** objX=new Obj*();
    //per ogni campione
    for(int nP=0;nP<causticPhoton;nP++){

    //viene scelta la luce da campionare tra le nLight luci
    if(aST==SOB){
        rnd1=generateRandom(aoSId[0],8,aST)*(nLight-1);
    }
    else{
        rnd1= generateRandom(rn,nP+1,aST)*(nLight-1);
    }
    //indice della luce:
    unsigned long l=(unsigned long)floorf(rnd1);
    //si carica l'area e l'identificativo del materiale della luce
    float area= luci[l]->areaObj;
    int lid= luci[l]->matId;
    //si inizializza la mappa di proiezione
    std::vector<float2> ProjectionMap;
    //si carica la potenza iniziale di ciascun fotone
    float3 P= material[lid].Le*M_PIf*area*nLight/(causticPhoton*aoCausticPhoton);
           //si campiona uniformemente la superficie dell'oggetto
           if(aST==SOB){
               rnd1=generateRandom(aoSId[0],5,aST);
               rnd2=generateRandom(aoSId[0],6,aST);
               rnd3=generateRandom(aoSId[0],7,aST);
           }
           else{
               rnd1= generateRandom(rn,s+1,aST);
               rnd2= generateRandom(rn,s+1,aST);
               rnd3= generateRandom(rn,s+1,aST);
           }
       //si campiona un punto all'interno della superficie della luce
       float3 point= luci[l]->randomPoint(rnd1,rnd2,rnd3);
       //si carica la normale nel punto
       float3 normal=luci[l]->normal(float3(point));
       //si calcola la mappa di proiezione rispetto al punto e alla sua normale
       Projection(point,normal,ProjectionMap);
       //si carica il numero di settori che formano la poligonale sferica
        int nAngle=(int)ProjectionMap.size();
          //si continua solo se son stati trovati oggetti speculari
```

```
if(nAngle>0){
//si crea la base ortonormale
    float3 u,v,w;
    w=normal;
    float3 up(0.0015f,1.0f,0.021f);
    v=w%up;
    v.norm();
    u=v%w;
//per ogni campione emisferico utilizzato ( per un massimo di
    aoCausticPhoton )
for(int nS=0;nS<aoCausticPhoton;nS++){

//si campiona uniformemente la mappa di proiezione
if(aST==SOB){
        rnd1=generateRandom(aoSId[0],9,aST)*(nAngle-1);
    }
else{
        rnd1= generateRandom(rn,s+1,aST)*(nAngle-1);
    }
//si carica il settore dalla mappa di proiezione
float2 angle= ProjectionMap[(unsigned long)floorf(rnd1)];

float Phi;
float Theta;
//campionamento uniforme del settore
if(aST==SOB){
    Phi=angle.x+generateRandom(aoSId[0],5,aST)
    *(2*M_PIf/(float)ProjectionResolution);
    Theta=acosf(cosf(angle.y)-(generateRandom(aoSId[0],6,aST)
    /((float)ProjectionResolution)));
}
else{
    Phi=angle.x+generateRandom(rn,nS+1,aST)
    *(2*M_PIf/(float)ProjectionResolution);
    Theta=acosf(cosf(angle.y)-(generateRandom(rn,nS+1,aST)
    /((float)ProjectionResolution)));
}
//creazione del raggio del fotone
float3
    dir=u*(cos(Phi)*sinf(Theta))+v*(sin(Phi)*sinf(Theta))+w*(cosf(Theta));
Ray pRay= Ray(point,dir);
float t=inf;
*objX=NULL;
int n=0;
if(intersect(pRay, t, objX)){
//si controlla se l'oggetto colpito \'{e} effettivamente speculare
if( (material[(*objX)->matId].Kg.max()>0)
||( (material[(*objX)->matId].Kr.max()>0)
&&(ReflectedCaustic))){
```

```
                //si scala la potenza del fotone in base all'angolo solido ricoperto
                    dalla poligonale
                float scale= nAngle/(float)(ProjectionResolution
                          *ProjectionResolution);
                //si calcola la potenza del fotone
                float3 P2=P*scale*fabs(cosf(Theta));
                //si crea il nuovo fotone
                float3 iP=pRay.o+pRay.d*t;
                photon* p2=new photon(iP,dir*(-1),P2);
                //il fotone continua il suo percorso:
                causticScatter(caustics,(*objX),n,p2);
            }
          }
        }
    }
 }
    delete objX;
}
```

La propagazione dei fotoni nella scena in questo caso continua finché non si incontra un oggetto diffusivo. Nel caso in cui si incontri un oggetto speculare, invece, la riflessione o rifrazione del fotone sarà decisa in base al coefficiente di Fresnel del materiale.

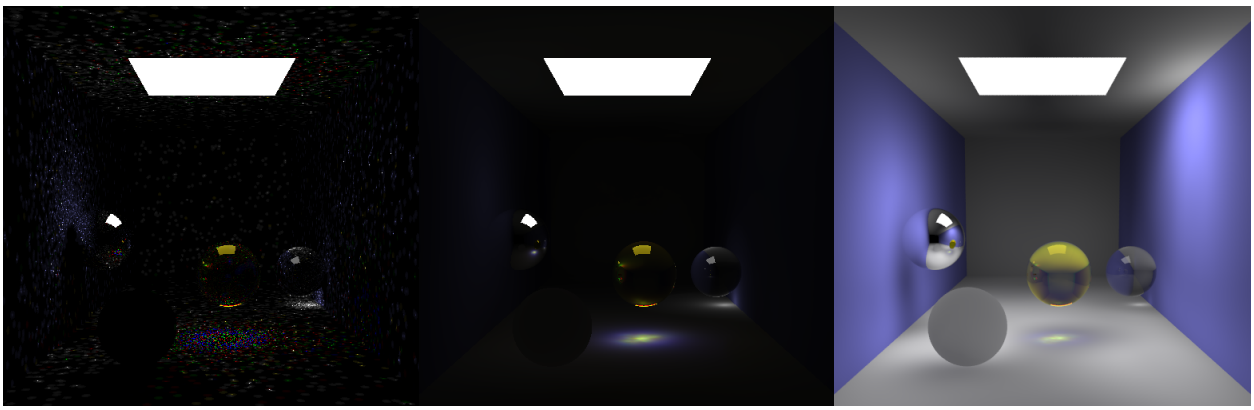


FIGURA 11.3.9. **Rendering delle caustiche con il photon map**: da sinistra a destra si possono analizzare, i fotoni che fanno parte della mappa (impostando 200 `causticPhoton` e 100 `aoCausticPhoton`), la visualizzazione di quest'ultima tramite 1382 fotoni nella stima e un filtro a cono ($k = 1.4$) e nell'ultima immagine

La probabilità di riflessione corrisponde al valore `Pref`, ovvero la media delle 3 componenti RGB del coefficiente di Fresnel, mentre la probabilità di rifrazione è esattamente $1 - $ `Pref`. Per far si che le caustiche provenienti da riflessioni siano visibili deve essere impostato il parametro `ReflectedCaustic` su `True`.

```
<<srt.cpp>>
```

```
//funzione per il salvataggio dei fotoni nella mappa delle caustiche
void causticScatter(std::vector<photon*>& caustics,Obj* objX, int n_,photon* p){
```

```
// si carica l'identificativo del materiale
 int mId= objX->matId;

// se il materiale \'{e} diffusivo si salva il fotone nella mappa
 if((material[mId].Kd.max()>0)||(material[mId].slope!=0)){
     caustics.push_back(p);
 }

 //si verifica che non \'{e} stato superato il numero massimo di riflessioni per
     le caustiche
 if(n_<MAX_DEPTH_CAUSTIC){

  Obj** objY= new Obj*();
  //si carica la normale all'oggetto
  float3 n = objX->normal(p->position);
 //si carica il raggio di entrata del fotone
 Ray entryRay= Ray(p->position,p->direction);
 //si calcola il coseno di entrata del fotone e il coefficiente di Fresnel
 float cos_i=n.dot(p->direction);
 float3 Fresn=material[mId].getFresn(cos_i);

 //si crea una probabilità utilizzando il coefficiente di Fresnel, con essa si
     decide se il fotone \'{e} riflesso o rifratto:
 float Pref=Fresn.media();

 float rnd;
 if(aST==SOB){
     rnd=generateRandom(aoSId[0],3,aST);
 }
 else{
     rnd=rand()/RAND_MAX;
 }

 //si verifica la riflessione del fotone
 if((material[mId].Kr.max()>0)&&(rnd<Pref)&&(ReflectedCaustic)){
     //calcolo della direzione di riflessione
     float3 refl= reflect(p->direction,n);
     Ray reflRay;
     //creazione del raggio riflesso
     reflRay.o=p->position;
     reflRay.d=refl;

     float t2=inf;
     if(intersect(reflRay,t2,objY)){
         //si aumenta il numero delle riflessioni totali relative al calcolo
             delle caustiche
         n_++;
         //creazione del nuovo fotone
```

```
            float3 iP=reflRay.o+reflRay.d*t2;
            float3 oldP=p->power;
            float3 BRDF = material[mId].S_BRDF(Fresn);
            float3 newP= oldP.mult(BRDF)*(1/Pref);

            photon* p2=new photon(iP,refl*(-1),newP);
            //il nuovo fotone continua il suo percorso
            causticScatter(caustics,*objY,n_,p2);
        }
    }

    //si verifica la rifrazione del fotone
    if((material[mId].Kg.max()>0)&&(material[mId].k.max()==0)
      &&(rnd>Pref)){

        //velocizzazione del calcolo:
        //se IOR \‘{e} uguale per tutte e tre le componenti RGB si usa un solo
            raggio rifratto
        if((material[mId].ior.x==material[mId].ior.y)
        &&(material[mId].ior.x==material[mId].ior.z)){
            float3 dir;
            //calcolo della direzione del raggio rifratto:
            if(refract(dir,entryRay.d,n,material[mId].ior.x)){
                //si crea il raggio rifratto
                Ray refrRay(entryRay.o,dir);
                float t2=inf;
                if(intersect(refrRay,t2,objY)){
                    //si aumenta il numero delle riflessioni totali relative al
                        calcolo delle caustiche
                    n_++;
                    //creazione del nuovo fotone
                    float3 iP=refrRay.o+refrRay.d*t2;
                    float3 oldP=p->power;
                    float3 BRDF= material[mId].T_BRDF(Fresn);
                    float3 newP= oldP.mult(BRDF)*(1/(1-Pref));
                    photon* p2=new photon(iP,refrRay.d*(-1),newP);
                    //il nuovo fotone continua il suo percorso
                    causticScatter(caustics,*objY,n_,p2);
                }
            }
        }
        //se l’indice di rifrazione \‘{e} differente per le tre componenti RGB
            allora si devono creare tre raggi rifratti,
        // uno per ogni componente
        else{

            //inizializzazione dei tre raggi rifratti
            Ray* refrRay= new Ray[3];
            refrRay[0].o=p->position;
```

```
refrRay[1].o=p->position;
refrRay[2].o=p->position;

//rifrazione dei raggi in base all'IOR
refract(refrRay,p->direction,n,material[mId].ior);
float t2=inf;
float3 oldP=p->power;
//aumento il numero delle riflessioni
n_++;
//calcolo della BRDF del materiale
float3 BRDF = material[mId].T_BRDF(Fresn);
//si verifica che non ci sia stata riflessione totale interna della
    componente Rossa
if(refrRay[0].depth!=0){
    *objY=NULL;
    if(intersect(refrRay[0],t2,objY)){
        //creazione del fotone con la componente Rossa
        float3 iP=refrRay[0].o+refrRay[0].d*t2;
        float3 newP= float3(oldP.x*BRDF.x,0,0)*(1/(1-Pref));
        photon* p2=new photon(iP,refrRay[0].d*(-1),newP);

        causticScatter(caustics,*objY,n_,p2);
    }
}
//si verifica che non ci sia stata riflessione totale interna della
    componente Verde
if(refrRay[1].depth!=0){

    t2=inf;
    *objY=NULL;
    if(intersect(refrRay[1],t2,objY)){
        //creazione del fotone con la componente Verde
        float3 iP=refrRay[1].o+refrRay[1].d*t2;
        float3 newP= float3(0,oldP.y*BRDF.y,0)*(1/(1-Pref));
        photon* p2=new photon(iP,refrRay[1].d*(-1),newP);

        causticScatter(caustics,*objY,n_,p2);
    }
}
//si verifica che non ci sia stata riflessione totale interna della
    componente Blu
if(refrRay[2].depth!=0){

    t2=inf;
    *objY=NULL;
    if(intersect(refrRay[2],t2,objY)){
        //creazione del fotone con la componente Blu
        float3 iP=refrRay[2].o+refrRay[2].d*t2;
        float3 newP= float3(0,0,oldP.z*BRDF.z)*(1/(1-Pref));
```

```
                    photon* p2=new photon(iP,refrRay[2].d*(-1),newP);

                    causticScatter(caustics,*objY,n_,p2);
                }
            }
               delete[] refrRay;
        }
    }
    delete objY;
  }
}
```

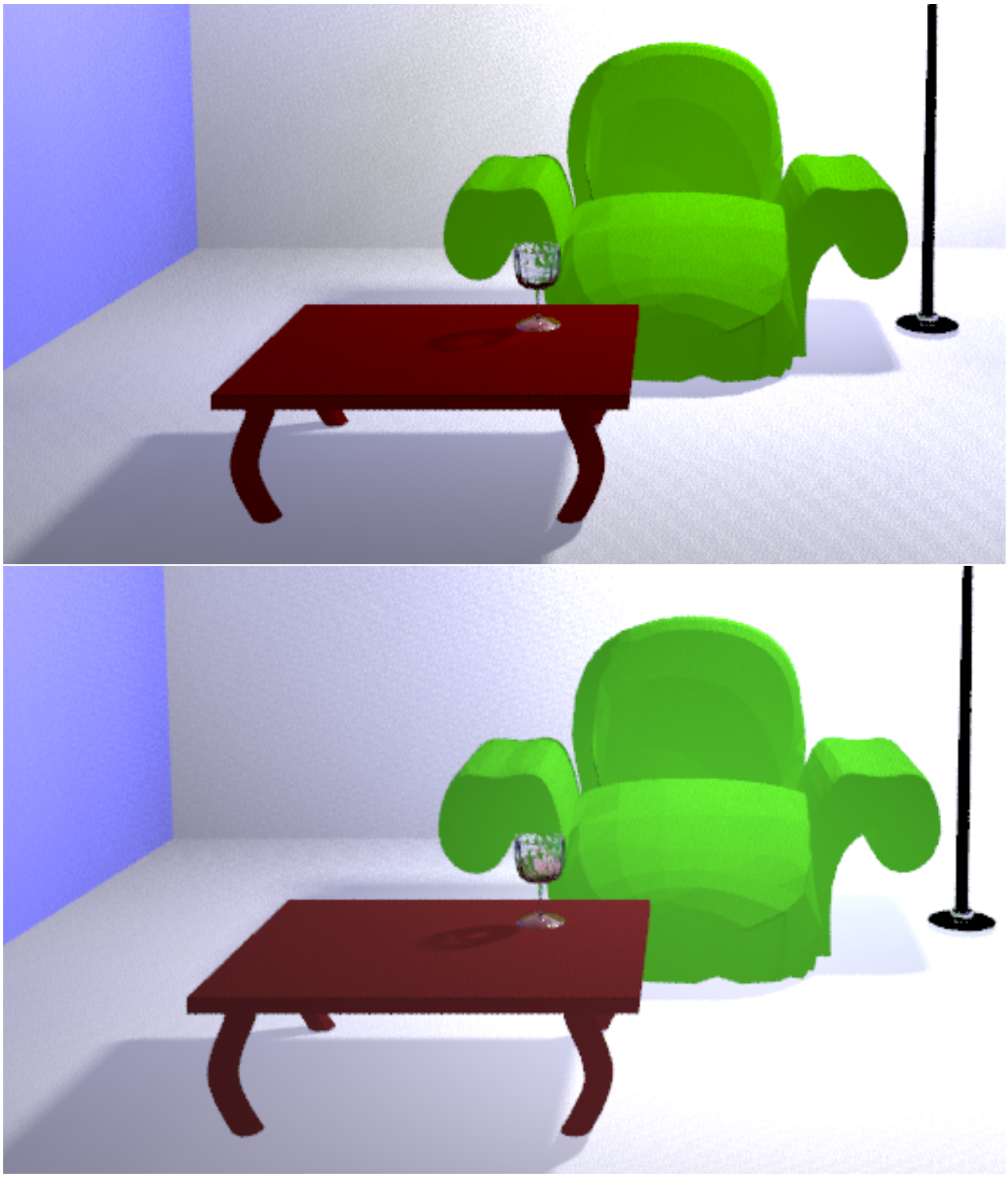


FIGURA 11.3.10. **Rendering delle caustiche**: in questa immagine si percepisce come la caustica cambia forma in base al contenuto del bicchiere.

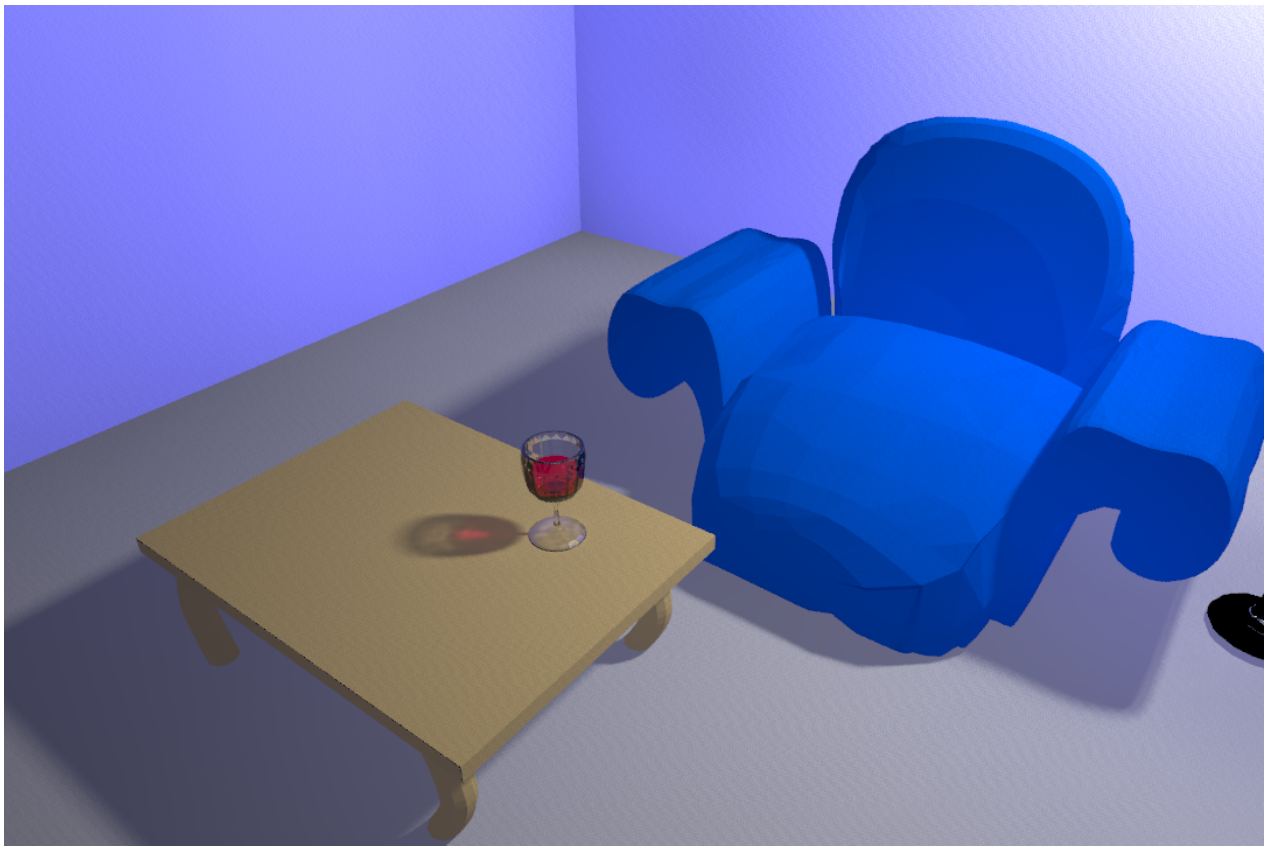


FIGURA 11.3.11. **Rendering delle caustiche**: se nel bicchiere è presente un liquido (indice di rifrazione 1.33) di colore rosso, anche la caustica cambia di colore.

CAPITOLO 12

Accelerazione del processo di rendering

Un programma di rendering fotorealistico tridimensionale richiede numerosi e lunghi calcoli da ripetere, sempre uguali, all'interno di tutta la scena. In differenti approcci al rendering ci sono parti della scena inutili da considerare ed eliminandole si ottiene un'accelerazione notevole. Ottimi strumenti utili a questo scopo sono gli alberi di ricerca. Nel programma sviluppiamo due diversi metodi, che sono metodi di ripartizione spaziale adatti ad accelerare differenti tipi di procedure:

- BSP (Binary Space Partition)
- Kd-Tree

Essi suddividono il Bounding Box, un contenitore di tutti gli elementi della scena, in più parti (che chiamiamo figli o foglie): queste parti sono anch'esse dei Box. Una volta che gli elementi sono smistati all'interno dei nuovi Box si può iterare il procedimento per ognuno di questi.

12.1. BSP (Binary Space Partition)

Il BSP divide il Box a metà (rispetto ad uno degli assi coordinati nel sistema di riferimento usato per la geometria, World Coordinates, si veda capitolo 13). L'asse perpendicolarmente al quale si prende il piano di separazione viene cambiato in ordine ciclico. Nel nostro renderer, questo metodo è utilizzato per l'intersezione tra gli oggetti della scena e i raggi del ray tracing. Ogni raggio del ray tracing viene fatto intersecare con un istanza della Struct `Box` di nome `Bound` (ovvero il Bounding Box della scena):

```
<<main.h>>
```

```
//primo elemento della lista di Box (BSP)
Box* Bound;
```

Se questa intersezione risulta soddisfatta, allora si procede a verificare l'intersezione del raggio con i figli e si continua in questo modo finché non si raggiunge l'ultima foglia dell'albero. A questo punto consideriamo l'intersezione solamente con gli oggetti presenti in questa foglia, tralasciando tutti gli altri. Questa operazione è eseguita dal metodo `intersectBSP`, che richiama se stesso finché il Box in esame `b` non ha più figli (ovvero i suoi puntatori sono nulli); se questo succede si procede con l'intersezione del raggio con tutti gli oggetti presenti nel Box. Se il raggio interseca un oggetto in una posizione `d` (parametro di intersezione del raggio con l'oggetto in esame) più piccola rispetto a `t` (parametro dell'intersezione del raggio con l'oggetto più vicino riscontrato precedentemente), allora vengono aggiornati i valori di `t` (che viene posto uguale a `d`) e di `*o` puntatore all'oggetto intersecato.

```
<<srt.cpp>>
```

```
//BSP
```

```
/// Intersect:b Box da intersecare con il raggio, r raggio , t intersezione più
    piccola con i triangoli della scena, o oggetto intersecato
inline bool intersectBSP(Box* b,const Ray &r, float &t, Obj** o){

    //se il Box è intersecato dal raggio r
    if(b->intersect(r)){

        //si controllano i figli del Box e si verificano che siano nulli (ovvero si
            verifica che il Box non abbia figli)
        if((b->leaf1!=NULL)&&(b->leaf2!=NULL)){
            //se il Box ha dei figli allora il metodo ricomincia la ricerca a
                partire da essi.
            intersectBSP(b->leaf1,r,t,o);
            intersectBSP(b->leaf2,r,t,o);
        }else{
            //si carica l'array degli oggetti all'interno del Box
            Obj** objects= b->objects;
            //si carica il numero degli oggetti all'interno del Box
            int nO= b->nObj;

            //d punto di intersezione del raggio
            float d=0;
            for(int i=0; i<nO; i++){
            //si effettua l'intersezione rispetto alla geometria dell'oggetto
                if((d=objects[i]->intersect(r))>=0.0f && d<t){
                    t=d;
                    *o=objects[i];
                }//objects intersect
            }//for

        }
    }//intersect b
    else{ return false;}
    // se t è minore del limite massimo della scena (con cui esso è inizializzato)
        allora si è verificata una intersezione
    //in questo caso il metodo restituisce true
     return t<inf;
}
```

Se il raggio `r` è utilizzato per il calcolo dell'illuminazione diretta allora si conosce già a priori l'oggetto che vorremmo intersecare: una luce. In questo caso si deve verificare solamente che il raggio non colpisca altri oggetti nel suo percorso. Per facilitare questa operazione si utilizza un ulteriore metodo `intersect`.

```
<<srt.cpp>>
```

```
inline bool intersect(const Ray &r, float &t, Obj** o){
    //se non si conosce a priori l'oggetto da intersecare si utilizza il metodo
        intersectBSP
```

```
    if((*o)==NULL){
        return intersectBSP(Bound,r,t,o);
    }else{
    //altrimenti si verifica che l'oggetto intersecato sia uguale a quello puntato
        da (*o)
        Obj* o1=(*o);
        *o=NULL;
        if(intersectBSP(Bound,r,t,o)){
            if((o1)==(*o)){
                return true;}
            else{
                return false;
            }
        }else{
            return false;
        }
    }
}
```

L'accelerazione è dovuta non solo al fatto di eliminare ad ogni passo metà della scena in esame, ma anche alla notevole velocità di intersezione tra un raggio e un Box, molto più elevata dell'usuale intersezione con un generico oggetto: infatti per sapere se un Box rettangolare viene intersecato basta verificare se, all'inizio ed alla fine della sua estensione in larghezza ed in lunghezza, il raggio passa per punti la cui ubicazione in altezza sta nel range del Box.

```
<<geometry.h>>
<<Struct Box>>
```

```
//funzione intersect del Box con un raggio:
bool intersect(const Ray& r){
    float Tnear=-INFINITY;
    float Tfar=INFINITY;
    float d[]={r.d.x,r.d.y,r.d.z};
    float o[]={r.o.x,r.o.y,r.o.z};
    float min[]={V[0].x,V[0].y,V[0].z};
    float max[]={V[1].x,V[1].y,V[1].z};

    //per ogni coppia di piani
    for(int i=0;i<3;i++){
        if(d[i]==0){
            if((o[i]<min[i])||(o[i]>max[i])){
                return false;
            }
        }
        else{
            float T1= (min[i] -o[i])/d[i];
            float T2= (max[i] -o[i])/d[i];
            //ordina dal più piccolo al più grande
            if(T1>T2) swap(T1,T2);
```

```
            //voglio il t vicino più grande
            if(T1>Tnear) Tnear=T1;
            //voglio il t lontano più piccolo
            if(T2<Tfar) Tfar=T2;

            //non c'è intersezione tra i due segmenti
            if(Tnear>Tfar){return false;}

            //il raggio interseca nella direzione opposta
            if(Tfar<0){return false;}
            }
        }
    return true;
}
```

FIGURA 12.1.1. **Rendering di una scena con 2715347 triangoli:** questo risultato, per cui si sono utilizzati 50 `sample`, 50 `dirsamps` e una camera di apertura 40, ha richiesto 1 giorno 13 ore e 53 minuti, nonostante il parametro `depth` sia stato impostato a 20. Difatti l'albero non è efficiente, in quanto la scena è molto estesa e la divisione dello spazio non implica la ripartizione dei triangoli all'interno dei Box. In questo caso un Kd-Tree avrebbe dato risultati migliori, in quanto esso basa la partizione spaziale sugli oggetti interni.

12.1.1. Creazione dell'albero BSP. Da quanto detto la struct Box deve avere certi parametro qui elencati:

```
<<geometry.h>>
<<struct Box>>
```

```
  //vertici che definiscono il Box
    float3 V[2];
    //il parametro lato ci dice il piano che dividerà il Box in due
    //corrispondenze piano:valore =  xy:0 yz:1 xz:2
    short lato=0;

    //array di puntatori ad oggetti che il Box contiene:
    Obj** objects=NULL;
    int nObj=0;

    // puntatori ai Box figli
    //nel caso rimangano Null il Box non ha nessun figlio
    Box* leaf1= NULL;
    Box* leaf2= NULL;

    //costruttore del Box
    Box(float3 min,float3 max,short l){
        V[0]=min;
        V[1]=max;
        lato=l;
    }
    //funzione per l'inizializzazione del primo Box
    void setObjects(Obj** o,int nO){
        objects=o;
        nObj=nO;
    }
    //distruttore
    ~Box(){
        if(leaf1!=NULL){
            delete leaf1;
        }
        if(leaf2!=NULL){
            delete leaf2;
        }
    }
```

Il BSP (Binary Space Partition) viene quindi creato a partire dal Bounding Box, ovvero dal Box definito dal minimo e il massimo punto della scena e che quindi la contiene interamente.

```
<<srt.cpp>>
<<funzione main>>
```

```
//viene creato il Bounding Box (primo elemento dell'albero)
Bound=new Box(min, max, l);
```

```
//si carica l'array degli oggetti all'interno del Bounding Box:
Bound->setObjects(objects,n0);
```

Il parametro `l` gestisce il piano che divide il Box nei due Box figli. Quindi questi sono divisi ciclicamente dai piani:

: XY corrispondente a `l`=0
: YZ corrispondente a `l`=1
: XZ corrispondente a `l`=2

a partire da `l`=0.

```
<<srt.cpp>>
```

```
//funzione per la creazione dell'albero binario
void makeChild(Box* B){

    float3 min= B->V[0];
    float3 max= B->V[1];
    int l= B->lato;

    //si divide a metà il Box con il piano z=(min.z+max.z)/2
    if(l==0){
        B->leaf1= new Box(min,float3(max.x,max.y,(min.z+max.z)/2),1);
        B->leaf2= new Box(float3(min.x,min.y,(min.z+max.z)/2),max,1);
    }
    //si divide a metà il Box con il piano x=(min.x+max.x)/2
   if(l==1){
        B->leaf1= new Box(min,float3((min.x+max.x)/2,max.y,max.z),2);;
        B->leaf2= new Box(float3((min.x+max.x)/2,min.y,min.z),max,2);
    }
    //si divide a metà il Box con il piano y=(min.y+min.y)/2
   if(l==2){
        B->leaf1= new Box(min,float3(max.x,(min.y+max.y)/2,max.z),0);;
        B->leaf2= new Box(float3(min.x,(min.y+max.y)/2,min.z),max,0);
    }
}
```

La partizione dei Box viene operata da un metodo ricorsivo,
`setPartition(int liv, Box* b)`,
che ferma il procedimento quando il parametro `liv` raggiunge il massimo livello di profondità `depth` (definito nel `main.h`) o il puntatore al Box diventa nullo o ancora il Box in esame ha un numero di oggetti all'interno minore della `sogliaBox` (anche questa soglia è definita nel `main.h`). Ad ogni ricorsione il metodo crea due figli del Box e quindi aumenta il livello (`liv`) di profondità dell'albero. A questo punto la funzione `setLeafObj()` all'interno della struct `Box` ripartisce gli oggetti del Box padre nei Box figli.

```
<<srt.cpp>>
<<funzione main>>
```

```
//crea il tree

setPartition(liv,Bound);
```

```
<<srt.cpp>>
```

```
//Metodo ricorsivo che crea una partizione spaziale della scena e ripartisce gli
    oggetti di un Box padre tra i suoi due Box figli.
//Il metodo continua fino a quando non si raggiunge la generazione depth
//La variabile liv viene passata alla funzione come Reference e corrisponde al
    livello di profondità all'interno dell'albero
void setPartition(int& liv,Box* b){

    //la procedura continua solo se il Box non è null, il livello di profondità non
        ha superato il parametro depth e se ci sono almeno un numero di oggetti
        maggiore del valore di sogliaBox (definito nel main.h) dentro al Box

    if(((liv)<depth)&&(b!=NULL)&&(b->nObj>sogliaBox)){

        //aumentiamo il livello di profondità dell'albero:
        (liv)++;

        //creiamo i figli del Box padre
        makeChild(b);
        //assegniamo ai figli (foglie dell'albero) gli oggetti appartenti al Box
            padre:
        b->setLeafObj();

        //continuiamo la partizione iterando il procedimento finchè non si
            raggiunge il massimo livello di profondità ammesso:
        setPartition(liv,b->leaf1);
        setPartition(liv,b->leaf2);

//una volta completata la ripartizione tra i figli, ritorniamo al livello iniziale:
        liv--;
    }
}
```

Il metodo che si occupa di dividere gli oggetti tra i vari Box è quindi `setLeafObj`, esso deve effettuare questa operazione sia per i triangoli che per le sfere. Se il triangolo o la sfera è presente in più di un Box esso deve essere duplicato, per questo motivo non è conveniente cercare di racchiudere pochi oggetti all'interno di un Box. In questo caso, infatti, il numero di oggetti aumenterebbe notevolmente rispetto all'inizio e il metodo avrebbe l'effetto di rallentare, invece di velocizzare, il processo. La suddivisione, dei triangoli, all'interno dei Box si basa sulla posizione del vertice e sulla variabile lato del Box (la quale definisce il piano di separazione del Box). Presentiamo di seguito il codice:

```
<<geometry.h>>
<<struct Box>>
```

```
// si dividono gli oggetti del Box tra i suoi figli basandosi sugli oggetti che
    questo contiene
void setLeafObj(){

    //creazione di due array dinamici
    std::vector<Obj*> leaf1_Objects;
    std::vector<Obj*> leaf2_Objects;

    //per ogni oggetto presente nel Box padre
    for(int i=0;i<nObj;i++){
            //si vuole scoprire in quale leaf è l'oggetto i:
            //flag primo figlio leaf1
            bool inleaf1=false;
            //flag secondo figlio leaf2
            bool inleaf2=false;

            //se l'oggetto è un triangolo
            if(objects[i]->t!=NULL){
            //si carica il triangolo
            Triangle* t = objects[i]->t;
            //per ogni vertice
            for (int j=0; j<3; j++){
                //se il Box è stato dimezzato rispetto all'asse z
                //ciò che discriminerà in quale leaf stà il vertice è la cordinata z
                    (le altre resteranno uguali)
                if(lato==0){
                        if(t->vertices[j].z < leaf1->V[1].z){
                        //il vertice stà nel Box leaf1
                        inleaf1=true;
                        }else {
                        //altrimenti il vertice stà nel Box leaf2
                        inleaf2=true;
                        }
                    }
                //se il Box è stato dimezzato rispetto all'asse x
                    if(lato==1){
                            if(t->vertices[j].x<leaf1->V[1].x){
                                inleaf1=true;
                            }else {
                                inleaf2=true;
                            }
                     }
                //se il Box è stato dimezzato rispetto all'asse y
                    if(lato==2){
                    if(t->vertices[j].y<leaf1->V[1].y){

                        inleaf1=true;
                        }else {
```

```
                    inleaf2=true;
                    }
                }
            }
        }
    // se l'oggetto è una sfera si procede diversamente:
    if(objects[i]->s!=NULL){

        //si carica la sfera
        Sphere* s = objects[i]->s;
        //si carica il raggio della sfera
        float rad= s->rad;
        if(lato==0){
            if(s->p.z-rad<leaf1->V[1].z){
                inleaf1=true;
            }
            if(s->p.z+rad>leaf1->V[0].z)
            {
                inleaf2=true;
            }
        }
        //se il Box è stato dimezzato rispetto all'asse x
        if(lato==1){
            if(s->p.x-rad<leaf1->V[1].x){
                inleaf1=true;
            }
            if(s->p.x+rad>leaf1->V[0].x)
            {
                inleaf2=true;
        }
    }
        //se il Box è stato dimezzato rispetto all'asse y
        if(lato==2){
            if(s->p.y-rad<leaf1->V[1].y){
                inleaf1=true;
            }
            if(s->p.y+rad>leaf1->V[0].y)
            {
                inleaf2=true;
            }
        }
    }
    //si carica l'oggetto all'interno dell'array del suo Box
    //nel caso che questo fosse presente in entrambi i Box, esso sarà caricato
        due volte
    if(inleaf1) { leaf1_Objects.push_back(objects[i]); }
    if(inleaf2) { leaf2_Objects.push_back(objects[i]); }
    }
//una volta conosciuta la grandezza degli array dei Box figli
```

```
    //si possono creare della giusta dimensione abbandonando l'array dinamico
    //questo permette di effettuare una più efficente gestione della memoria
    int nO=(int)leaf1_Objects.size();
    leaf1->nObj=nO;
    leaf1->objects=new Obj*[nO];

    for(int i=0; i<nO;i++){
        leaf1->objects[i]=leaf1_Objects[i];
    }
    nO=(int)leaf2_Objects.size();
    leaf2->nObj=nO;
    leaf2->objects=new Obj*[nO];
    for(int i=0; i<nO;i++){
        leaf2->objects[i]=leaf2_Objects[i];
    }
    //si libera la memoria allocata dal Box padre in modo che solo le ultime foglie
        dell'albero contengano gli oggetti
    delete[] objects;
}
```

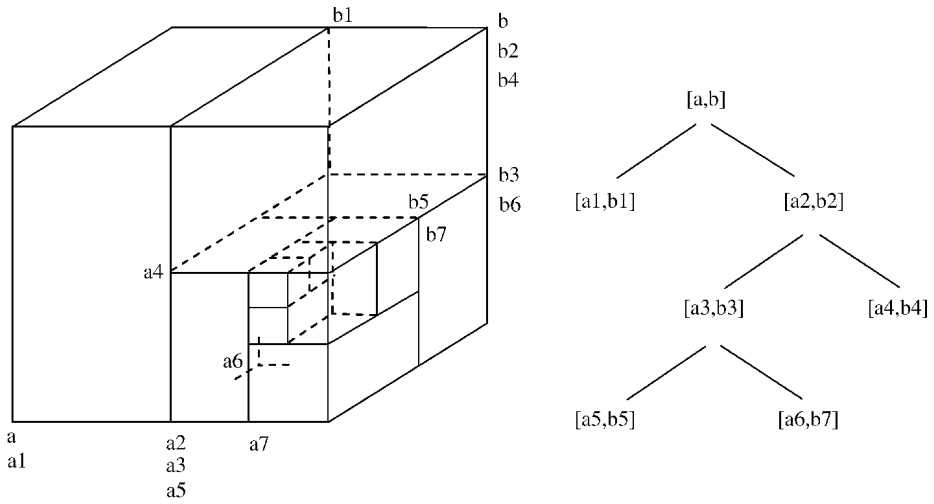


FIGURA 12.1.2. **Binary space partition.**

12.2. Kd-Tree

il Kd-Tree è un caso particolare del BSP, nel quale, per ogni Box, viene scelto il piano di divisione attraverso la mediana dei punti che esso contiene. Nel programma è utilizzato per il salvataggio della photon map, ossia della distribuzione degli hits di fotoni (vedi 11.3.2.2 a pagina 451). Al fine di velocizzare le operazioni, abbiamo scelto di inserire l'albero dentro un unico array, indicizzando i figli di ogni Box `i` come `2*i` per il primo figlio e `(2*i)+1` per il secondo. La struct adibita a questa funzione si chiama `photonBox`. Inseriamo qui il codice predisposto alla creazione del Kd-Tree.

```
<<srt.cpp>>
<<funzione main>>
```

```
// creazione della photon map
   if((photonMap)||(multiPassPhotonMap)){

       P=0;
       //viene calcolato il numero dei photonBox in base al parametro Kdepth
       for(int i=1; i<Kdepth; i++){
           P+=pow(2,i);
       }

      //creazione dell'array di photonBox:
       KdTree = new photonBox*[P];
       //inizializzazione parametro per il bilanciamento del Kd-Tree
       liv=0;
       //Vector che conterranno (solo inizialmente) i fotoni globali e quelli per
           le caustiche
       std::vector<photon*> photons;
       std::vector<photon*> Causticphotons;

       //si emettono i fotoni a partire dalle luci della scena
       emitPhotons(photons);
       //numero di fotoni nella Photon Map:
       int dim1=(int)photons.size();
       //se il parametro nPhotonSearch è pari a 0 allora viene utilizzato il
           parametro P_accuracy per stabilire il numero di fotoni
       //da utilizzare per la ricerca
       if(nPhotonSearch==0){
           nPhotonSearch=(int)floor(pow(dim1,P_accuracy));
       }
       //Si inizializza l'array dei fotoni utilizzato al posto degli oggetti Vector
       //in questo modo si evita il partizionamento della memoria
       photonArr= new photon*[dim1];
       for(int i=0; i<dim1; i++){
           photonArr[i]=photons[i];
       }
       //creazione del Kd-Tree per la Photon Map globale
       KdTree[0]=new photonBox(min,max,photonArr,dim1);
       //bilanciamento dell'albero
       Balance(KdTree,1,liv);

       //si effettua lo stesso procedimento per la Caustic Map
       // i fotoni saranno caricati in un altro KdTree di nome causticTree
       if(causticPhoton>0){
           causticTree= new photonBox*[P];
           pbn=0;
           caustic(Causticphotons);
           int dim2=(int)Causticphotons.size();

           if(nCausticSearch==0){
```

```
            nCausticSearch=(int)floor(pow(dim2,C_accuracy));
        }
        causticArr= new photon*[dim2];

        for(int i=0; i<dim2; i++){
            causticArr[i]=Causticphotons[i];
        }
        causticTree[0]=new photonBox(min,max,causticArr,dim2);
        Balance(causticTree,1,liv);
    }
}
```

Nel Kd-Tree la divisione di un Box con un piano avviene in base alla posizione dei fotoni al suo interno e non più ciclicamente come nel BSP. In primo luogo si calcola il bounding Box dei fotoni, cioè il minimo e massimo punto in cui è possibile trovare un fotone all'interno della `photonBox`. Il piano più adatto per suddividere il bounding Box è quello che lo divide nel suo lato più lungo. Il parametro `dim` conserva la scelta dell'asse corrispondente a questo lato:

: asse X corrisponde a dim=0
: asse Y corrisponde a dim=1
: asse Z corrisponde a dim=2

Occorre infine calcolare la posizione del piano rispetto all'asse scelto, la quale viene assegnata alla variabile `planePos`: per far questo si utilizza la mediana tra i fotoni del Box. Si procede quindi all'ordinamento dell'array dei fotoni in base alla loro posizione sull'asse; il punto mediano corrisponde alla posizione del fotone a metà dell'array ordinato. In questo modo ci si assicura che i fotoni siano suddivisi equamente tra i due Box. La posizione del piano corrisponde quindi alla componente `dim` della mediana.

```
<<geometry.h>>
```

```
//KdTree per l'accellerazione della ricerca dei fotoni all'interno della scena
struct photonBox{

    //vertici del Box
    float3 V[2];
    //array di fotoni
    photon** ph=NULL;
    int nph=0;
    //dimensione corrispondente alla normale del piano con cui viene suddiviso il
        Box
    int  dim;
    //posizione del piano lungo la dimensione dim
    float planePos;
    photonBox(){nph=0; ph=NULL;}

    //costruttore del Box di fotoni:
    photonBox(float3 v1, float3 v2,photon** p,int n){
        //si inizializzano le variabili del Box
        V[0]=v1;
```

```
V[1]=v2;
ph=p;nph=n;
//se il Box non è vuoto si continua il processo
if(n>0){
float3 max;
float3 min;
float3 median;

//viene calcolato il bounding Box dei fotoni:

for(int i=0; i<nph; i++){
    //si carica la posizione del fotone i:
    float3 p= ph[i]->position;
    //si controlla se il vertice è il massimo (se lo è lo si imposta come
       massimo)
    if(p.x>max.x)max.x= p.x;
    if(p.y>max.y)max.y= p.y;
    if(p.z>max.z)max.z= p.z;
    //si controlla se il vertice è il minimo (se lo è lo si imposta come
       minimo)
    if(p.x<min.x)min.x= p.x;
    if(p.y<min.y)min.y= p.y;
    if(p.z<min.z)min.z= p.z;
}
//si sceglie il parametro dim in base al lato del bounding Box più lungo
//per far questo si calcola la distanza tra il punto minimo e il punto
   massimo del bounding Box dei fotoni
float3 d= max-min;
d.abs();
float dist[]={d.x,d.y,d.z};

//confronto delle distanze per ciascun asse xyz
dim=0;
if(d.y>dist[dim]){dim=1;}
if(d.z>dist[dim]){dim=2;}

//calcolo della mediana:

//prima fase: ordinamento dell'array dei fotoni in base alla loro posizione

//si scorrono tutti gli elementi dell'array dei fotoni saltando il primo
for(int i=1; i<nph; i++){
    //si salva il fotone in esame da una parte
    photon* p=ph[i];
    //la posizione viene inserita in un array di 3 elementi
    float pos_i[]= {p->position.x,p->position.y,p->position.z};
    //si controlla l'elemento precedente al fotone in esame
    int j = i-1;
    //si carica l'elemento precedente in un array di 3 elementi
```

```
            float pos_j[]={ph[j]->position.x,ph[j]->position.y,ph[j]->position.z};

            //se non è stata controllata tutta la lista ordinata e l'elemento in
                posizione j è più grande di quello in i allora i due fotoni vengono
                scambiati
            while ((j >= 0) && (pos_j[dim]>pos_i[dim])){

                //scambio:
                ph[j + 1] = ph[j];
                //si scorrono gli elementi
                j = j-1;
                //si effettuano queste operazioni solamente se la lista ordinata non
                    è stata scorsa completamente:
                if(j>=0){
                pos_j[0]=ph[j]->position.x;
                pos_j[1]=ph[j]->position.y;
                pos_j[2]=ph[j]->position.z;}
                //l'elemento che prima era in posizione j ora è p
                ph[j+1]=p;
                }
        }

        //fase 2: calcolo della mediana

        //si cerca il fotone a metà dell'array appena ordinato in base alla distanza
        int mpos= floor(nph/2);
        //si assegna alla mediana la posizione di questo fotone
        median.copy(ph[mpos]->position);
        //si può quindi assegnare il valore della mediana che ci interessa m[dim]
            come posizione del piano che suddivide il Box
        float m[]={median.x,median.y,median.z};
        planePos=m[dim];
        }
    }

    //distruttore
    ~photonBox(){
        if(ph!=NULL){
        delete[] ph;
        }
    }

};
```

Non ci resta che vedere come avviene il bilanciamento del Kd-Tree, quindi come si creano le varie `photonBox` e come i fotoni sono loro assegnati. La funzione ricorsiva, che ha il compito di effettuare questo bilanciamento, è `Balance(Tree,index,liv)`. In essa viene utilizzato l'oggetto `std::Vector`, che ci permette una più pratica allocazione dinamica della memoria. Una volta che i fotoni sono stati divisi tra i Box, però, sono caricati all'interno di un array che ci permette di accedere alla memoria

in maniera più veloce. La creazione dell'albero si arresta automaticamente una volta superata la profondità definita nel main, `Kdepth`. Inseriamo qui il codice relativo al metodo:

<<srt.cpp>>

```
//funzione ricorsiva per il bilanciamento del Kd-Tree
void Balance(photonBox** Tree,int index,int liv){

    //si aumenta il livello di profondità del Kd-Tree
    liv++;
    //il metodo si ferma una volta raggiunta la profondità massima Kdepth
    if(liv<Kdepth){
    //si caricano i dati relativi al photonBox in posizione [index-1]
    int dim= Tree[index-1]->dim;
    float median= Tree[index-1]->planePos;
    float n= Tree[index-1]->nph;
    photon** ph=Tree[index-1]->ph;

    float3 min= Tree[index-1]->V[0];
    float3 max= Tree[index-1]->V[1];
    //variabili che conterranno i fotoni dei due photonBox finali
    photon** ph1;
    photon** ph2;
    //variabili Vector che conterranno temporaneamente i fotoni da dividere nei due
        Box
    std::vector<photon*> ph1_v;
    std::vector<photon*> ph2_v;

    //in base al valore dim, il photonBox in posizione [index-1] viene diviso in
        due nuovi Box da un piano posizionato in median
    //Asse X:
    if(dim==0){
        //suddivisione dei fotoni in base al piano:
        for(int i=0; i<n; i++){
            if(ph[i]->position.x<median){
                ph1_v.push_back(ph[i]);
            }else{ ph2_v.push_back(ph[i]);}
        }
        //fotoni del primo photonBox
        int n1=(int)ph1_v.size();
        ph1=new photon*[n1];
        //fotoni del secondo photonBox
        int n2=(int)ph2_v.size();
        ph2=new photon*[n2];
        //si caricano i fotoni dentro un array a memoria non partizionata
        for(int i=0; i<n1;i++){
            ph1[i]=ph1_v[i];
        }
        for(int i=0; i<n2;i++){
```

```
            ph2[i]=ph2_v[i];
        }
        //creazione del primo photonBox
        Tree[(2*index)-1]= new photonBox(min,float3(median,max.y,max.z),ph1,n1);
        //creazione del secondo photonBox
        Tree[(2*index)]= new photonBox(float3(median,min.y,min.z),max,ph2,n2);

        //si continua il bilanciamento con i Box appena creati:
        Balance(Tree,2*index,liv);
        Balance(Tree,(2*index)+1,liv);

    }
    //Asse Y:
    if(dim==1){

        for(int i=0; i<n; i++){
            if(ph[i]->position.y<median){
                ph1_v.push_back(ph[i]);
            }else{ ph2_v.push_back(ph[i]);}
        }
        int n1=(int)ph1_v.size();
        ph1=new photon*[n1];

        int n2=(int)ph2_v.size();
        ph2=new photon*[n2];

        for(int i=0; i<n1;i++){
            ph1[i]=ph1_v[i];
        }
        for(int i=0; i<n2;i++){
            ph2[i]=ph2_v[i];
        }

        Tree[(2*index)-1]= new photonBox(min,float3(max.x,median,max.z),ph1,n1);
        Tree[2*index]=new photonBox(float3(min.x,median,min.z),max,ph2,n2);
        Balance(Tree,2*index,liv);
        Balance(Tree,(2*index)+1,liv);
    }
    //Asse Z:
    if(dim==2){

        for(int i=0; i<n; i++){
            if(ph[i]->position.z<median){
                ph1_v.push_back(ph[i]);
            }else{ ph2_v.push_back(ph[i]);}
        }
        int n1=(int)ph1_v.size();
        ph1=new photon*[n1];
```

```
        int n2=(int)ph2_v.size();
        ph2=new photon*[n2];

        for(int i=0; i<n1;i++){
            ph1[i]=ph1_v[i];
        }
        for(int i=0; i<n2;i++){
            ph2[i]=ph2_v[i];
        }

        Tree[(2*index)-1]= new photonBox(min,float3(max.x,max.y,median),ph1,n1);
        Tree[2*index]= new photonBox(float3(min.x,min.y,median),max,ph2,n2);
        Balance(Tree,2*index,liv);
        Balance(Tree,(2*index)+1,liv);
    }
    }
}
```

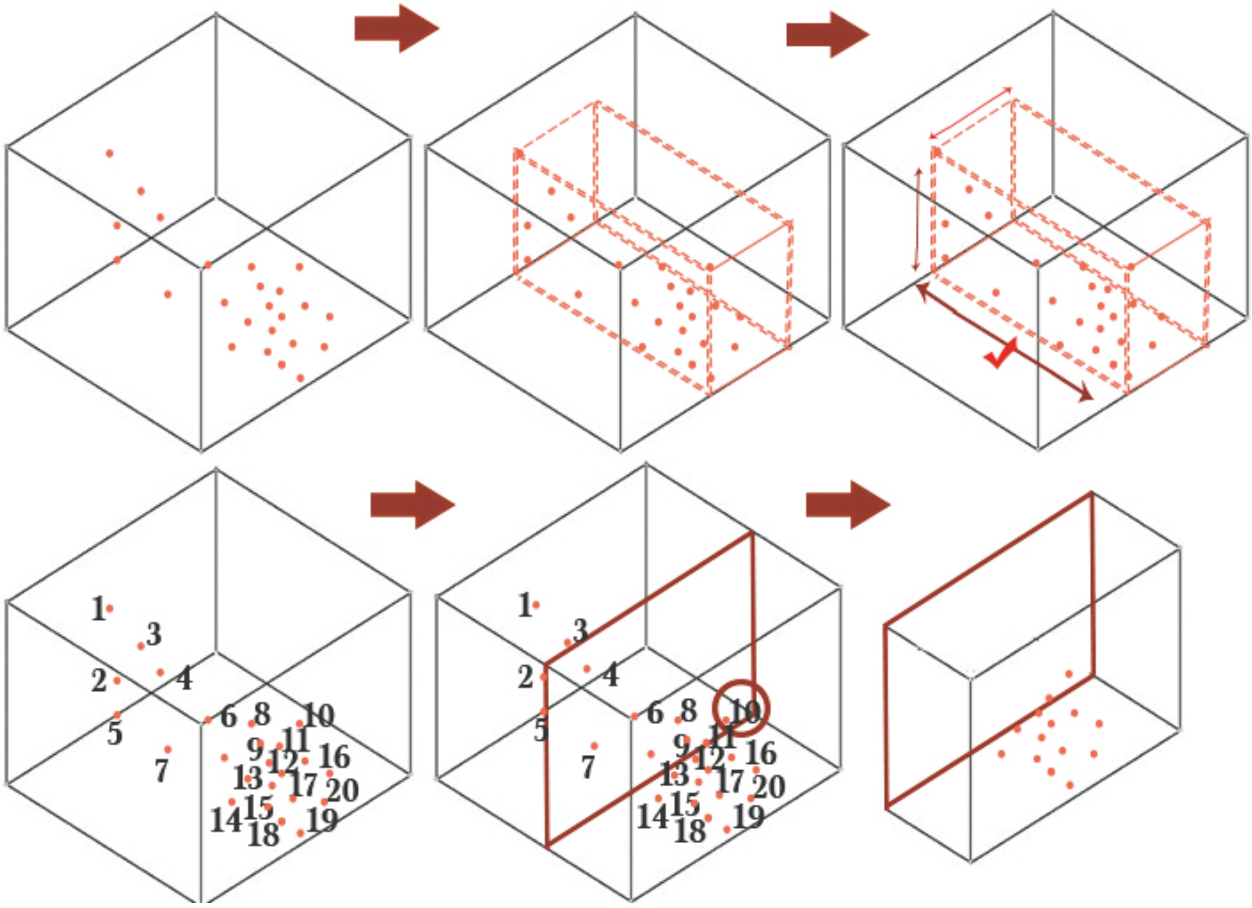


FIGURA 12.2.1. **Kd-Tree.**

12.2.1. Ricerca dei fotoni nel Kd-Tree. La funzione ricorsiva adibita alla ricerca dei fotoni è
`locate_photons(nearPh,iP,index,objX,Tree,d_2,nph)`,
le cui variabili sono:

- `iP`: punto di intersezione con la scena di cui bisogna approssimare l'illuminazione.
- `nearPh`: oggetto `sta::map` utilizzato allo scopo di raccogliere i fotoni intorno ad iP. Esso inoltre ordina automaticamente i fotoni caricati, rispetto alla distanza dal punto osservato.
- `Tree`: Kd-Tree contenente i fotoni.
- `index`: posizione del nodo nel Kd-Tree.
- `objX`: oggetto intersecato; necessario per scegliere il metodo di ricerca del fotone. In questo modo la ricerca sarà sferica su sfere e planare su triangoli.
- `d_2`: distanza massima di ricerca del fotone al quadrato.
- `nph`: numero di fotoni da ricercare.

Il metodo verifica preliminarmente se l'indice `index` corrisponde ad un Box all'estremità dell'albero; se così non è, si calcola la distanza del punto cercato dal piano di divisione del Box. La ricerca continua solo nei Box la cui distanza è minore della variabile `d_2`. Una volta raggiunto un Box all'estremità dell'albero si caricano tutti i fotoni al suo interno, al fine di verificare la distanza dal punto osservato. Solamente gli `nph` fotoni più vicini sono caricati all'interno di `nearPh`; nel caso in cui questo numero viene superato si devono cancellare gli elementi in eccesso. Alla distanza massima di ricerca, `d_2`, viene assegnata la distanza del fotone più lontano al punto osservato. In questo modo si considerano solo i fotoni più vicini rispetto agli `nph` fotoni già caricati. Nella funzione inoltre si fa distinzione tra i triangoli, in cui la ricerca può essere anche planare, e le sfere, in cui è solamente sferica.

<<srt.cpp>>

```
//funzione adibita alla ricerca dei fotoni attorno ad un punto iP:
void locate_photons(std::map< float ,photon*>& nearPh,float3& iP,int index,Obj*
    objX,photonBox** Tree,float& d_2,int nph){

//si verifica che il Box index non sia vuoto
if(Tree[index-1]->nph!=0){

    //si verifica che il nuovo indice non abbia ha superato la lunghezza
       dell'albero (P), in tal caso l'indice corrisponde ad un Box all'estremità
       dell'albero
    if((2*index)+1< P){

        //viene caricato il punto di intersezione in un array di 3 elementi
        float pos[]={iP.x,iP.y,iP.z};
        //si calcola la distanza del punto iP dal piano di divisione del nodo in
           esame
        float delta=pos[Tree[index-1]->dim]-Tree[index-1]->planePos;
        //si verifica che il punto sia contenuto nella foglia sinistra
        if(delta<0){

            //foglia sinistra:
            //si continua la ricerca dei fotoni
```

```
        locate_photons(nearPh,iP,2*index,objX,Tree,d_2,nph);
        //se la distanza dal piano di divisione è minore della distanza massima
            di ricerca bisogna continuare la ricerca anche nel Box adiacente
        if(pow(delta,2)<d_2){

            //foglia destra:
            //si continua la ricerca dei fotoni
            locate_photons(nearPh,iP,2*index+1,objX,Tree,d_2,nph);
        }
    }else{

        //foglia destra (stesso procedimento invertito della foglia sinistra):
        locate_photons(nearPh,iP,2*index+1,objX,Tree,d_2,nph);
        if(pow(delta,2)<d_2){
                locate_photons(nearPh,iP,2*index,objX,Tree,d_2,nph);
           }
       }
}else{

    //se ci si trova all'estremità dell'albero:
    //si caricano tutti i fotoni del Box dentro alla variabile p
    int n=Tree[index-1]->nph;
    photon** p= Tree[index-1]->ph;
    //per ogni fotone
    for(int i=0;i<n;i++){

        //si calcola la distanza del fotone dal punto di intersezione
        float3 dist=p[i]->position-iP;
        //si verifica che l'oggetto sia un triangolo
        if(objX->t!=NULL){

        //se l'oggetto è un triangolo la ricerca sarà planare:
        //calcolo del versore relativo alla distanza del fotone dal punto
            osservato
        float3 d= dist*(1/dist.normval());
        //si calcola la proiezione del versore rispetto alla normale
            dell'oggetto objX
        //questo valore corrisponde al coseno tra i due versori
        float projN = d.dot(objX->normal(iP).norm());
        //valore massimo di ricerca rispetto al coseno tra il fotone e la
            normale
        float val=sphericalSearch+EPS;
        //il metodo continua solo se projN è compreso in [-val,val]
        if((projN<=val)&&(projN>=-val)){

        //norma al quadrato della distanza
        float d2p=dist.norm2val();
          //si verifica che la distanza sia minore della distanza massima di
              ricerca
```

```
                if(d2p<d_2){

                    //si carica il fotone nell'oggetto nearPh
                    nearPh[d2p]=p[i];
                    //se si supera il numero massimo di fotoni da ricercare
                    if(nearPh.size()>nph){

                        //si cancella il fotone più lontano
                        nearPh.erase(--nearPh.end());
                        //viene assegnata la distanza dell'ultimo elemento
                            all'interno dell'oggetto nearPh alla distanza massima di
                            ricerca
                        d_2=(--nearPh.end())->first;
                    }
                }
            }
            }else if(objX->s!=NULL){

                //se l'oggetto è una sfera la ricerca sarà sferica:
                float d2p=dist.norm2val();
                if(d2p<d_2){
                    nearPh[d2p]=p[i];
                    if(nearPh.size()>nph){
                        nearPh.erase(--nearPh.end());
                        d_2=(--nearPh.end())->first;
                    }
                }
            }
        }
    }
 }
}
```

FIGURA 12.2.2. **Rendering, ottenuto direttamente dalla mappa fotonica, di una scena con 2715347 triangoli e 1000000 di fotoni:** questo risultato, ottenuto con 1 solo campione per ciascun pixel, ha richiesto 4 ore e 58 minuti, il parametro `Kdepth` è stato impostato a 15.

CAPITOLO 13

Il codice nel dettaglio (in una versione semplificata di C++)

In questo capitolo vogliamo presentare il funzionamento e il codice del nostro renderer. Per facilitare la comprensione degli argomenti anche a chi non è pratico del linguaggio C++, si evita, all'interno del programma, di utilizzare concetti ad esso troppo legati (come ad esempio l'ereditarietà e le classi).

13.1. Struct Obj

La struttura `Obj` si occupa di descrivere la geometria della scena, qualunque essa sia. Nel programma ne sono implementate due: il triangolo e la sfera. Possiamo definire l'oggetto `Obj` come loro "padre", in quanto in esso sono presenti i metodi e le variabili che le due strutture condividono. Integriamo di seguito con il codice relativo a questa struttura:

```
<<geometry.h>>
```

```
//La struct Obj funge da "padre" delle varie geometrie presenti nel programma
    (Triangle e Sphere)
//grazie ad essa i vari algoritmi agiscono ugualmente su ogni geometria.
struct Obj{

    //puntatori alla sfera o al triangolo:
    Sphere* s=NULL;
    Triangle* t=NULL;
    //bounding box dell'oggetto, può essere utilizzato per velocizzare la fase di
        intersect
    float3 min;
    float3 max;

    //area dell'oggetto:
    float areaObj=0;

    //potenza emessa dall'oggetto:
    //viene utilizzata all'interno dell'algoritmo radiosity
    float3 P=float3(0);
    //indice del materiale:
    int matId;

    Obj(){

        s=NULL;
        t=NULL;}
```

```
//si creano due tipi di costruttori a seconda che si voglia costruire un
    triangolo o una sfera
//costruttore sfera:
Obj(Sphere* s,int matId):s(s), matId(matId){

    t=NULL;
    //calcolo dell'area dell'oggetto:
    areaObj=area();
   //calcolo il bounding box dell'oggetto:
    min=s->p-float3(s->rad,s->rad,s->rad);
    max=s->p+float3(s->rad,s->rad,s->rad);
 }
//costruttore triangolo:
Obj(Triangle* t,int matId):t(t), matId(matId){

    s=NULL;
    //calcolo l'area dell'oggetto
    areaObj=area();

    //calcolo del bounding box dell'oggetto:
    max=t->vertices[0];
    min=t->vertices[0];
    //per ogni vertice
    for(int j=1; j<3; j++){

        //si trova il massimo tra tutti i vertici
        if(t->vertices[j].x>max.x)max.x= t->vertices[j].x;
        if(t->vertices[j].y>max.y)max.y= t->vertices[j].y;
        if(t->vertices[j].z>max.z)max.z= t->vertices[j].z;
        //si trova il minimo tra tutti i vertici
        if(t->vertices[j].x<min.x)min.x= t->vertices[j].x;
        if(t->vertices[j].y<min.y)min.y= t->vertices[j].y;
        if(t->vertices[j].z<min.z)min.z= t->vertices[j].z;
        }
    }
    //si imposta il materiale dell'oggetto
    void setMaterial(int m){
        matId=m;
    }
    //distruttore
    ~Obj(){
        delete s;
        delete t;
    }
    //metodo per l'utilizzo della normale alla superficie dell'oggetto:
    //iP corrisponde al punto di intersezione con la superficie
    float3 normal(float3 iP){
        if(s){
            return s->normal(iP);
```

```
    }
    if(t){
        return t->normal();
    }
    else{ return float3(0.0f); }
}
//metodo per l'utilizzo di un punto scelto uniformemente all'interno della
    superficie dell'oggetto:
//rnd1,rnd2,rnd3 sono variabili aleatorie in [0,1]
float3 randomPoint(float rnd1,float rnd2,float rnd3){
    if(s){
        float x=2*s->rad*rnd1-s->rad;
        float y=2*s->rad*rnd2-s->rad;
        float z=2*s->rad*rnd3-s->rad;
       // si utilizza la tecnica del rejection sampling
        while(x*x+y*y+z*z!=s->rad*s->rad){

            rnd1=generateRandom(aoSId[0],5,aST);
            rnd2=generateRandom(aoSId[0],6,aST);
            rnd3=generateRandom(aoSId[0],7,aST);
             x=2*s->rad*rnd1-s->rad;
             y=2*s->rad*rnd2-s->rad;
             z=2*s->rad*rnd3-s->rad;
        }
        return s->p+float3(x,y,z);
    }
    if(t){

    //si utilizzano le coordinate convesse
        float d=rnd1+rnd2+rnd3;
            rnd1/=d;
            rnd2/=d;
            rnd3/=d;
            return t->vertices[0]*rnd1+t->vertices[1]*rnd2
             +t->vertices[2]*(rnd3);
    }
    else{ return float3(0.0f);}
}
//metodo per il calcolo dell'area dell'oggetto:
float area(){
   if(s){
        return 4*M_PIf*s->rad*s->rad;
    }
    if(t){
        //si caricano i vertici del triangolo
        float3* v= t->vertices;
        //calcolo dell'area del triangolo
        float3 l1=(v[1]-v[0]);
        float3 l2=(v[2]-v[0]);
```

```
                //si utilizza l'altezza, ottenuta moltiplicando il lato per il seno
                   dell'angolo compreso tra i due vettori, per il calcolo dell'area
                float l1_norma=sqrtf(l1.x*l1.x+l1.y*l1.y+l1.z*l1.z);
                float l2_norma=sqrtf(l2.x*l2.x+l2.y*l2.y+l2.z*l2.z);
                float cosl1_l2=l1.dot(l2)/(l1_norma*l2_norma);
                float sinl1_l2=sqrtf(1-cosl1_l2*cosl1_l2);
                return l1_norma*sinl1_l2*l2_norma;
            }
            return 0;
        }
        //metodo intersect:
        //si richiama il metodo intersect della struct "figlia"
        float intersect(const Ray &raggio) const {
            if(s){
                return s->intersect(raggio);
            }
            if(t){
                return t->intersect(raggio);
            }else{
            return 0.0f;
            }
        }
};
```

13.1.1. Struct Sphere. La struttura della sfera racchiude al suo interno le variabili riguardanti il raggio e il centro di questa. In essa è contenuto, inoltre, il calcolo dell'intersezione di un raggio con la sfera, che necessita della risoluzione di un'equazione quadratica. La normale alla superficie nel punto `iP` è il versore del vettore che congiunge il centro della sfera con il medesimo punto.

```
<<geometry.h>>
```

```
//gemetria della sfera
struct Sphere {
   //raggio:
   float rad;
   //centro:
   float3 p;

    //costruttore:
   Sphere(float rad, float3 p): rad(rad), p(p){}
    //construttore default:
    Sphere():rad(0),p(float3(0.0f)){}

    //funzione di intersezione con un raggio:
    //se non c'è intersezione restituisce il valore -1
   float intersect(const Ray &r) const {
        //sostituendo il raggio o+td all'equazione della sfera
           (td+(o-p)).(td+(o-p)) -R^2 =0 si ottiene l'intersezione
```

```
      // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
      // A=d.d=1 (r.d is normalized)
      // B=2*(o-p).d
      // C=(o-p).(o-p)-R^2
      float3 op = p-r.o;
      // Distance
      float t;
      float B=op.dot(r.d); // 2*t*B -> simplified using b/2 instead of B
      float C=op.dot(op)-rad*rad;
        //determinante dell'equazione:
      float det=B*B-C;
        // se il determinante è negativo non c'è intersezione tra l'oggetto e il
            raggio altrimenti si calcola la radice
      if (det<0) return -1.0f; else det=sqrt(det);
        // si restituisce il risultato, relativo all'intersezione del raggio, t,se
            esso è maggiore di EPS, errore massimo considerato.
        //nel caso in cui questo non si verifichi: l'intersezione avviene nella
            direzione opposta a quella del raggio.
      return (t=B-det)>EPS ? t : ((t=B+det)>EPS ? t : -1.0f);
   }

    //si calcola la normale nel punto iP della sfera
    float3 normal(float3 iP){
        return (iP-p).norm();
    }
};
```

13.1.2. Struct Triangle. La struttura del triangolo contiene al suo interno l'array dei vertici e la normale ad esso. Il calcolo di quest'ultima, eseguito all'interno del costruttore del triangolo, è effettuato tramite il prodotto vettoriale rispetto ai due lati del triangolo.

$$a = v_1 - v_0 \quad e \quad b = v_2 - v_0\,,$$

dove v_i, $i = 0, 1, 2$, rappresenta il vertice i del triangolo. L'intersezione del triangolo con un raggio, inserita nel metodo `intersect` della struttura, si trova risolvendo un sistema di equazioni. Riportiamo qui di seguito il codice relativo a questa struttura:

```
<<geometry.h>>
```

```
//geometria del triangolo:
struct Triangle{
   // array dei vertici:
   float3 *vertices;
   // normale:
   float3 n;
    //costruttore di default:
    Triangle(){vertices=NULL; n=float3();}

    //costruttore:
```

```
 //i parametri in input sono i tre vertici del triangolo
Triangle(float3 v0, float3 v1, float3 v2){
    // triangolo definito da un array di 3 float3 :
  vertices = new float3[3];
  vertices[0]=v0;
  vertices[1]=v1;
  vertices[2]=v2;
    //calcolo della normale:
    //si calcolano i vettori dei due lati del triangolo a e b
  float3 a=v1-v0;
  float3 b=v2-v0;
    //si calcola il vettore normale tramite prodotto vettoriale di a e b,
       normalizzato.
  n=(a%b).norm();
}
 //distruttore
~Triangle(){
    delete[] vertices;
 }
 //funzione di intersezione con un raggio:
 //se non c'è intersezione restituisce il valore -1
float intersect(const Ray &r) const {
    //risolvo il sistema o+Td=a1+ beta(b1-a1) + gamma(c1-a1) => riscritto come
       [ beta(a1-b1)+gamma(a-c1)+Td1 = a-o ] dove T, beta e gamma sono le
       incognite, mentre a1 b1 e c1 sono i vertici del triangolo e d1 è la
       direzione del raggio.

    //componenti X:
    // (a1-b1).x
  double a = vertices[0].x-vertices[1].x;
    // (a1-c1).x
  double b=vertices[0].x-vertices[2].x;
    //d1.x
  double c=r.d.x;
    // (a-o).x
  double d=vertices[0].x-r.o.x;

    //componenti Y:
  double e=vertices[0].y-vertices[1].y;
  double f=vertices[0].y-vertices[2].y;
  double g=r.d.y;
  double h=vertices[0].y-r.o.y;

    //componenti Z:
  double i=vertices[0].z-vertices[1].z;
  double j=vertices[0].z-vertices[2].z;
  double k=r.d.z;
  double l=vertices[0].z-r.o.z;
```

```
        //ora si hanno tutte le componenti del sistema:
        // | a b   c  ||d|
        // | e f   g  ||h|
        // | i j   k  ||l|

        //risoluzione del sistema:
      double m=f*k-g*j;
      double n=h*k-g*l;
      double p=f*l-h*j;
      double q=g*i-e*k;
      double s=e*j-f*i;

      double inv_denom=1.0/(a*m+b*q+c*s);
      double e1=d*m-b*n-c*p;
      double beta=e1*inv_denom;
      if(beta<0.0){
         return(-1.0f);
      }
      double r1=e*l-h*i;
      double e2=a*n+d*q+c*r1;
      double gamma=e2*inv_denom;

      if(gamma<0.0){
         return(-1.0f);
      }
      if(beta+gamma>1.0){
         return(-1.0f);
      }
      double e3=a*p-b*r1+d*s;
      float t=e3*inv_denom;
      return t<EPS ? -1.0f : t;
   }
    //metodo per l'utilizzo della normale al triangolo:
    float3 normal(){
        return n;
    }
};
```

13.2. Struct Camera

Esistono due tipi di coordinate all'interno del programma:

- globali (*world coordinates*),
- locali rispetto al punto di osservazione (*local coordinates*).

Le coordinate globali sono necessarie per descrivere lo spazio e collocare gli oggetti all'interno della scena. La base ortonormale per cui sono definite è quella canonica:

$$e_1 = (1,0,0) \quad e_2 = (0,1,0) \quad e_3(0,0,1)$$

per mezzo di essa si possono ottenere, anche, la posizione e la direzione della camera, ossia il nostro punto di osservazione da cui viene costruita l'immagine. Tramite la camera si possono definire nuove

coordinate locali, che permettono di descrivere qualsiasi punto dello spazio a partire dall'osservatore. Ciò è possibile grazie ad una nuova base ortonormale posizionata al centro della camera. Presentiamo quindi il codice di questa struct:

<<camera.h>>

```
//modellazione della camera, punto di vista dell'osservatore
struct camera{
   /// posizione della camera
   float3 eye;
   /// direzione di visuale
   float3 lookAt;
   ///vettore Up per l'orientazione della camera (generalmente posto come asse Y)
   float3 up;
    //risoluzione dell'immagine:
   /// xResolution
   int width;
   /// yResolution
   int height;
   /// distanza dal sensore:
   float d;
   ///Vettori ortonormali della camera.
    ///Sono utilizzati peer le coordinate locali nello spazio, cioè rispetto al
        punto di vista dell'osservatore
   float3 U,V,W;
    //parametri relativi alla modellazione di una camera reale:
    float aperturaDiaframma = 0;
    float fuoco=0;
    //costruttore di default:
   camera(): eye(float3(0.0f)), lookAt(float3(0.0f)), up(float3(0.0f,0.0f,1.0f)),
       width(320), height(240), d(35.0f), U(float3(0.0f)), V(float3(0.0f)),
       W(float3(0.0f)){}
    //costruttore della camera:
    camera(float3 eye, float3 lookAt, float3 up, int width, int height, float d) :
        eye(eye), lookAt(lookAt), up(up), width(width), height(height), d(d),
        U(float3(0.0f)), V(float3(0.0f)), W(float3(0.0f)){
      // si crea il sistema ortonormale della camera
        //viene posto come vettore W, corrispondente alla coordinata z, la
            direzione di visuale dell'osservatore.
        //nella Computer Graphics però, per convenzione, è impostato con direzione
            opposta.
      W=eye-lookAt;
      W.norm();
        //si usa il prodotto vettoriale con up per ottenere il secondo vettore
            ortonormale a W
      U=(W*-1.0f)%up.norm();
      U.norm();
        //si crea infine l'ultimo vettore ortonormale che sarà, per costruzione,
            molto simile al vettore up.
```

```
      V=U%W;
   }
};
```

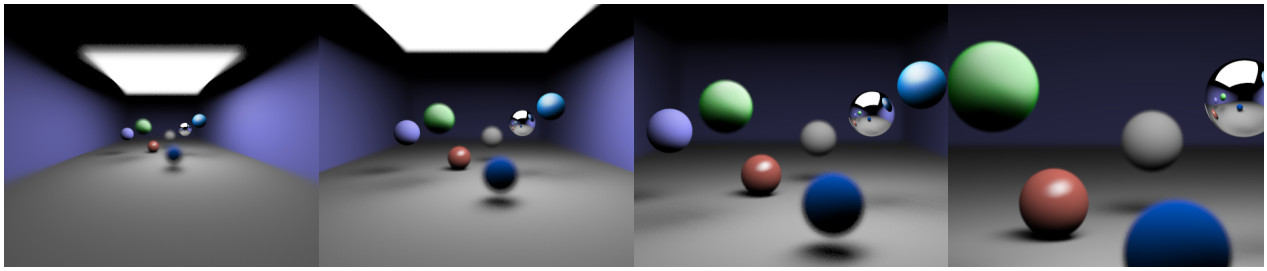


FIGURA 13.2.1. **variazione della distanza tra il centro di proiezione e il sensore**: il parametro della camera relativo è stato impostato, da sinistra a destra, come: 200, 400, 900, 1200. Notiamo che le prime immagini nella realtà corrispondo a dei grandangolari mentre le ultime a dei teleobiettivi. In queste immagini il parametro di apertura è stato posto uguale a 20.

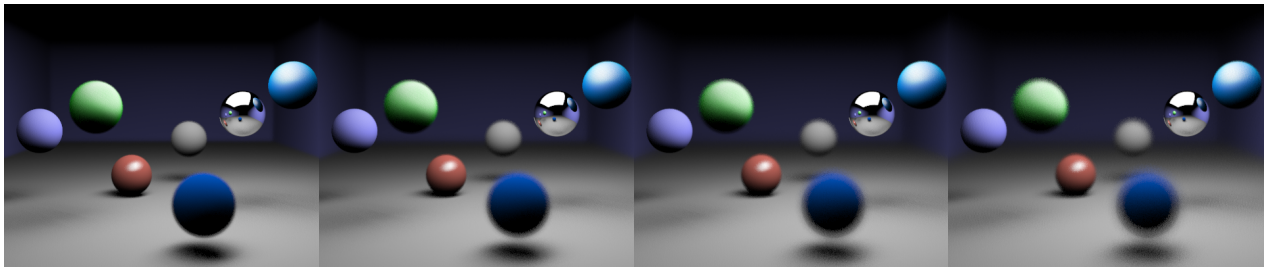


FIGURA 13.2.2. **variazione del parametro di apertura della camera**: il parametro della camera relativo è stato impostato, da sinistra a destra, come: 10, 20, 30, 40; Il numero di sample utilizzato è 50, la messa a fuoco è centrata sulla sfera riflettente.

13.3. Main

Si analizza in questa sezione come si struttura la funzione `main` nel nostro renderer. La scena che vogliamo visualizzare è composta da oggetti raggruppati nelle struct `Mesh`. In questa sono riportati il puntatore all'array degli oggetti e il numero di essi. Di seguito presentiamo il codice della struct `Mesh`:

```
<<geometry.h>>
```

```
//Struct adibita al raggruppamento degli oggetti
struct Mesh{
    //nome della mesh:
    char* nome;

    //array di puntatori agli oggetti della mesh:
    Obj** objects=NULL;
    //numero di oggetti presenti nella mesh:
```

```
int nObj=0;
//costruttore della Mesh:
Mesh(char* n, Obj** o,int nO){
    nome= n;
    objects=o;
    nObj=nO;
}
//si imposta il materiale per tutti gli oggetti della Mesh
void setMaterial(int m){
    for(int i=0; i<nObj;i++){
        objects[i]->setMaterial(m);
    }
}
//funzione valida solo per Mesh di triangoli:
//si suddividono i triangoli della Mesh in due, duplicando il numero di
    triangoli della Mesh
void suddividi(){
    //si crea il nuovo array dei triangoli
    Obj** objects2 =new Obj*[nObj*2];
    int n=0;
    for(int i=0; i<nObj; i++){
        //si verifica che l'oggetto sia un triangolo
        if(objects[i]->t){
            //si caricano i vertici del triangolo
            float3* v=objects[i]->t->vertices;
            //si caricano i lati del triangolo
            float3 l[]={ v[2]-v[0],v[0]-v[1],v[1]-v[2]};
            //si cerca il lato più lungo del triangolo
            int pos=0;
            if(l[1].normval()>l[pos].normval())pos=1;
            if(l[2].normval()>l[pos].normval())pos=2;
            //si carica il punto a metà del lato più lungo del triangolo
            float val=l[pos].normval()*0.5;
            //si trasforma il lato più lungo in un versore
            l[pos].norm();
            //si caricano i nuovi vertici del triangolo
            float3 nv[]={v[(pos+1)%3],v[(pos+2)%3],v[pos],l[pos]*val+v[pos]};
            //creazione dei due nuovi triangoli:
            objects2[n]=new Obj(new
                Triangle(nv[0],nv[1],nv[3]),objects[i]->matId);
            n++;
            objects2[n]=new Obj(new
                Triangle(nv[0],nv[3],nv[2]),objects[i]->matId);
            n++;
        }
        //si cancella il triangolo iniziale
        delete objects[i];
    }
    //si cancella l'array iniziale
```

```
        delete[] objects;
        //si scambia il nuovo array al posto del vecchio
        objects= objects2;
        nObj=nObj*2;
    }

    //distruttore della mesh:
    ~Mesh(){
        //si cancellano gli oggetti caricati
        for(int i=0; i<nObj;i++){
            delete objects[i];
        }
        //si cancella l'array
        delete[] objects;
    }
};
```

Una volta deciso il numero di Mesh da caricare le inseriamo dentro un array. All'interno di questo è sempre presente la Cornell Box, cioè la stanza che racchiude tutte le altre Mesh, per questa ragione il numero delle Mesh parte da 1. Nel programma è presente, inoltre, la possibilità di creare delle Mesh di triangoli a partire da file `obj`. Da questi estraiamo 2 importanti informazioni relative alla Mesh:

- l'array dei vertici `v`,
- gli indici `f` dell'array dei vertici tramite i quali si definiscono le facce triangolari,

Il metodo `CaricaMesh` si occupa di analizzare il file e caricare i triangoli all'interno delle Mesh. Esso carica ogni riga del file in un buffer di 100 byte, dopodiché verifica se in essa sono contenute le informazioni riguardo ad un vertice o ad un indice. In base a questa analisi carica i dati nei rispettivi array, che sono cancellati una volta creati i triangoli e gli oggetti di cui è composta la nuova Mesh. Di seguito inseriamo il codice relativo al caricamento della Mesh di triangoli:

```
<<srt.cpp>>
```

```
//Metodo per il caricamento di mesh da file .obj, questi devono contenere
    informazioni riguardo ai vertici e agli indici per creare dei triangoli
Mesh* CaricaMesh(char* nomefile, int mId){
    //array dei vertici :
    float3* vertex=NULL;
    //numero dei vertici:
    int nV=0;
    //file obj da aprire:
    FILE* f2;
    //array degli indici:
    float3* index=NULL;
    //numero di triangoli (ogni tre indici si costituisce un triangolo):
    int nT=0;
    //buffer di 100 byte in cui si carica ogni riga del file:
    char buffer[100];
    //se il file cercato è leggibile il metodo continua
```

```
if ((f2 = fopen (nomefile, "r"))!= NULL)
{
   //si caricano le righe del file all'interno del buffer
    while (fgets (buffer, sizeof (buffer), f2) != NULL)
    {
       //se la riga contiene un vertice:
           if((buffer[0]=='v')&&(buffer[1]!='t'))
           {
               float v[3];
              //si divide la riga in base agli spazi e si trasforma ogni
                 elemento in un float
               v[0]= atof(strtok(buffer+2," "));
               v[1]= atof(strtok(NULL," "));
               v[2]= atof(strtok(NULL," "));
               //viene aumentato il numero dei vertici
               nV++;
               //viene riallocata la memoria per l'array di vertici
               vertex=(float3*)realloc(vertex,(nV)*sizeof(float3));
               //si carica il nuovo vertice
               vertex[nV-1]=float3(v[0],v[1],v[2]);
           }
        //se la riga contiene un indice relativo alla faccia del triangolo
        if(buffer[0]=='f'){
           char* s[3];
            //si divide la riga in base agli spazi
            s[0]= strtok(buffer+2," ");
            s[1]= strtok(NULL," ");
            s[2]= strtok(NULL," ");
            //indice del vertice
            float iV[3];

            //si separano gli indici per l'array dei vertici dagli altri indici
            //ogni tipologia è separata dal carattere \ per cui si separano le
               nostre stringhe in base a questo carattere
            //primo indice
            iV[0]=atof(strtok(s[0],"\\") );
            //secondo indice
            iV[1]=atof(strtok(s[1],"\\") );
            //terzo indice
            iV[2]=atof(strtok(s[2],"\\") );
           //si aumenta il numero dei triangoli
            nT++;
            //inserimento nell'array degli indici:
            index=(float3*)realloc(index,nT*sizeof(float3));
            index[nT-1]=float3(iV[0],iV[1],iV[2]);
        }
    }
}
//si chiude il file obj
```

```
    fclose (f2);
    //inizializziamo gli array finali dei triangoli e degli oggetti
   Triangle** t=new Triangle*[nT];
   Obj** objects=new Obj*[nT];

 //caricamento degli array (i valori degli indici partono da 1 mentre l'array dei
     vertici da 0 per cui si deve effettuare uno slittamento di 1.
    for(int i=0; i<nT; i++){
        //creo il triangolo
        t[i]=new
            Triangle(vertex[(int)index[i].x-1],vertex[(int)index[i].y-1],vertex[(int)index[i].z-1]
        objects[i]=new Obj(t[i],mId);
    }
    //cancelliamo gli array che non sono più necessari
    delete[] vertex;
    delete[] index;
    //viene restituita la nuova mesh
    return new Mesh(nomefile,objects,nT);
}
```

Si carica un file `obj` nel programma inserendone il nome nell'array `filename` all'interno del «`main.h`»; l'indice ad esso associato è invece conservato in `matIdObj`.

```
<<srt.cpp>>
<<funzione main>>
```

```
//si caricano le mesh di triangoli attraverso dei file .obj
//i nomi di questi file sono definiti nel <<main.h>> e vengono caricati nell'array
    filename
//si carica invece il materiale di queste Mesh nell'array matIdObj
m=new Mesh*[nfile+1];
nMesh=nfile;
for(int i=0; i<nfile; i++){
            m[i]= CaricaMesh(filename[i],matIdObj[i]);
 }
```

Dalle Mesh possiamo ricavare il Bounding Box della scena, con cui creare la Cornell Box nella quale si eseguono i rendering. Per poter utilizzare al meglio il metodo di Jacobi la stanza (ovvero il box) deve avere una partizione in oggetti la più fitta possibile.

```
for(int q=0;q<sceneDepth;q++){
        m[nMesh]->suddividi();
    }
```

Tramite il parametro `sceneDepth` e il metodo `suddividi()` della struct `Mesh` possiamo infittire la descrizione della stanza arbitrariamente. Ora è necessario caricare i puntatori a tutti gli oggetti in un unico `std::Vector`, che sarà poi passato al nodo root del Binary-tree (si veda capitolo 12) per inserire gli oggetti della scena nei rispettivi box. Il calcolo dell'illuminazione viene eseguito pixel

per pixel con metodi di ray tracing. Il metodo di Jacobi e il metodo della photon map richiedono però una fase preliminare (si veda capitolo 11). Si inizia quindi il ciclo su ogni pixel, per ognuno dei quali possiamo scegliere un numero arbitrario di `samples`. Questi hanno un significato diverso nel caso l'apertura del diaframma della fotocamera sia 0 o maggiore: il secondo caso serve a simulare l'approssimazione di messa a fuoco di un obiettivo in base alla sua apertura di diaframma, il primo caso simula invece un obbiettivo puntiforme (*pinhole*) che quindi mette a fuoco tutta la scena da 0 a infinito. Nel caso in cui l'apertura è 0 i samples sono campioni uniformi all'interno del pixel; l'uso di più campioni permette le procedure di media (che fisicamente corrispondono ad una sfocatura) necessarie per l'*anti-aliasing*[1].

```
//aggiungo alla posizione del pixel (presa all'angolo in basso a sinistra) le
    quantità random comprese in [0,1]
raster_x+=rndX;
raster_y+=rndY;
```

Se il diaframma è maggiore di 0, i samples sono campioni dei raggi di luce che attraversano l'obiettivo della fotocamera e convergono in un pixel del sensore. Naturalmente l'obiettivo viene rappresentato da un'unica lente equivalente, modellata come un disco nel suo piano ottico centrale, di raggio `cam.aperturaDiaframma`: prendiamo un punto a caso su questo disco

```
raster_x+=cosf(2*M_PIf*rndX)*cam.aperturaDiaframma*rndY;
raster_y+=sinf(2*M_PIf*rndX)*cam.aperturaDiaframma*rndY;
```

A seconda del fuoco della camera i raggi non convergono se `cam.fuoco <=1`, altrimenti convergono tutti in un unico punto, che è il punto messo a fuoco. Naturalmente qui il rettangolo dei pixel modella il sensore della macchina fotografica e la lente è un disco parallelo a questo piano disposto tra esso e la scena da rendere. Però questo disco non viene mai esplicitamente modellato: le proprietà ottiche della lente vengono qui rappresentate da

```
origin=
    origin+cam.U*cam.fuoco*(x-raster_x)+cam.V*cam.fuoco*(y-raster_y);
```

Chiamiamo `pf` il punto dello spazio, posto inizialmente nel piano del sensore, che vogliamo sia messo a fuoco dalla camera. La modifica della sua distanza dal viewport modella la procedura ottica di messa a fuoco dell'immagine. Chiamiamo `cam.eye` il centro del disco che modella la lente, e `cam.U`, `cam.V` e `cam.W` un sistema ortonormale di tre versori applicati a `cam.eye`, con i primi due versori giacenti nel piano del disco ed il terzo,`cam.W`, perpendicolare al disco e diretto verso la scena, ovvero verso il semispazio opposto al viewport. Quindi, se `cam.d` misura la distanza fra la lente ed il piano del viewport, il punto del viewport (pixel) più vicino al centro `cam.eye` della lente è `-cam.W*cam.d` (il segno meno è dovuto al fatto che il versore `cam.W` è diretto dal lato opposto a quello del viewport). Riassumendo, la distanza fra la lente ed il punto messo a fuoco nella scena è la lunghezza del vettore `pf-cam.eye`, mentre la distanza fra la lente ed il viewport è `cam.d`, ossia la lunghezza del vettore `cam.d*cam.W`. Indichiamo allora con `cam.fuoco` il rapporto fra queste due distanze, ossia

$$\texttt{cam.fuoco} = \frac{||\texttt{pf-cam.eye}||}{||\texttt{cam.W*(-cam.d)}||}$$

[1] tecnica per ridurre l'effetto aliasing (in italiano, scalettatura, gradinatura o scalettamento) quando un segnale a bassa risoluzione viene mostrato ad alta risoluzione. L'antialiasing ammorbidisce le linee smussandone i bordi e migliorando l'immagine.

Poiché però è scomodo e costa tempo calcolare le norme, otteniamo lo stesso rapporto, grazie alla proporzionalità, come rapporto di componenti omologhe dei vettori al numeratore e denominatore. Delle tre componenti xyz utilizziamo la z, poiché questo ci assicura che `cam.W.z` è sicuramente diverso da 0 (è zero solo nel caso in cui la fotocamera sia posta orizzontalmente, ossia con asse ottico `cam.W` disposto verticalmente: questo significa fotografare la scena a 90 gradi dall'alto o dal basso, una scelta molto innaturale). Quindi si ottiene:

```
cam.fuoco=(pf.z-cam.eye.z)/(cam.W.z*(-cam.d));
```

Anche se l'immagine viene resa nel viewport, la procedura di rendering ubbidisce alla legge della prospettiva centrale: per poter scrivere le formule di prospettiva (che sono una proiezione centrale verso il centro di visuale, ossia l'osservatore) occorre fissare la posizione dell'osservatore, che indichiamo con il vettore `origin`. Se l'osservatore guarda verso un punto della lente nel sistema di riferimento intrinseco al disco della lente, egli sta guardando verso un punto del viewport (ossia un pixel) la cui posizione indichiamo con `(raster_x,raster_y,-cam.d)`: infatti la distanza con segno fra i piani paralleli dalla lente al viewport è `-cam.d` (qui x e y indicano le dimensioni di larghezza e altezza del rettangolo del viewport). Pertanto, scrivendo come nel codice C il numero di tipo float $\frac{1}{2}$ come 0.5f, il raggio di visuale è:

```
// Get camera direction
float3 ray_direction;
ray_direction
  =cam.U*(raster_x-0.5f*w)+cam.V*(raster_y-0.5f*h)+cam.W*(-cam.d);
ray_direction.norm();

// Create a camera ray
cameraRay= Ray(origin,ray_direction);
```

Il passaggio successivo è quello di stimare il flusso di luce che colpisce il pixel a seconda del metodo scelto. Ciò avviene tramite il metodo `radiance`, il quale si occupa di utilizzare l'algoritmo per il calcolo della radianza scelto nel «`main.h`». I parametri passati in argomento sono il raggio `first` di uscita della luce dal punto osservato (il cui versore ha direzione opposta rispetto al raggio della camera), l'oggetto `o` colpito, gli indici `x` e `y` del pixel, e il numero di raggi riflessi `nRay`.

```
          //intersezione del raggio con gli elementi della scena:
if(intersect(cameraRay, t, o)){
         //si calcola il punto di intersezione
         float3 iP=cameraRay.o+cameraRay.d*t;
         //viene creato il primo raggio per il calcolo della radianza
         //questo raggio parte dal punto ed è diretto verso l'osservatore
         Ray first(iP,cameraRay.d*(-1));
         //si calcola la radianza nel punto iP
         r=r+radiance(first,o,x,y,nRay);
   }
else{
   //nel caso non si trovi nessun oggetto viene visualizzato il colore di background
        r=r+background;
}
```

Nella funzione `radiance` viene fatta distinzione rispetto al materiale dell'oggetto colpito. Se il materiale è di tipo speculare, infatti, la scelta migliore è quella di utilizzare il classico ray tracing; per gli altri materiali che rispondono al modello di Cook e Torrance, invece, sono utilizzati gli algoritmi di illuminazione globale illustrati nei precedenti capitoli. Nel programma è stata inoltre definita un'ulteriore possibilità per i materiali non perfettamente speculari: in questo caso si utilizza un parametro del materiale di nome `refImperfection`. Questo è utilizzato per modificare la distribuzione coseno, che abbiamo usato precedentemente per distribuire i raggi lungo l'emisfero. Elevando l'Inverse Cumulative Distribution Function per questo parametro, che deve essere compreso in $[0, 0.1]$ per garantire risultati soddisfacenti, possiamo gestire in maniera intuitiva la probabiltà di campionare un punto arbitrariamente vicino al versore `W`. Abbiamo assegnato finora a questo versore la normale all'oggetto, ma, se utilizzassimo invece la direzione di riflessione perfetta del raggio, potremmo modellare il campionamento di raggi di un materiale non perfettamente speculare. Questo metodo genera, però, campioni anche al di sotto della superficie dell'oggetto; per evitare questo inconveniente si utilizza un ciclo `while`, che non termina finché non sono stati utilizzati un numero `refsample` di raggi, tali che il coseno dell'angolo che essi formano con la normale sia positivo.

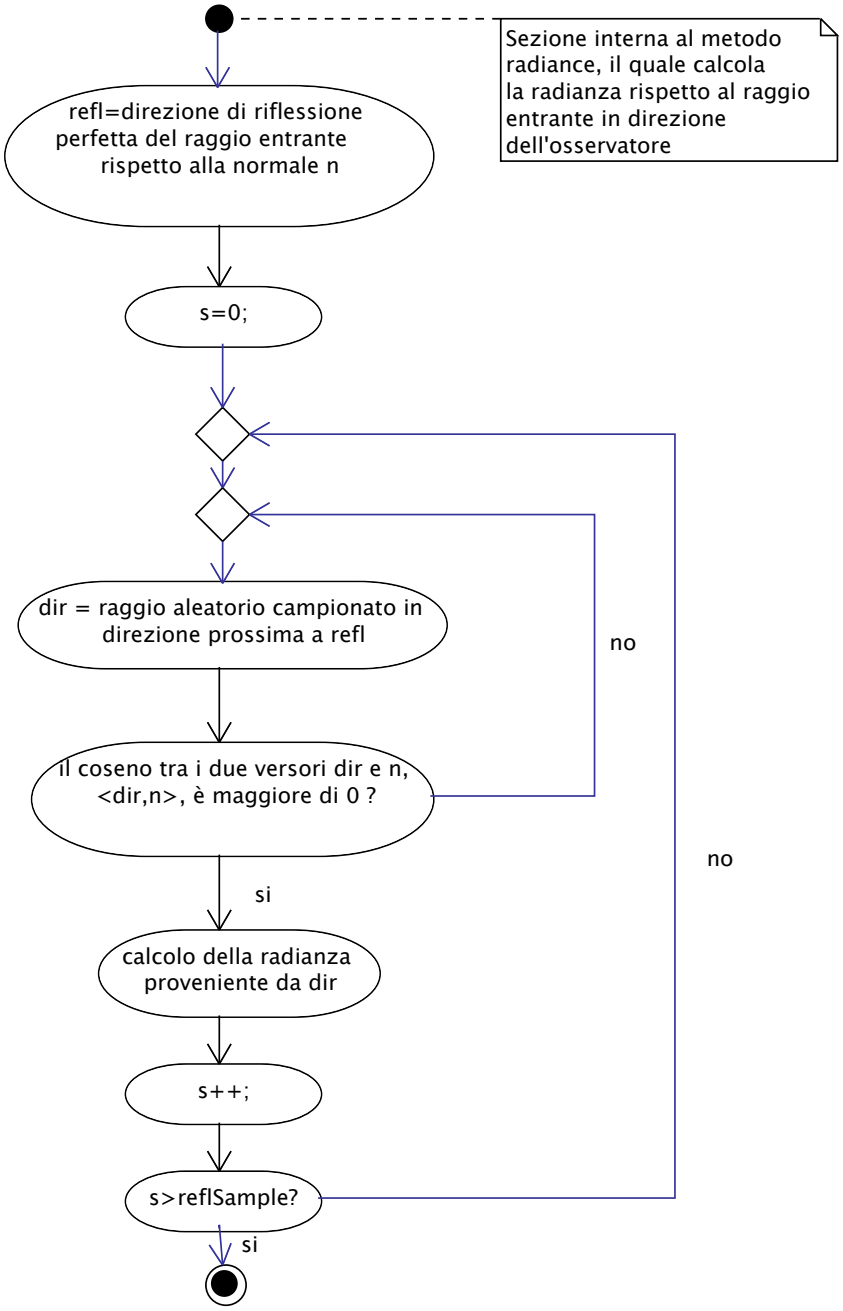


FIGURA 13.3.1. schema riassuntivo dell'algoritmo utilizzato per ottenere un materiale non perfettamente speculare.

Inseriamo di seguito il codice relativo al metodo `radance`:

```
<<srt.cpp>>
```

```
float3 radiance(Ray r, Obj** o,int x, int y,int& n){

    float3 radianceRefl;//radianza riflessa
    float3 radianceRefr;//radianza rifratta
    //si carica l'indice del materiale dell'oggetto o
    int mId=(*o)->matId;
    //si carica la normale dell'oggetto nel punto osservato
    float3 n1= (*o)->normal(r.o);
    //si calcola il coseno tra il raggio entrante e la normale
    float cos_i= r.d.dot(n1);
    //si calcola il fattore di Fresnel del materiale
    float3 Fresn = material[mId].getFresn(cos_i);

//si verifica se il materiale ha una componente speculare e che non sia stato
    superato il numero massimo di riflessioni
if((material[mId].Kr.max()>0)&&(n<MAX_DEPTH+1)){
    //si evitano le riflessioni interne al materiale
    if(cos_i>0){
                // riflessione del raggio in entrata rispetto alla normale n1:
                float3 refl= reflect(r.d,n1);
                //si verifica che il materiale sia perfettamente speculare o che non
                    sia la prima riflessione (si evita in questo modo l'aumento
                    esponenziale dei raggi nel caso in cui ci siano riflessioni
                    multiple tra specchi imperfetti)
                if((material[mId].refImperfection==0)||(n>0)){

                //raggio riflesso:
                Ray reflRay;
                reflRay.o=r.o;
                reflRay.d=refl;
                //distanza massima del raggio
                float t=inf;
                //puntatore all'oggetto che si andrà ad intersecare
                Obj** objX=new Obj*();
                *objX=NULL;
                //intersezione del raggio con gli elementi della scena
                if(intersect(reflRay, t, objX)){
                //si calcola il punto di intersezione
                float3 iP=reflRay.o+reflRay.d*t;
                //si crea il nuovo raggio
                Ray r2=Ray(iP,refl*(-1));
                //si aumentano il numero di riflessioni per il raytracing
                n++;

                //si calcola la radianza riflessa
```

```
//contiene una ricorsività della funzione radiance
radianceRefl=radiance(r2,objX,x,y,n)
.mult(material[mId].S_BRDF(Fresn));
//si diminuiscono il numero di riflessioni
n--;
}
//si cancella il doppio puntatore all'oggetto
delete objX;
//specchi imperfetti:
}else{
    //si aumenta il numero di riflessioni una volta per tutti i
        sample
    n++;
    //per ogni sample
    for(int s=0; s<refSample; s++){

        float random1;
        float random2;

       //flag per la verifica dell'orientazione del raggio
        bool okdir=true;
        //direzione del nuovo raggio
        float3 dir;
        //finchè non abbiamo un raggio con giusta orientazione il
            metodo continua a generare campioni
        while(okdir){
        //creazione delle variabili aleatorie uniformi
        if(aST==SOB){
            random1=generateRandom(aoSId[0],5,aST);
            random2=generateRandom(aoSId[0],6,aST);
        }
        else{
            random1= generateRandom(refSamples1[x+y*w],s+1,aST);
            random2= generateRandom(refSamples2[x+y*w],s+1,aST);
        }

        float rndPhi=2*M_PIf*(random1);
        //Inverse Cumulative Distribution Function del coseno
            modificata
        float rndTeta=acosf(
        powf(random2,material[mId].refImperfection));

        // creazione della base ortonormale rispetto alla direzione
            di riflessione
        float3 u,v,w;
        w=refl;
        float3 up(0.0015f,1.0f,0.021f);
        v=w%up;
        v.norm();
```

```
                    u=v%w;
                    //creazione della direzione
                    dir=u*(cos(rndPhi)*sinf(rndTeta))
                    +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));

                    //si verifica che la direzione formi un angolo di massimo 90
                        gradi con la normale
                    if(dir.dot(n1)>0)okdir=false;
                    //altrimenti si cerca una nuova direzione
                    }
                    //creazione del raggio riflesso
                    Ray reflRay;
                    reflRay.o=r.o;
                    reflRay.d=dir;
                    //massima distanza del raggio
                    float t=inf;
                    Obj** objX=new Obj*();
                    *objX=NULL;
                    //intersezione con gli oggetti della scena
                    if(intersect(reflRay, t, objX)){
                        //punto di intersezione del raggio con l'oggetto objX:
                        float3 iP=reflRay.o+reflRay.d*t;
                        //nuovo raggio:
                        Ray r2=Ray(iP,refl*(-1));

                        //calcolo della radianza riflessa
                        radianceRefl=radianceRefl+radiance(r2,objX,x,y,n)
                        .mult(material[mId].S_BRDF(Fresn));
                    }
                        delete objX;
                }
                n--;
                //divido per il numero di raggi che sono stati creati
                radianceRefl=radianceRefl*(1/(float)refSample);

                }
            }
        }
    //rifrazione:
    //si verifica che l'oggetto abbia valore Kg>0 non sia un metallo e che non si
        sia superato il numero massimo di riflessioni del ray tracer
    if((material[mId].Kg.max()>0)&&(n<MAX_DEPTH+1)&&(material[mId].k.max()==0)){

        //si verifica che l'indice di rifrazione sia uguale per tutte le componenti
            RGB
        //in questo caso il calcolo per la rifrazione sarà semplificato
        if((material[mId].ior.x==material[mId].ior.y)
        &&(material[mId].ior.x==material[mId].ior.z)){
            //direzione del raggio rifratto:
```

```
float3 dir;
//calcolo della direzione per il raggio rifratto:
if(refract(dir,r.d,n1,material[mId].ior.x)){
    //viene creato il raggio rifratto
    Ray refrRay(r.o,dir);

    //si verifica il parametro di imperfezione del materiale
    if((material[mId].refImperfection==0)||(n>0)){

        float t=inf;
        Obj** objX=new Obj*();
        *objX=NULL;
        if(intersect(refrRay, t, objX)){
            float3 iP=refrRay.o+refrRay.d*t;
            Ray r2=Ray(iP,dir*(-1));

            n++;
        //calcolo della radianza rifratta
        radianceRefr=radiance(r2,objX,x,y,n)
        .mult(material[mId].T_BRDF(Fresn));
        n--;
        }
    delete objX;
    }else{
        n++;
        //per ogni sample
        for(int s=0; s<refSample; s++){

            float random1;
            float random2;
            //flag di controllo sulla direzione creata
            bool okdir=true;
            float3 dir;
            while(okdir){
            //variabili aleatorie uniformi in [0,1]
            if(aST==SOB){
                random1=generateRandom(aoSId[0],5,aST);
                random2=generateRandom(aoSId[0],6,aST);
            }
            else{
                random1= generateRandom(refSamples1[x+y*w],s+1,aST);
                random2= generateRandom(refSamples1[x+y*w],s+1,aST);
            }
            //distribuisco i numeri random sull'emisfero
            float rndPhi=2*M_PIf*(random1);
            float rndTeta=acosf(
            powf(random2,material[mId].refImperfection));
            // creazione della base ortonormale creata a partire dal
                raggio rifratto:
```

```
                    float3 u,v,w;
                    w=refrRay.d;
                    float3 up(0.0015f,1.0f,0.021f);
                    v=w%up;
                    v.norm();
                    u=v%w;
                    //si calcola la direzione del raggio
                    dir=u*(cos(rndPhi)*sinf(rndTeta))
                    +v*(sin(rndPhi)*sinf(rndTeta))+w*(cosf(rndTeta));
                    //si verifica che l'angolo tra la normale e la nuova
                       direzione sia maggiore di 90 gradi
                    if(dir.dot(n1*(-1))>0)okdir=false;
                    }
                    //nuovo raggio:
                    Ray rRay;
                    rRay.o=r.o;
                    rRay.d=dir;
                    float t=inf;
                    Obj** objX=new Obj*();
                    *objX=NULL;
 if(intersect(refrRay, t, objX)){
                        //punto di intersezione
                        float3 iP=refrRay.o+refrRay.d*t;
                        Ray r2=Ray(iP,dir*(-1));
                    //calcolo della radianca rifratta
                    radianceRefr=radianceRefr
                    +radiance(r2,objX,x,y,n).mult(material[mId].T_BRDF(Fresn));
                    }
                }
                n--;
                //si mediano i contributi di tutti i raggi usati
                radianceRefr=radianceRefr*(1/(float)refSample);
            }
        }
    }
    //se l'indice di rifrazione è diverso nelle 3 componeneti RGB allora si
       devono calcolare 3 raggi uno per ogni componente
    //in questo caso però per facilitare il calcolo non viene considerato
       l'indice di imperfezione del materiale
    else{
        // Ray refraction based on normal
        // Raggio utilizzato per la rifrazione
        Ray* refrRay= new Ray[3];
        refrRay[0].o=r.o;
        refrRay[1].o=r.o;
        refrRay[2].o=r.o;
        float K[]={0,0,0};
        //calcolo dei 3 raggi rifratti
        refract(refrRay,r.d,n1,material[mId].ior);
```

```
        //carichiamo la brdf su un vettore di 3 elementi cosi da accedervi più
            facilmente
        float3 brdf= material[mId].T_BRDF(Fresn);
        float g[]={brdf.x,brdf.y,brdf.z};
        n++;
        //per ogni componente RGB
        for(int i=0;i<3;i++){
            //si verifica che non ci sia stata riflessione totale
            if(refrRay[i].depth!=0){
                float t=inf;
                Obj** objX=new Obj*();
                *objX=NULL;
                if(intersect(refrRay[i], t, objX)){
                    n++;
                    float3 iP=r.o+refrRay[i].d*t;
                    Ray r2=Ray(iP,refrRay[i].d*(-1));
                //viene calcolato il valore di radianza del raggio
                float3 pr= radiance(r2,objX,x,y,n);
                //si deve ora estrapolare la componente i di tale radianza
                float rg[3];
                rg[0]=pr.x;
                rg[1]=pr.y;
                rg[2]=pr.z;
                //si moltiplica infine per la componente della brdf
                    corrispondente
                K[i]+=rg[i]*g[i];

                }
                delete objX;
            }
        }
        radianceRefr=float3(K[0],K[1],K[2]);
        n--;
        delete[] refrRay;
    }
}
//algoritmi per il calcolo dell'illuminazione globale:

if((material[mId].Kd.max()>0)||(material[mId].slope>0)
||(material[mId].Le.max()>0)){

    //photon mapping:

    if((photonMap)&&(!Fg)){
        float3 radOut=
            photonRadiance(r,*o,KdTree,photond_2,nPhotonSearch,Pfilter)
        +radianceRefr+radianceRefl+Le(r,*o);
        if(causticPhoton>0){radOut=radOut
        +photonRadiance(r,*o,causticTree,causticd_2,nCausticSearch,Cfilter);}
```

```
            return radOut;
        }

        //photon mapping con calcolo dell'illuminazione diretta tramite raytracing
            stocastico:

        if(multiPassPhotonMap){
            return multiPassPhotonRadiance(r,x,y,*o,n)+radianceRefr+radianceRefl;
        }

        //raytracing stocastico (path tracing) :

        if(stoc){
            return radianceSTOC(r,(*o),x,y,n)+radianceRefr+radianceRefl;
       }

        //metodo di Jacobi stocastico:

        if((jacob)&&(!Fg)){
            float3 L=(*o)->P*(1/(*o)->areaObj)*M_1_PIf;
            return L+radianceRefr+radianceRefl;
       }

        //Final Gathering:

        if(Fg){
            float3 Out=FinalGathering(r,x,y,(*o))+radianceRefr+radianceRefl;
         //illuminazione caustiche:
        if((photonMap)&&(causticPhoton>0)){
       Out=Out
          +photonRadiance(r,(*o),causticTree,causticd_2,nCausticSearch,Cfilter);
            }
        return Out;
        }

        //se non è stato impostato nessuno di tali metodi allora viene restituito
            il colore nero
        return float3(0);
   }else{
       //se il materiale è riflettente o trasparente
       return radianceRefl+radianceRefr;
   }
}
```

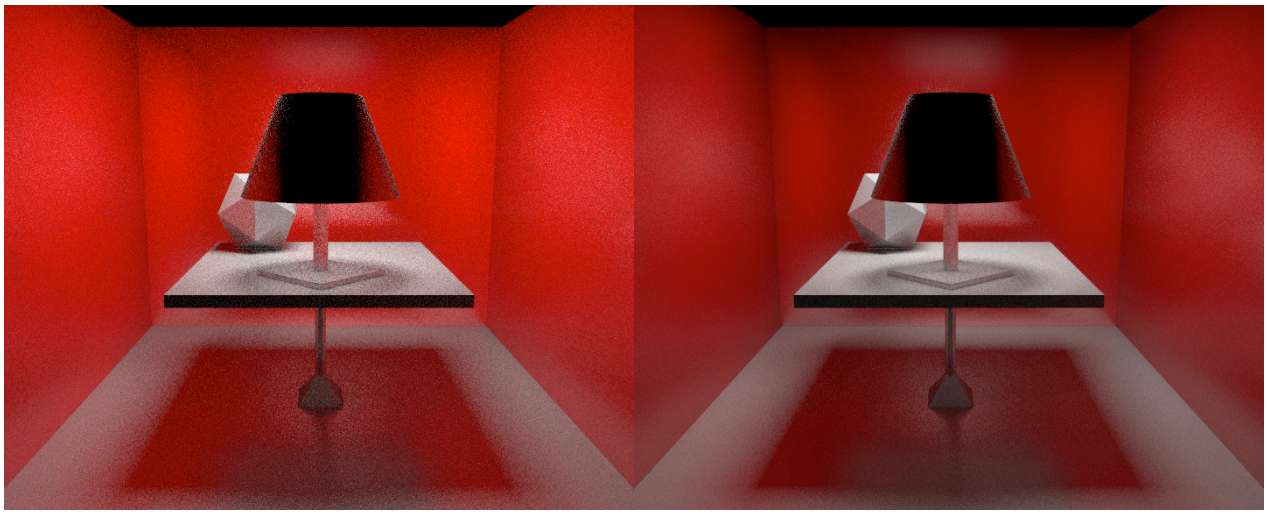


FIGURA 13.3.2. **riflessione non perfettamente speculare**: nell'immagine a destra sono stati utilizzati il doppio dei sample rispetto all'immagine di sinistra.

I valori ottenuti tramite la funzione `radiance` sono mediati per ogni `sample` scelto sul pixel e salvati in un array `image`, che contiene tutti i pixel dell'immagine (opportunamente tagliati con il metodo `clamp` in modo che i valori rimangano in $[0,1]^2$):

```
<<srt.cpp>>
<<funzione main>>
```

```
image[x+y*w].x=clamp(r.x);
image[x+y*w].y=clamp(r.y);
image[x+y*w].z=clamp(r.z);
```

Questi valori, calcolati per ogni pixel, vengono scritti in un file: scegliamo file di tipo `.ppm` per la facilità di utilizzo. Le seguenti righe di codice aprono il file e scrivono i dati.

```
 FILE *f = fopen("image.ppm", "w");
//intestazione ppm ci và il tipo (P3) la larghezza e la lunghezza dell'immagine e
    la profondità dei pixel
fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);

//per ogni pixel
for (int i=0; i<w*h; i++){

//per ogni pixel scriviamo il valore RGB
fprintf(f,"%d %d %d ", toInt(image[i].x), toInt(image[i].y), toInt(image[i].z)); }
```

Alla fine viene chiuso il file e liberata la memoria allocata:

```
//chiusura
printf("File written\n");
```

[2]nella computer graphics, per evitare di eliminare informazioni importanti, si cerca di creare sempre immagini in HDR così da decidere l'esposizione dell'immagine in post-produzione.

```
fclose (f);

//cancellazzione degli oggetti dalla memoria
 for(int i=0; i<nMesh+1;i++){
       delete m[i];
 }
delete[] m;
free(luci);
delete[] image;
delete[] samplesX;
delete[] samplesY;
delete[] aoSamplesX;
delete[] aoSamplesY;
```

CAPITOLO 14

Analisi dei risultati

In questo capitolo vogliamo confrontare e commentare i risultati ottenenti dai diversi algoritmi proposti nei capitoli precedenti. Sono valutati, inoltre, i diversi tempi di resa dell'immagine utilizzando il programma su un processore *Intel Core i7* a 2,3 GHz.

14.1. Gli artefatti del Final Gathering

Si analizza qui quanto il numero di patch, di cui una scena è composta, può incidere nel risultato finale del Final Gathering. Ricordiamo, infatti, che questo metodo utilizza una soluzione precalcolata di radiosità, la quale è costante su ogni oggetto della scena. Aumentando il numero di triangoli con cui una scena è definita l'approssimazione diventa migliore, ma, di conseguenza, aumenta anche il numero di sample necessari per il metodo di Jacobi stocastico incrementale. Nella figura (14.1.1) si può vedere cosa ciò implica, nel Final Gathering, utilizzando per il parametro `sceneDepth` il valore 0 (colonna di sinistra) e il valore 7 (colonna di destra). Nella prima colonna vi sono 2 patch per ogni muro della stanza; il colore del pavimento e della parete frontale è un grigio medio, in quanto la diffusione del colore avviene solo in una piccola zona, in confronto a tutto il muro, vicino alle pareti blu. Nel Final Gathering viene considerato questo colore e di conseguenza il risultato finale sarà meno blu. Nella figura (14.1.2) evidenziamo la differenza, effettuata pixel per pixel in ciascun canale RGB dell'immagine, tra le due soluzioni ottenute.

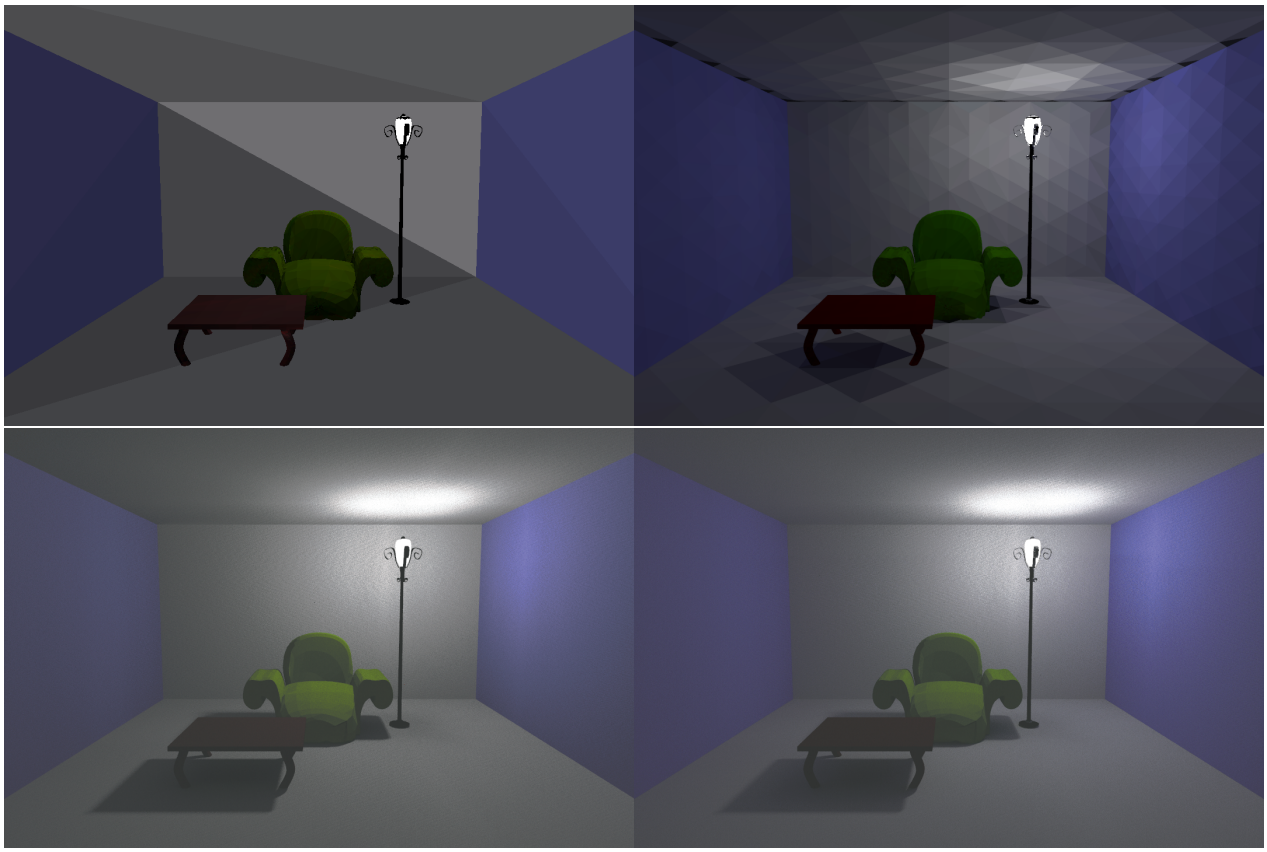


FIGURA 14.1.1. **parametro `sceneDepth` nel Final Gathering:** in questa immagine la prima riga è la soluzione tramite il metodo di Jacobi mentre la seconda il Final Gathering che l'utilizza. Mentre nella prima colonna è stato usato il valore 0 per il parametro `sceneDepth`, nella seconda colonna questo parametro è impostato a 7.

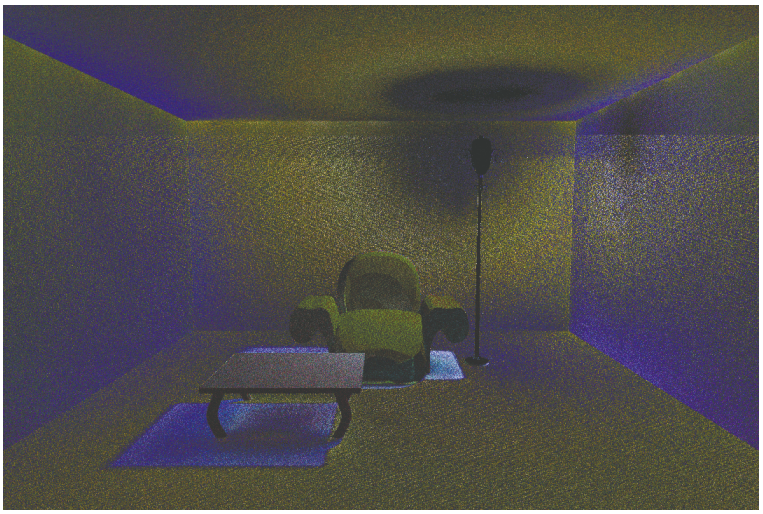


FIGURA 14.1.2. **Differenza pixel per pixel delle due soluzioni di Final Gathering**: Per visualizzare meglio questa differenza è stato aumentato il contrasto dell'immagine risultante.

14.2. Gestione del numero di fotoni nel Photon Mapping

Vogliamo effettuare un confronto sui tempi di rendering di una scena aumentando contemporaneamente il parametro per il numero di fotoni emessi dalle fonti di luce, `nPhoton`, e quello per il numero di fotoni utilizzati nella stima della radianza in ogni punto della scena, `nPhotonSearch`. È infatti molto difficile gestire questi due parametri senza una accurata conoscenza del metodo; vedremo inoltre che la velocità di realizzazione dell'immagine dipende anche da un terzo parametro che qui non è preso in considerazione: il raggio della sfera con cui si effettua la ricerca dei fotoni, che nel «`main.h`» viene chiamato `photond_2` per la mappa fotonica globale e `causticd_2` per la mappa delle caustiche.

nPhoton \ nPhotonSearch	80	240	480
200	0:2:30	0:3:37	N.V.
400	0:2:30	0:4:44	N.V.
800	0:2:30	0:5:57	0:7:10
1600	0:2:30	0:6:32	0:8:54
3200	0:2:30	0:6:34	0:10:51
6400	0:2:36	0:6:47	0:11:33
12800	0:3:60	0:7:18	0:11:53

TABELLA 14.2.1. Tabella dei tempi di rendering nel photon mapping. La sigla N.V. corrisponde a non valido, in quanto il numero di fotoni emessi è troppo basso rispetto a quelli ricercati. In questo caso infatti il calcolo della densità avverrà sul disco di massima estensione definito nel «`main.h`» come `photond_2`.

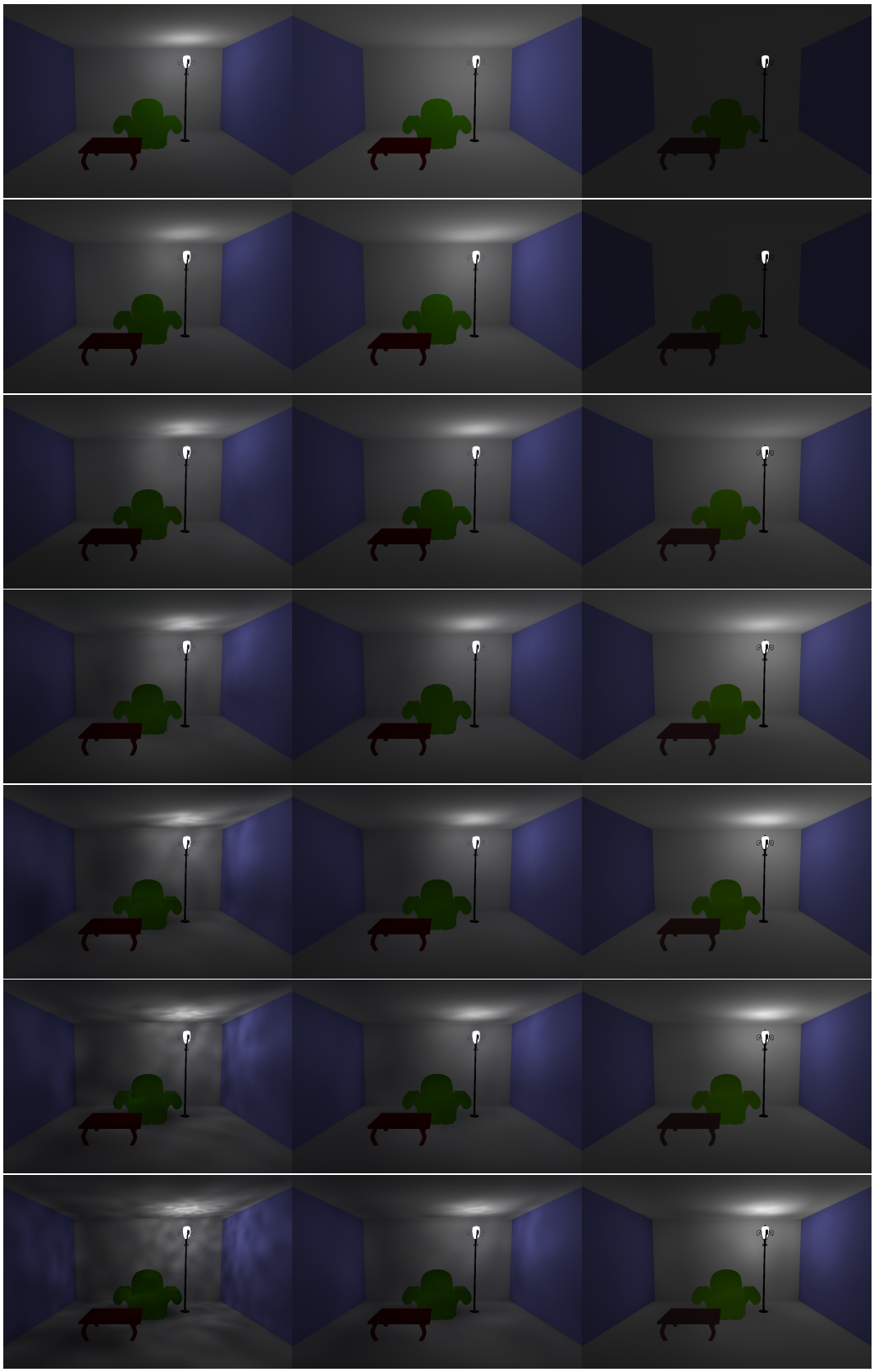


FIGURA 14.2.1. **confronto tra l'aumento dei fotoni emessi e quelli usati per la ricerca:** in ogni colonna le immagini sono state realizzate con un diverso numero di fotoni per la stima: 80, 240, 480. In ogni riga, invece, si utilizza un diverso numero di fotoni emessi: 200, 400, 800, 1600, 3200, 6400, 12800. Nelle prime due immagini della terza colonna il numero di fotoni emessi è però insufficiente.

Dalla tabella (14.2.1) è facile intuire che aumentando il numero, `nPhotonSearch`, dei fotoni utilizzati per la stima della radianza, aumenta di molto anche il tempo di visualizzazione dell'immagine; a confronto aumentare il numero, `nPhoton`, dei fotoni emessi comporta un aumento dei tempi di calcolo minore. Questo grazie all'utilizzo del Kd-Tree, che ci permette di suddividere i fotoni in Box e considerarne solo pochi alla volta. La velocità di rendering rispetto al numero di fotoni, usati nella stima, varia col raggio massimo di ricerca `photond_2`, la cui scelta dipende dal numero di fotoni all'interno della mappa e dalla scena che si vuole visualizzare. Il metodo raggiunge la massima velocità quando si utilizza il minimo raggio per cui si trovano `nPhotonSearch` fotoni per la stima della radianza in ogni punto della scena.

14.3. Caustiche

In questa sezione confrontiamo i due diversi metodi, presi in esame nel corso della trattazione, per la creazione delle caustiche. Nella figura 14.3.1, in alto, è stata utilizzata la formulazione emisferica dell'equazione del rendering, mentre nelle immagini in basso il Final Gathering con photon mapping. Nonostante le prime due immagini sembrino possedere la stesso livello di rumore, la prima è stata creata utilizzando 200 sample (100 per ogni triangolo della luce), mentre la seconda utilizzando 2 sample (1 per ogni triangolo della luce). Nella terza immagine invece, in cui sono stati utilizzati 200 sample come nella prima, il miglioramento è evidente. Questo dimostra quanto l'utilizzo di una formulazione ad area per l'illuminazione diretta possa diminuire il rumore in un'immagine. La mappa delle caustiche presenta però dei difetti: se la superficie della luce è ampia, infatti, il sottocampionamento è evidente. Ne consegue che la forma risulta errata in due delle tre caustiche mentre risulta esatta solo nella formulazione emisferica; la varianza di quest'ultima, al contrario di quanto avviene nella mappa delle caustiche, diminuisce all'aumentare della dimensione della luce. Notiamo inoltre che nella prima immagine si riesce a visualizzare l'effetto di una caustica multipla, generata quindi da un'altra caustica. Questi effetti non possono essere visualizzati nella mappa delle caustiche da noi creata, in quanto la proiezione per l'emissione dei fotoni avviene solamente a partire dalla luce.

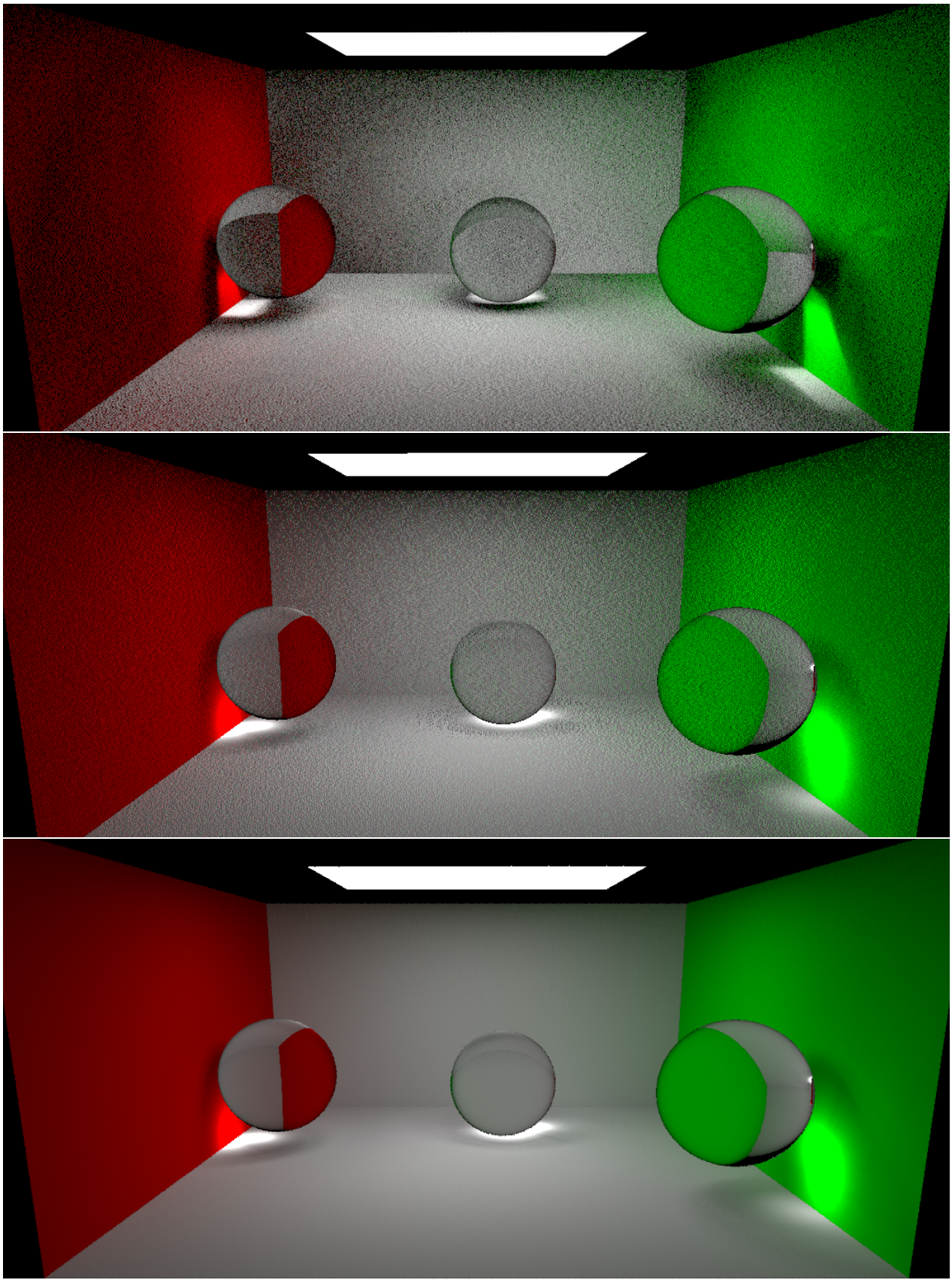
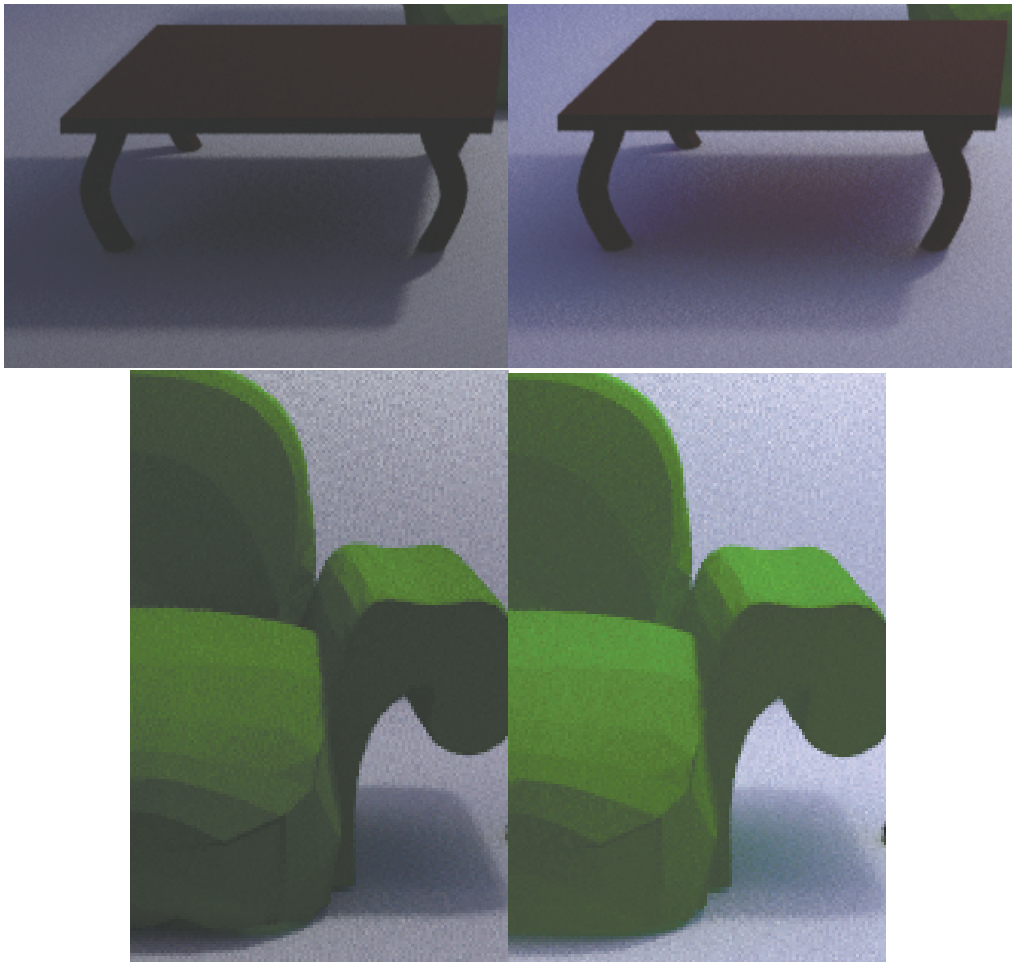


FIGURA 14.3.1. **Confronto tra la formulazione emisferica e la Caustic Map per la visualizzazione delle caustiche.**

14.4. Confronti tra i diversi algoritmi

In figura 14.4.1 si confrontano gli algoritmi di rendering presentati nei precedenti capitoli: `radianceSTOC` (prima immagine) e `Final Gathering`, quest'ultimo effettuato tramite soluzione di radiosità (seconda immagine) e mappa dei fotoni (terza immagine). Possiamo notare come il primo metodo generi un'immagine ad alto rumore rispetto a quella realizzata con il Final Gathering. Per un'immagine così semplice l'uso del photon mapping è eccessivo, in quanto non sono presenti ne geometrie ne BRDF particolari; in questo caso il rendering effettuato tramite Final Gathering con il metodo di Jacobi dà risultati migliori. Il photon mapping infatti è un metodo viziato (*biased*) e di conseguenza la terza immagine ha un valore atteso diverso dalle altre due. Sappiamo inoltre che l'errore di bias diminuisce all'aumentare dei fotoni, di conseguenza diventa grande nelle parti dell'immagine con pochi fotoni, come quelle in ombra che ricevono luce solo dalle riflessioni multiple. In queste parti infatti il numero di fotoni che colpiscono la superficie è minore a causa dell'utilizzo della Roulette Russa per la propagazione. Per questo motivo, come notiamo nella terza immagine, l'ombra della poltrona verde e del tavolo rosso hanno un'illuminazione innaturale.



Anche dai tempi di rendering delle immagini risulta che la scelta del photon mapping è inadeguata per questa scena. Elenchiamo qui di seguito i dati relativi alle tre immagini realizzate a partire da una scena con 43778 triangoli:

(1) Ray Tracing stocastico (`radianceStoc`):
- tempo di rendering: 3 ore 8 minuti e 24 secondi;

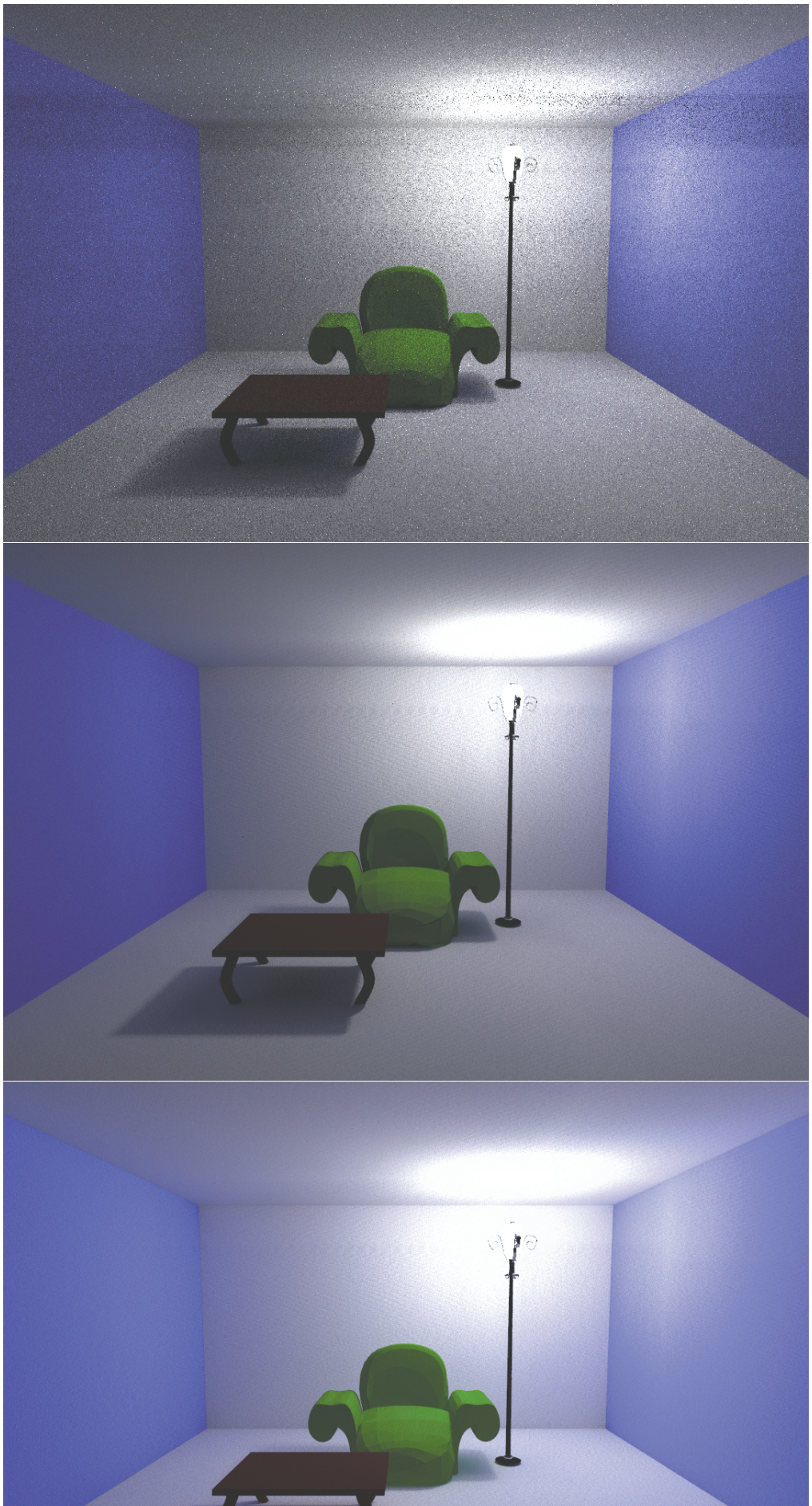
- 10 sample per pixel;
- 3 sample di illuminazione diretta (`dirsamps`);
- 10 sample di illuminazione indiretta (`aosamps`);

(2) Final Gathering (`FinalGathering`) con soluzione precalcolata di radiosità (`jacob`):

- tempo di rendering: 5 ore 56 minuti e 2 secondi;
- 10 sample per pixel;
- 15 sample di illuminazione diretta (`dirsamps`);
- 20 sample di illuminazione indiretta (`aosamps`);
- 15000000 sample per il calcolo della radiosità;
- 15 iterazioni del metodo di Jacobi;
- 7, `sceneDepth`;

(3) Final Gathering (`FinalGathering`) con l'utilizzo della mappa fotonica (`photonMap`):

- tempo di rendering: 4 giorni 1 ora 23 minuti e 7 secondi;
- 10 sample per pixel;
- 15 sample di illuminazione diretta (`dirsamps`);
- 20 sample di illuminazione indiretta (`aosamps`);
- 64000 fotoni emessi per la costruzione della photon map (`nPhoton`);
- 1000 fotoni usati per la ricerca (`nPhotonSearch`);



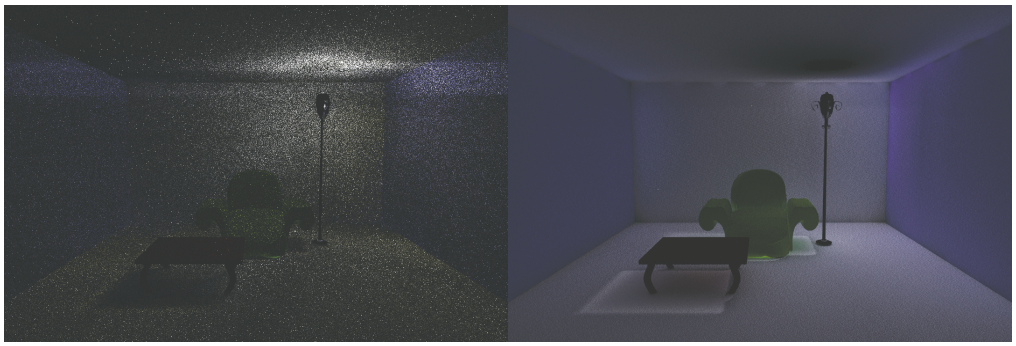


FIGURA 14.4.2. **Analisi delle differenze, effettuate pixel per pixel in ogni canale RGB, tra le immagini**: nella prima immagine si visualizza la differenza tra il Final Gathering effettuato con soluzione di radiosità e il Ray Tracing stocastico, nella seconda la differenza tra quest'ultimo e il Final Gathering ottenuto tramite mappa fotonica. Mentre nella prima differenza è presente solo del rumore, nella seconda è ben visibile l'errore dovuto al bias del Photon Mapping.

Nel programma sono due i metodi che utilizzano la Roulette Russa: il Photon Mapping e il Ray Tracing stocastico. In figura (14.4.4) essi sono messi a confronto: si può notare che quando il numero di riflessioni necessarie per ottenere un contributo all'illuminazione è elevato, il Ray Tracing stocastico non dà buoni risultati. Il Photon Mapping, invece, permette di ottenere risultati più apprezzabili nell'illuminazione indiretta (ma non in quella diretta) eliminando il rumore. I due metodi possono essere uniti utilizzando il Ray Tracing per l'illuminazione diretta e il Photon Mapping per quella indiretta. Nell'immagine è stato utilizzata come fonte di illuminazione un quadrato: sarebbe stato più corretto utilizzare una sfera che necessita, però, del Rejection Sampling per essere campionata. Questo aumenta il tempo di attesa che passa da 3 ore 50 minuti (con 100000 fotoni e 5 sample) a 12 ore e 9 minuti (con 8000 fotoni e 5 sample).

FIGURA 14.4.3. in questa immagine la fonte di luce è una sfera. Tempo di rendering 1 ore e 9 minuti e una mappa di 8000 fotoni emessi.

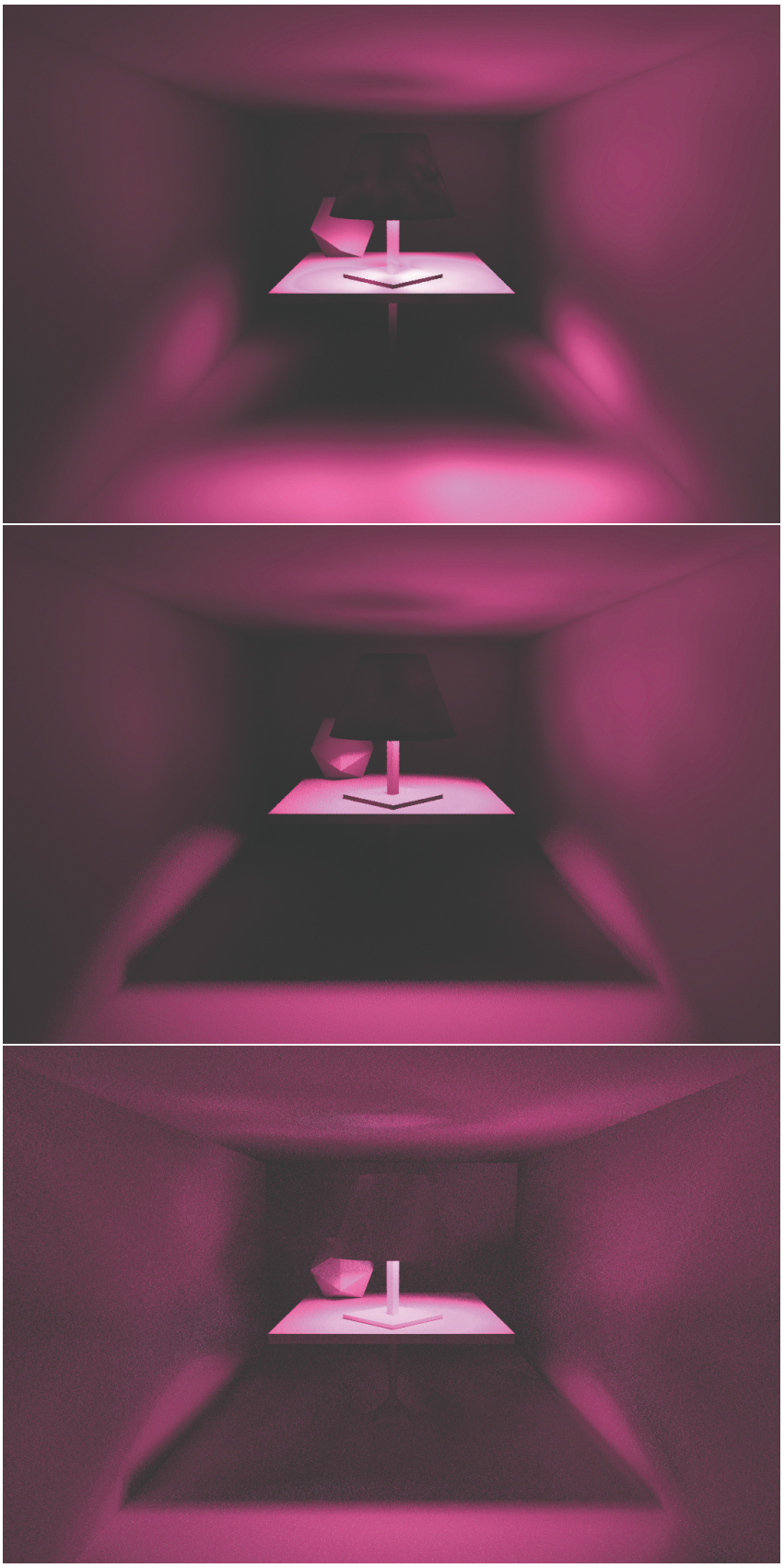


FIGURA 14.4.4. **metodo della Roulette Russa per le riflessioni:** Nella prima immagine si visualizza la soluzione ottenuta direttamente dal Photon Mapping, nella seconda si è usato invece il metodo `multiPassPhotonMap` e nella terza il Ray Tracing stocastico.

Parte 4

Appendici

CAPITOLO 15

Appendice: matrici di rotazione e quaternioni

15.1. Rotazioni in $\mathbb{R}^3$ e coniugazione di quaternioni

Poiché la moltiplicazione di quaternioni è non commutativa, l'operazione di coniugazione,

$$\Omega_{\mathbf{q}}\mathbf{p} = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}\,,$$

è non banale. Inoltre, grazie alla moltiplicatività della norma (**??**), se $\mathbf{q}$ è un quaternione unitario l'operatore lineare $\Omega_{\mathbf{q}}$ preserva la norma, e quindi è un operatore ortogonale sullo spazio dei quaternioni $\mathbb{H}$ pensato come spazio vettoriale reale a dimensione 4. Si noti che, se $\mathbf{q}$ è un quaternione unitario, $\mathbf{q}^{-1}$ coincide con il coniugato $\mathbf{q}$ in base a (**??**), e quindi $\Omega_{\mathbf{q}} = \mathbf{q}\mathbf{p}\mathbf{q}^*$.

TEOREMA 15.1.1. **(Coniugazione di quaternioni e rotazioni in $\mathbb{R}^3$.)** *Sia $\mathbf{p}$ un punto in uno spazio tridimensionale pensato come una classe di equivalenza per dilatazione in $\mathbb{R}^4$ grazie alle sue coordinate omogenee (p_x, p_y, p_z, p_w). Identifichiamo $\mathbf{p}$ con il quaternione*

$$\mathbf{p} = ((p_x, p_y, p_z), p_w) = (\mathbf{p}_v, p_w)\,.$$

Sia $\mathbf{q}$ un quaternione non nullo. Allora:

- (*i*) *L'operatore di coniugazione $\Omega_{\mathbf{q}}\mathbf{p} = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}$ trasforma $\mathbf{p} = (\mathbf{p}_v, p_w)$ in un quaternione $\mathbf{p}' = (\mathbf{p}'_v, p_w) = (p'_x, p'_y, p'_z, p_w)$, tale che $\|\mathbf{p}_v\| = \|\mathbf{p}'_v\|$, e quindi, ristretto al sottospazio tridimensionale dei quaternioni immaginari puri $\mathbf{p}_v, 0)$, è un operatore unitario (ossia di rotazione).*
- (*ii*) *Qualsiasi multiplo reale, diverso da zero, di $\mathbf{q}$ effettua la stessa trasformazione, e quindi la rotazione di punti tridimensionali, vista come coniugazione rispetto ad un opportuno quaternione, è compatibile con la rappresentazione di quei punti come classe di equivalenza di vettori in $\mathbb{R}^4$, ossia non dipende dalla scelta dei rappresentanti della classe di equivalenza.*
- (*iii*) *Se il quaternione $\mathbf{q}$ è unitario e come in (**??**) lo scriviamo $\mathbf{q} = (\mathbf{u}_{\mathbf{q}} \sin\omega, \cos\omega)$, allora $\Omega_{\mathbf{q}}$, ristretto al sottospazio tridimensionale dei quaternioni immaginari puri , è l'operatore di rotazione antioraria di angolo 2ω attorno all'asse $\mathbf{u}_{\mathbf{q}}$. Qui la rotazione si intende antioraria quando vista da un osservatore orientato come il vettore $\mathbf{u}_{\mathbf{q}}$ che guarda verso l'origine.*

*Dimostrazione*Osserviamo anzitutto che la parte (*ii*) è banale, in quanto l'inversa di $s\mathbf{q}$ è $\mathbf{q}^{-1}s^{-1}$, e la moltiplicazione con scalare gode della proprietà commutativa. Perciò $(s\mathbf{q})\mathbf{p}(s\mathbf{q})^{-1} = s\mathbf{q}\mathbf{p}\mathbf{q}^{-1}s^{-1} = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}ss^{-1} = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}$. Possiamo quindi assumere $\mathbf{q}$ come un quaternione unitario senza perdita di generalità. Per un quaternione unitario $\mathbf{q}$, $\mathbf{q}^{-1} = \mathbf{q}^*$; possiamo quindi scrivere $\mathbf{q}\mathbf{p}\mathbf{q}^{-1}$ come $\mathbf{q}\mathbf{p}\mathbf{q}^*$.

Dimostriamo la parte (*i*). La parte reale di un qualsiasi quaternione, $\operatorname{Re}\mathbf{q}$, può essere estratta usando la formula $2\operatorname{Re}\mathbf{q} = \mathbf{q} + \mathbf{q}^*$. Consideriamo $2\operatorname{Re}(\mathbf{q}\mathbf{p}\mathbf{q}^*) = \mathbf{q}\mathbf{p}\mathbf{q}^* + (\mathbf{q}\mathbf{p}\mathbf{q}^*)^* = \mathbf{q}\mathbf{p}\mathbf{q}^* + \mathbf{q}\mathbf{p}^*\mathbf{q}^*$. Visto che la moltiplicazione tra quaternioni è bilineare, possiamo scrivere l'ultimo membro come $\mathbf{q}(\mathbf{p} + \mathbf{q}^*)\mathbf{q}^* = 2\mathbf{q}\operatorname{Re}\mathbf{p}\mathbf{q}^* = 2\operatorname{Re}\mathbf{p}$. Quindi $\mathbf{q}$ coniuga $\mathbf{p} = (\mathbf{p}_v, p_w)$ in $\Omega_{\mathbf{q}}\mathbf{p} = \mathbf{p}' = (\mathbf{p}'_v, p_w)$, preservando la parte reale di $\mathbf{p}$. Inoltre l'operazione di moltiplicazione mantiene la norma perché la norma è moltiplicativa (si veda (**??**)) e $\mathbf{q}$ è unitario: quindi $\|\mathbf{p}\| = \|\mathbf{p}'\|$. Infine, poiché p_w resta inalterata, $\|\mathbf{p}_v\| = \|\mathbf{p}'_v\|$.

Per ultimo dimostriamo (*iii*). Abbiamo visto che si può scegliere $\mathbf{q} = (\mathbf{q}_v, q_w)$ quaternione *unitario*. Sia $\mathbf{p}$ un quaternione *immaginario puro*. Dall'ultima identità della regola di moltiplicazione

(**??**) si verifica facilmente che

$$\Omega_{\mathbf{q}}\mathbf{p} = ((q_w^2 - \mathbf{q}_v \cdot \mathbf{q}_v)\mathbf{p}_v + 2(\mathbf{q}_v \cdot \mathbf{p}_v)\mathbf{q}_v + 2q_w(\mathbf{q}_v \times \mathbf{p}_v),\, 0)\,,$$

e poiché $\mathbf{q} = \mathbf{u}_{\mathbf{q}} \sin\omega,\, \cos\omega)$ abbiamo

$$\begin{aligned}\Omega_{\mathbf{q}}\mathbf{p} &= ((\cos^2\theta - \sin^2\theta)\mathbf{p}_v + 2\sin^2\theta(\mathbf{p}_v \cdot \mathbf{u}_{\mathbf{q}})\mathbf{u}_{\mathbf{q}} + 2\cos\theta\sin\theta\,\mathbf{u}_{\mathbf{q}} \times \mathbf{p}_v), 0\\ &= (\cos 2\theta\,\mathbf{p}_v + (1 - \cos 2\theta)(\mathbf{p}_v \cdot \mathbf{u}_{\mathbf{q}})\mathbf{u}_{\mathbf{q}} + \sin 2\theta\,\mathbf{u}_{\mathbf{q}} \times \mathbf{p}_v,\, 0)\,.\end{aligned}$$

□

Esempio 15.1.2. Calcoliamo il quaternione unitario associato alla rotazione determinata di un angolo ω in senso antiorario rispetto all'asse x, o y, o z.

Dalla parte (*iii*) del Teorema 15.1.1 si trova che, per ciascun versore canonico di base $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, il quaternione è $\mathbf{q}_i = (\sin\omega\,\mathbf{e}_i,\, \cos\omega)$. □

15.1.1. Composizione di rotazioni e prodotto di quaternioni. Vediamo come la composizione di rotazioni si associa al prodotto di quaternioni. Consideriamo due quaternioni unitari, $\mathbf{q}_1$ e $\mathbf{q}_2$, ed un vettore $\mathbf{p} \in \mathbb{R}^3$ trasformato in un quaternione mediante le sue coordinate omogenee, $\mathbf{p} = (p_x, p_y, p_z, p_w)$. In base al Teorema 15.1.1, l'operazione

$$\mathbf{q}_2(\mathbf{q}_1\mathbf{p}\mathbf{q}_1^*)\mathbf{q}_2^*$$

ristretta al sottospazio tridimensionale dei quaternioni puramente immmaginari coincide la composizione delle rotazioni associate a $\mathbf{q}_1$ ed a $\mathbf{q}_2$, in questo ordine. Ponendo $\mathbf{q} = \mathbf{q}_2\mathbf{q}_1$ la formula precedente può essere scritta come

$$\mathbf{q}\mathbf{p}\mathbf{q}^*\,.$$

Riassumendo,

Corollario 15.1.3. *La mappa Ω è un omomorfismo dal gruppo moltiplicativo $\mathbb{H}$ al gruppo delle matrici ortogonali reali su $\mathbb{R}^4$:*

15.1.2. Matrice di rotazione in termini di quaternioni. Ora troviamo la forma matriciale dell'operatore $\Omega_{\mathbf{q}}$ su $\mathbb{R}^4 \sim \mathbb{H}$ dato dalla coniugazione con il quaternione $\mathbf{q}$. Ritorniamo, per maggiore generalità, al caso di un quaternione $\mathbf{q}$ non necessariamente unitario.

Poiché la moltiplicazione fra quaternioni è bilineare, possiamo esprimere questa operazione sui quaternioni (pensati come vettori in $\mathbb{R}^4$) tramite matrici a dimensione 4, spezzandola nella moltiplicazione sulla sinistra, $\mathbf{q}\mathbf{p}$, e la moltiplicazione sulla destra, $\mathbf{p}\mathbf{q}^*$.

Scriviamo $\mathbf{L}^q\mathbf{p}$ la moltiplicazione sulla sinistra, $\mathbf{p} \mapsto \mathbf{q}\mathbf{p}$, con $\mathbf{q} = (q_x, q_y, q_z, q_w) = (\mathbf{q}_v, q_w)$. Segue immediatamente dalla regola di moltiplicazione (**??**) che la matrice associata all'operatore lineare $\mathbf{L}^q$ è

$$\mathbf{L}^q = \begin{pmatrix} q_w & -q_z & q_y & q_x \\ q_z & q_w & -q_x & q_y \\ -q_y & q_x & q_w & q_z \\ -q_x & -q_y & -q_x & q_w \end{pmatrix}$$

Scriviamo ora la moltiplicazione sulla destra, $\mathbf{p} \mapsto \mathbf{p}\mathbf{q}^*$ come $\mathbf{R}^{q^*}\mathbf{p}$, dove l'operatore $\mathbf{R}^{q^*}$ viene espresso, grazie alla regola di moltiplicazione, come

$$\mathbf{R}^{q^*} = \begin{pmatrix} q_w & -q_z & q_y & -q_x \\ q_z & q_w & -q_x & -q_y \\ -q_y & q_x & q_w & -q_z \\ q_x & q_y & q_x & q_w \end{pmatrix}$$

Ora un calcolo elementare ma tedioso mostra che la matrice $\mathbf{M}^{\mathbf{q}}$ associata all'operatore di coniugazione $\Omega_{\mathbf{q}}$ è

$$\begin{aligned}\mathbf{M}^{\mathbf{q}} &= \mathbf{L}^{q}\mathbf{R}^{q^*} \\ &= \begin{pmatrix} q_w^2+q_x^2-q_y^2-q_z^2 & 2(q_xq_y-q_wq_z) & 2(q_xq_z+q_wq_y) & 0 \\ 2(q_xq_y+q_wq_z) & q_w^2-q_x^2+q_y^2-q_z^2 & 2(q_yq_z-q_wq_x) & 0 \\ 2(q_xq_z-q_wq_y) & 2(q_yq_z+q_wq_x) & q_w^2-q_x^2-q_y^2+q_z^2 & 0 \\ 0 & 0 & 0 & q_w^2+q_x^2+q_y^2+q_z^2 \end{pmatrix}.\end{aligned}$$

Semplificando otteniamo

$$\mathbf{M}^{\mathbf{q}} = \begin{pmatrix} \|\mathbf{q}\|^2-2(q_y^2+q_z^2) & 2(q_xq_y-q_wq_z) & 2(q_xq_z+q_wq_y) & 0 \\ 2(q_xq_y+q_wq_z) & \|\mathbf{q}\|^2-2(q_x^2+q_z^2) & 2(q_yq_z-q_wq_x) & 0 \\ 2(q_xq_z-q_wq_y) & 2(q_yq_z+q_wq_x) & \|\mathbf{q}\|^2-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & \|\mathbf{q}\|^2 \end{pmatrix}$$

Nel caso il quaternione $\mathbf{q}$ sia unitario si ha una ulteriore semplificazione:

$$\mathbf{M}^{\mathbf{q}} = \begin{pmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y-q_wq_z) & 2(q_xq_z+q_wq_y) & 0 \\ 2(q_xq_y+q_wq_z) & 1-s(q_x^2+q_z^2) & 2(q_yq_z-q_wq_x) & 0 \\ 2(q_xq_z-q_wq_y) & 2(q_yq_z+q_wq_x) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{15.1.1}$$

Questo dimostra la prima parte del seguente risultato:

COROLLARIO 15.1.4. **(Matrici di rotazione espresse in termini di quaternioni.)** *Consideriamo la matrice di rotazione (ossia ortogonale reale con determinante $+1$) $\mathbf{M}^q$ dell'operatore di coniugazione $\Omega_{\mathbf{q}}$ dove $\mathbf{q}$ è un quaternione unitario: essa ha la forma espressa in (15.1.1). Viceversa, ogni matrice di rotazione su $\mathbb{R}^3$, espressa nel modo seguente in forma di matrice affine a dimensione quattro,*

$$\mathbf{M}^{\mathbf{q}} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

è associata al quaternione unitario $\mathbf{q} = (q_x,\, q_y,\, q_z\, q_w)$ dato da

$$\begin{aligned} q_w &= \pm\frac{1}{2}\sqrt{m_{00}+m_{11}+m_{22}+1} = \pm\frac{1}{2}\sqrt{\mathrm{tr}\,(\mathbf{M}^{\mathbf{q}})} \\ q_x &= \frac{m_{21}-m_{12}}{4q_w} \\ q_y &= \frac{m_{02}-m_{20}}{4q_w} \\ q_z &= \frac{m_{10}-m_{01}}{4q_w}. \end{aligned} \tag{15.1.2}$$

*Dimostrazione*Dobbiamo solo dimostrare la seconda parte dell'enunciato, ossia la ricostruzione del quaternione a partire dalla matrice. Questo significa invertire la prima parte dell'enunciato, ossia ricavare il quaternione dall'espressione (15.1.1). Da questa espressione si vede subito che la traccia (ossia la somma dei coefficienti diagonali) della matrice è $4(1-q_x^2-q_y^2-q_z^2)$. Poiché $\mathbf{q}$ è un quaternione unitario, $q_x^2+q_y^2+q_z^2+q_w^2=1$ e quindi

$$\mathrm{tr}\,(\mathbf{M}^{\mathbf{q}}) = 4q_w^2\,.$$

Questo prova la prima identità in (15.1.2). Esaminando ancora (15.1.1) si vede che $m_{21} = 2(q_w q_x + q_y q_z)$ e $m_{12} = 2(q_y q_z - q_w q_x)$, da cui segue la seconda identità in (15.1.2. Le restanti due identità si provano allo stesso modo. □

Esprimere le matrici di rotazione in termini di quaternioni è una notevole opportunità di ridurre la mole di calcoli in Computer Graphics, dove le rotazioni intervengono al cambiare della posizione dell'osservatore (come accade continuamente durante le animazioni). Esewguire il calcolo tramite moltiplicazione di quaternioni è numericamante vantaggioso, ma soprattutto questa procedura è facile da implementare in hardware, dal momento che la moltiplicazione di due quaternioni è lineare nelle coordinate di entrambi: l'implementazione in hardware permette di eseguirla in maniera velocissima. Pertanto può essere utile svolgere i seguenti esercizi.

Esercizio 15.1.5. Rivedere il precedente esempio 15.1.2 e ritrovarne il risultato mediante il Corollario 15.1.4.

Esercizio 15.1.6. Calcolare il quaternione unitario associato alla rotazione determinata dagli angoli di Eulero θ e ϕ introdotti nella Definizione **??**.

Esercizio 15.1.7. Calcolare il quaternione associato alla componente di rotazione della matrice affine di rototraslazione dello spostamento della macchina da ripresa, calcolata nel Corollario **??**.

CAPITOLO 16

Appendice: matematica della prospettiva

16.1. Introduzione alle trasformazioni prospettiche

Questa Appendice, tratta dal Capitolo 16 del libro di testo di Geometria **[39]** (al quale rinviamo il lettore per una analisi approfondita della natura geometrica della trasformazione della prospettiva centrale), presenta vari tipi di trasformazioni prospettiche (dette anche *assonometriche*. Queste trasformazioni si suddividono in due categorie: *proiezione centrale*, se ci sono fasci di rette parallele che dopo la proiezione convergono verso opportuni punti di fuga, oppure *proiezioni parallele*, se questo non succede. Nel caso delle proiezioni parallele, i fasci di rette parallele rimangono paralleli, e la priezione è determinata dalla scelta di un piano di proiezione e dalla direzione di proiezione. Se tale direzione è perpendicolare al piano di proiezione si dice che la proiezione è *ortogonale* (o *ortografica*), altrimenti *obliqua*. In generale il piano di proiezione non contiene l'origine, e quindi le proiezioni prospettiche non fissano l'origine e pertanto non sono operazioni lineari. Vedremo che la proiezione centrale si rappresenta con matrici proiettive, le altre con matrici affini. La proiezione centrale è quella più videorealistica, utilizzata in Computer Graphics, e la studiamo per prima (un caso particolarmente semplice nella Sezione 16.2, il caso generale nelle Sezioni 16.5 e 16.6). La proiezione ortogonale è trattata alla Sezione 16.3, in modo da poter fornire una versione unificata della forma matriciale delle proiezioni centrale ed ortogonale nella Sezione 16.4. Sarebbe possibile estendere questa trattazione unificata anche alle altre proiezioni parallele, ma lasciamo questo tedioso compito al lettore.

Nel calcolare le matrici delle trasformazioni prospettiche dobbiamo mettere in guardia il lettore che la Computer Graphics ha una tradizione assai peculiare, quella di far agire le matrici sui vettori non da sinistra ma da destra. La ragione storica di ciò è che i primordi della Computer Graphics furono sviluppati non da matematici, bensì da studiosi a cui pareva strano che se due operatori A e B agiscono su un vettore $\mathbf{p}$ *in questo ordine* allora si debba avere che la composizione dei due uno dopo l'altro si debba scrivere $BA\mathbf{p} = B(A\mathbf{p})$ invece che $AB\mathbf{p}$. Per rovesciare l'ordine con cui i due simboli si succedono sulla carta, questi studiosi preferirono scrivere l'azione da destra, in modo che la composizione diventasse $\mathbf{p}AB = (\mathbf{p}A)B$. Per nostra fortuna a questo punto il lettore avrà studiato la matematica e compreso la ragione dell'ordine naturale, e gli sarà facile capire gli articoli di Computer Graphics, nei quali purtroppo l'ordine è opposto.

16.2. Prospettiva centrale, proiezione standard

Prendendo in esame l'Esempio 13.5.4 di **[39]**, osserviamo anzitutto, a scanso di malintesi, che anche quando i rappresentanti delle classi in $\mathbb{P}^n$ si scelgono del tipo $(x_1\,,\,x_2\,,\,\ldots\,,\,x_{n-1}\,,\,1)$, la forma della matrice in $PGL_n(\mathbb{R})$ associata ad una trasformazione prospettica non ha necessariamente l'ultima riga $(0\,,\,\ldots\,,\,0\,,\,1)$, come invece avviene per le matrici delle trasformazioni affini **[39**, Proposizione 14.1.4]. Infatti questa sarebbe la forma giusta per l'azione sui punti al finito se la trasformazione mandasse l'iperpiano $\{x_n = 1\}$ in sé, come appunto avviene per le trasformazioni affini di $\mathbb{R}^{n-1}$ quando considerate come trasformazioni lineari in $\mathbb{R}^n$ che lasciano invariante tale iperpiano: ma ciò non avviene per le trasformazioni proiettive, per le quali l'azione sui vettori non è neppure definita, visto che i rappresentanti delle classi dell'equivalenza proiettiva possono essere dilatati (e quindi

uscire dal suddetto piano) senza che l'azione ne risenta. Si può vedere un esempio concreto alla fine di [**39**, Esempio 13.5.4]ed esattamente in [**39**, eq. (13.5.2)], che riprendiamo in esame nella prossima Sezione.

Come è consuetudine in Computer Graphics, chiamiamo x e y le coordinate orizzontale e verticale, e z la profondità, orientata in modo da aumentare dall'origine verso l'osservatore. Il piano di visuale è parallelo agli assi x e y e quindi perpendicolare all'asse z, diciamo in posizione $z = d$ (qui stiamo supponendo che l'osservatore non si trovi verticalmente sopra l'origine, cioè sul piano x-y: in genere, in Computer Graphics, l'osservatore è poco al di sopra dell'asse z, nel senso che la sua posizione ha z positivo grande, y non molto grande e positivo, e $x = 0$). Si osservi che non stiamo richiedendo che il punto $\mathbf{p}$ abbia valore positivo di z: anzi, se invece che modellare un osservatore che guarda una scena stessimo modellando una camera oscura come un cubo con un piccolo foro nell'origine disposto nel semispazio $z \leqslant 0$, il piano di visuale, cioè in questo caso il piano della pellicola, passerebbe attraverso l'interno del cubo, e quindi avrebbe d negativo: in tal caso l'immagine creata dalla proiezione sarebbe ribaltata rispetto alla scena reale, perché i raggi si incrociano nel passare tutti attraverso l'origine.
Con questa scelta di coordinate, la proiezione sul piano di visuale porta gli assi x e y della scena tridimensionale su rispettivi assi orizzontale e verticale in tale piano: quindi i nomi delle coordinate sono quelli naturali per la compatibilità, perché se immaginiamo che questo sia il piano del monitor, è naturale chiamare questi assi del monitor x e y, rispettivamente.

Fissiamo ora in due modi diversi i parametri della proiezione prospettica della prospettiva centrale. Il primo modo è quello più naturale se si pensa di aver fissato una volta per tutte la posizione d del piano di visuale. In tal caso, collochiamo per ora l'osservatore nell'origine: cioè, l'origine è il *centro di proiezione*. Tratteremo nella prossima Sezione **??** il caso generale in cui il centro di proiezione è generico (non necessariamente l'origine).
La proiezione manda il generico punto $\mathbf{q} = (x, y, z)$ sul punto determinato sul piano di visuale dall'intersezione con la retta che passa per l'origine e per il punto $\mathbf{q}$. Riconosciamo in questo modo di procedere l'analogia con il principio della proiezione stereografica nella geometria proiettiva (Sezione **??**).
La proiezione avviene quindi tramite una similitudine, cioè una proporzione: il punto $\mathbf{q}$ viene mandato in

$$\mathbf{p} = (x_p, y_p, d) = \left(\frac{x}{z/d}, \frac{y}{z/d}, \frac{z}{z/d}\right) \equiv \left(\frac{x}{z/d}, \frac{y}{z/d}, d\right).$$

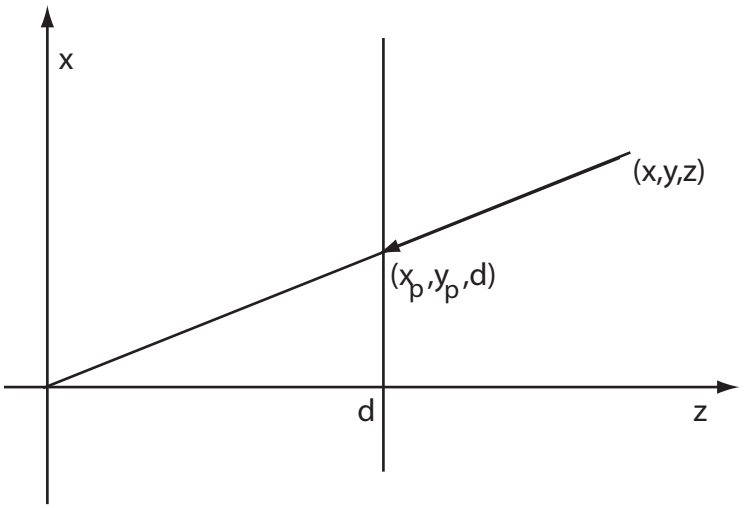


FIGURA 16.2.1. Proiezione della prospettiva centrale: piano di visuale in $z = d$

Possiamo riformulare questo risultato in termini di trasformazioni prospettiche: la trasformazione prospettica si scrive i ni di una (classe di equivalenza di) matrice $M_d^{(c)}$in $PM_4(\mathbb{R})$ come in [**39**, Esempio 13.5.4, eq. (13.5.1)]. Il punto **p** corrisponde ad un punto proiettivo al finito $[x, y, z, 1]$, e si ha

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = M_d^{(c)} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} . \tag{16.2.1}$$

Quindi $[X, Y, Z, W] = [x, y, z, \frac{z}{d}]$, e la quarta coordinata omogenea vale $W = z/d$. Perciò come rappresentante della classe immagine

$$[X, Y, Z, W]$$

possiamo scegliere il consueto rappresentante stereografico

$$\Big(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W}, 1\Big),$$

che corrisponde in $\mathbb{R}^3$ al punto $(x_p, y_p, z_p) = \Big(\frac{x}{z/d}, \frac{y}{z/d}, d\Big)$.

Ora veniamo al secondo modo utile di fissare i parametri prospettici. Poniamo il centro di prospettiva non più nell'origine, bensì in $(0, 0, -d)$, ed il piano di visuale in $z = 0$ (è consuetudine in Computer Graphics, per semplificare il processo di trasformazione da coordinate tridimensionali nel piano di visuale in $\mathbb{R}^3$ a coordinate bidimensionali del monitor, collocare il piano di visuale in $\{z = 0\}$). In tal modo, quando facciamo crescere la distanza d), il piano di visuale non si sposta, e possiamo più agevolmente confrontare i risultati della trasformazione prospettica).
Lo stesso argomento di proporzionalità adottato prima ora porta alle seguenti equazioni:

$$\begin{aligned} \frac{x_p}{d} &= \frac{x}{z+d} \\ \frac{y_p}{d} &= \frac{y}{z+d} \\ z_p &= 0 \end{aligned} \tag{16.2.2}$$

da cui si ricava

$$\begin{aligned} x_p &= \frac{x}{1+\frac{z}{d}} \\ y_p &= \frac{y}{1+\frac{z}{d}} \\ z_p &= 0\,. \end{aligned}$$

Pertanto ora la "matrice" $M_d^{(c)}$ della trasformazione prospettica è la classe di equivalenza, cioè l'elemento in $PM_4(\mathbb{R})$, che ha per rappresentante la matrice

$$M_d^{(c)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{pmatrix} . \tag{16.2.3}$$

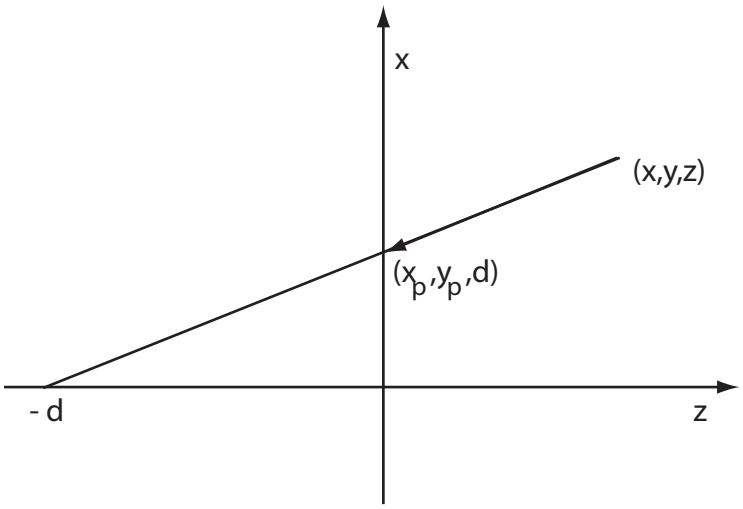


FIGURA 16.2.2. Proiezione della prospettiva centrale: piano di visuale in $z = 0$

Poiché gli elementi di $PM_4(\mathbb{R})$ sono classi di equivalenza per dilatazione, possiamo scegliere un altro rappresentante della stessa classe, dilatando quello appena scritto in modo da eliminare il denominatore. Così si ottiene la seguente forma della matrice della trasformazione prospettica:

$$M_d^{(c)} = \begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{pmatrix} . \tag{16.2.4}$$

Si noti che questa forma corrisponde nel modo più naturale alle equazioni (16.2.2) della trasformazione.

ESERCIZIO 16.2.1. Consideriamo la prospettiva centrale con centro di proiezione ubicato sull'asse z al punto $z = -d$ e piano di proiezione $\{z = 0\}$, la quale porta alla matrice di proiezione prospettica (16.2.3). Sia S una superficie emisferica di raggio 1 nel semispazio $\{x \geqslant 0\}$ con centro nel punto $(0, 0, 2)$, e C un cilindro di raggio $\frac{1}{2}$ avente per asse centrale la retta $\{y = 0, z = 2\}$. Sia $J = S \cap C$. Calcolare l'immagine prospettica di J sul piano di proiezione.

Svolgimento. Anzitutto determiniamo J. L'equazione di S è $x^2+y^2+(z-2)^2 = 1, x \geqslant 0$. L'equazione di C è $y^2+(z-2)^2 = \frac{1}{4}$. Quindi i punti di J sono tutti e soli quelli che soddisfano le seguenti equazioni e disequazioni:

$$\begin{aligned} x &\geqslant 0 \\ x^2 + y^2 + (z-2)^2 &= 1 \\ y^2 + (z-2)^2 &= \frac{1}{4} . \end{aligned}$$

Se ne ricava

$$x^2 = \frac{3}{4}$$

$$x \geqslant 0$$

$$y^2 + (z-2)^2 = \frac{1}{4}$$

ossia $x = \sqrt{3}/2$, $y^2 + (z-2)^2 = \frac{1}{4}$. Si tratta, ovviamente, di una circonferenza sul piano $x = \sqrt{3}/2$, che parametrizziamo nel modo seguente:

$$x = \sqrt{3}/2 \tag{16.2.5}$$

$$y = \frac{1}{2}\cos t \tag{16.2.6}$$

$$z = 2 + \frac{\sin t}{2} \tag{16.2.7}$$

dove l'angolo t varia fra 0 e 2π. Scriviamo i punti di J in termini di coordinate omogenee (x, y, z, w) con quarta coordinata $w = 1$ (scelta del rappresentante stereografico per punti al finito), ed applichiamo la matrice di proiezione prospettica (16.2.3). Analogamente a (16.2.1), da (16.2.3) e (16.2.5) si ottiene

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = M_d^{(c)} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{pmatrix} \begin{bmatrix} \sqrt{3}/2 \\ \frac{1}{2}\cos t \\ 1 + \frac{1}{2}\sin t \\ 1 \end{bmatrix} \tag{16.2.8}$$

$$= \begin{bmatrix} \sqrt{3}/2 \\ \frac{1}{2}\cos t \\ 0 \\ 1 + \frac{1+\frac{1}{2}\sin t}{d} \end{bmatrix} . \tag{16.2.9}$$

Ora riportiamo in coordinate omogenee con quarta componente 1 il punto proiettivo (X, Y, Z, W) così calcolato, dividendo per il valore di W: si ottiene così un altro rappresentante della classe di equivalenza proiettiva $[X, Y, Z, W]$, e precisamente il punto $(x, y, z, 1)$ dove

$$x = \frac{d}{d + 1 + \frac{1}{2}\sin t} \; \frac{\sqrt{3}}{2}$$

$$y = \frac{d\cos t}{2(d + 2 + \frac{1}{2}\sin t)}$$

$$z = 0\,.$$

Quindi la curva immagine di J sul piano di proiezione $\{z = 0\}$, munito delle coordinate x e y, ha la seguente equazione:

$$x = \frac{\sqrt{3}d}{2d + 4 + \sin t}$$

$$y = \frac{d\cos t}{2d + 4 + \sin t}\,.$$

Si osservi che, se facciamo tendere d a $-\infty$ (ossia se passiamo alla proiezione ortogonale sul piano $\{z = 0\}$, si ottiene, come previsto, $x = \sqrt{3}/2$, $y = \frac{1}{2}\cos t$: al variare di t questo punto immagine

si muove sul segmento $x = \sqrt{3}/2$, $-1/2 \leqslant y \leqslant 1/2$, che è esattamente la proiezione ortogonale dell'intersezione fra sfera e cilindro, perché essa è una circonferenza nello spazio tridimensionale che giace sul piano $x = \sqrt{3}/2$, ha componente y del centro ufguale a 0 e raggio $1/2$. □

NOTA 16.2.2. **(Punto di fuga della proiezione standard.)** È interessante osservare che, in questo caso particolare della proiezione prospettica centrale, che si chiama la *proiezione standard*, il piano di visuale è ortogonale alla direzione dall'osservatore all'origine. Consideriamo un fascio di rette perpendicolare al piano di visuale: nella proiezione standard nella forma appena sviluppata, si tratta del fascio delle rette parallele all'asse z. Chiamiamo $\mathbf{r_0} = (x_0, y_0, 0)$ il punto in cui una tale retta $\mathbf{r}$ interseca il piano $\{z = 0\}$: allora la equazione parametrica di $\mathbf{r}$ è $\mathbf{r}(t) = \mathbf{r_0} + t\mathbf{e_3}$. Scegliamo $\mathbf{r_0}$ nel piano di proiezione $\{z = 0\}$, diciamo $\mathbf{r_0} = (a, b, 0)$, e poniamo $\mathbf{r}(t) = (x(t), y(t), z(t)) = (a, b, t)$. Associamo ai punti della retta $\mathbf{r}$ i rappresentanti speciali delle loro classi di equivalenza proiettiva, ossia, in coordinate omogenee, $\mathbf{R}(t) = (x(t), y(t), z(t), 1) = (a, b, t, 1)$; analogamente, scriviamo $\mathbf{S}$ per la corrispondente estensione quadridimensionale di ogni altra retta $\mathbf{s}$.
Quando applichiamo la trasformazione trovata in (16.2.4), la retta $\mathbf{R}$ viene trasformata nella retta $\mathbf{S} = T\mathbf{R}$ di equazioni parametriche

$$\mathbf{S}(t) = M_d^{(c)}\mathbf{R}(t) = \begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{pmatrix} \begin{pmatrix} a \\ b \\ t \\ 1 \end{pmatrix} = \begin{pmatrix} da \\ db \\ 0 \\ t + d \end{pmatrix} .$$

Riconduciamo il risultato alla consueta espressione stereografica delle classi di equivalenza proiettive, ossia con i rappresentanti speciali di ultima coordinata 1, rinormalizzando con la divisione per $t + d$. Si ottiene

$$\mathbf{S}(t) = \left(\frac{da}{t+d}, \frac{db}{t+d}, 0, 1 \right) . \tag{16.2.10}$$

La retta immagine tridimensionale $\mathbf{s}$, che si ottiene considerando le prime tre componenti dopo questa normalizzazione, è

$$\mathbf{s}(t) = \left(\frac{da}{t+d}, \frac{db}{t+d}, 0 \right) . \tag{16.2.11}$$

Osserviamo che essa giace, come deve essere, nel piano $\{z = 0\}$ (il piano di visuale), e quando $t \to \infty$ tende all'origine, quali che siano a e b. In altre parole, il fascio di rette parallele perpendicolare al piano di visuale viene trasformato nel fascio delle semirette radiali nel piano $\{z = 0\}$. L'origine è quindi il *punto di fuga* prospettico di questo fascio di rette.
Osserviamo che l'origine è esattamente il punto ottenuto sommando al centro di proiezione il versore direzionale del fascio di rette moltiplicato per la distanza d fra il centro di proiezione ed il piano di visuale. Se avessimo scelto un centro di proiezione diverso da $(0, 0, -d)$, avremmo potuto ripetere lo stesso calcolo della trasformazione prospettica, che svolgeremo in dettaglio nella prossima Sezione 16.5. A partire da esso, riprenderemo in esame e generalizzeremo questo risultato nella Sezione 16.6.

Per finire, consideriamo come è fatta la proiezione prospettica di una retta che *attraversa* il piano di visuale, riconsiderando la retta immagine $\mathbf{s}$ del fascio ortogonale al piano di visuale, $\mathbf{r}(t) = (a, b, t)$, che è stata calcolata in (16.2.11): $\mathbf{s}(t) = \left(\frac{da}{t+d}, \frac{db}{t+d}, 0 \right)$. Facciamo variare il parametro t in modo che il punto $\mathbf{r}(t)$ passi dal semispazio anteriore all'osservatore posto nel centro di prospettiva $(0, 0, -d)$ a quello posteriore, ossia da $t > -d$ a $t < -d$. Quando t decresce verso $-d$ il punto $\mathbf{s}(t)$ si muove nel piano di visuale radialmente fuori dall'origine (ossia il punto di fuga) verso l'infinito. Quando t diventa inferiore a $-d$ e continua a decrescere, il punto $\mathbf{s}(t)$ salta dalla parte opposta, sulla stessa retta radiale ma sulla semiretta opposta (come succede ai raggi che passano per una lente quando la scena e l'osservatore sono ai lati opposti dell'obiettivo, che è il centro di proiezione), e si avvicina

dall'infinito al punto di fuga. Quindi una retta che attraversa il piano di visuale ha immagine che va all'infinito con un salto. Questo fatto crea una distorsione prospettica assai drastica: si provi ad immaginare l'immagine prospettica del cubo unitario quando l'osservatore si trova dentro il cubo! Per quato motivo, in Computer Graphics, il centro di prospettiva, ossia l'osservatore, si trova sempre da un lato del piano di visuale e la scena osservata dall'altro lato. □

ESEMPIO 16.2.3. **(Trasformazione prospettica standard del cubo unitario.)** Consideriamo il cubo Q i cui vertici sono $\{\pm\mathbf{e_1}+\pm\mathbf{e_2}+\pm\mathbf{e_3}\}$. È improprio chiamare unitario questo cubo, perché ha lato 2, ma possiamo sempre dividere per due alla fine se proprio vogliamo, e quindi procediamo con questa scelta, che evita i denominatori. Consideriamo le classi di equivalenza proiettiva dei vettori $\mathbf{e_i}$, scrivendo ad esempio i loro rappresentanti standard come $\pm\mathbf{E_1}^{\pm} = (\pm1, 0, 0, 1)$, $\mathbf{E_2}^{\pm} = (0, \pm1, 0, 1)$, $\mathbf{E_3}^{\pm} = (0, 0, \pm1, 1)$. Applichiamo agli $\mathbf{E_i}$ la matrice $M_d^{(c)}$ in (16.2.4) e scriviamo $\mathbf{V_i}^{\pm} = M_d^{(c)}\mathbf{E_i}^{\pm}$. Si ottiene $\mathbf{V_i}^{\pm} = \mathbf{E_i}^{\pm}$ per $i = 1$ e 2, e$\mathbf{V_3}^{\pm} = (0, 0, 0, d \pm 1)$.

Rinormalizzando per ritornare ai rappresentanti proiettivi standard con 1 alla quarta componente, e considerando solo le prime tre componenti del risultato per ritrovare i vettori tridimensionali trasformati $\mathbf{v_i}$, e scrivendo M per la trasformazione prospettica tridimensionale in tal modo ottenuta da $M_d^{(c)}$, si verifica subito che si ha

$$\begin{aligned}
T(1,1,1) &= \frac{1}{d+1}(1,1,0)\\
T(1,1,-1) &= \frac{1}{d-1}(1,1,0)\\
T(1,-1,1) &= \frac{1}{d+1}(1,-1,0)\\
T(1,-1,-1) &= \frac{1}{d-1}(1,-1,0)\\
T(-1,1,1) &= \frac{1}{d+1}(-1,1,0)\\
T(-1,1,-1) &= \frac{1}{d-1}(-1,1,0)\\
T(-1,-1,1) &= \frac{1}{d+1}(-1,-1,0)\\
T(-1,-1,-1) &= \frac{1}{d-1}(-1,-1,0)\,.
\end{aligned}$$

dall'osservatore (ovvero i punti $(\pm1, \pm1, -1)$. Abbiamo appena visto che essi vengono compressi di un fattore $1/(d-1)$, mentre i quattro vertici posteriori di un fattore più grande, $1/(d+1)$. Questa è la compressione prospettica.

Fin qui abbiamo supposto $d > 1$. Se $0 < d < 1$ allora $d - 1 < 0$ ed i vertici anteriori sono in realtà alle spalle dell'osservatore, e vengono quindi scambiati di segno, ossia ribaltati, come succede ai raggi che passano per una lente quando la scena e l'osservatore sono ai lati opposti dell'obiettivo (centro di proiezione). Se poi $d = 1$, allora i vertici anteriori sono ai lati dell'osservatore, e la proiezione prospettica non è definita su di essi (le semirette da essi al centro di proiezione non passano per il piano di visuale $\{z = 0\}$, restano tutte nel piano $\{z = -1\}$. Osserviamo come è fatta la proiezione di una retta che attraversa il piano dell'osservatore, riconsiderando la retta immagine $\mathbf{s}$ del fascio ortogonale al piano di visuale, $\mathbf{r}(t) = (a, b, t)$, che è stata calcolata in (16.2.11): $\mathbf{s}(t) = (da/(t+d), db/(t+d), 0)$. Facciamo variare il parametro t in modo che il punto $\mathbf{r}(t)$ passi dal semispazio anteriore all'osservatore posto nel centro di prospettiva $(0, 0, -d)$ a quello posteriore, ossia da $t > -d$ a $t < -d$. Quando t decresce verso $-d$ il punto $\mathbf{s}(t)$ si muove nel piano di visuale radialmente fuori dall'origine (ossia il punto di fuga) verso l'infinito: più precisamente, verso il punto

all'infinito di questo piano le cui coordinate proiettive sono $(a, b, 0)$. Quando t diventa inferiore a $-d$ e contimua a decrescere, il punto $\mathbf{s}(t)$ salta dalla parte opposta, sulla stessa retta radiale ma sulla semiretta opposta, e si avvicina dall'infinito al punto di fuga. Quindi una retta che attraversa il piano di visuale ha immagine che va all'infinito con un salto. Questo fatto crea una ditorsione prospettica assai drastica: si provi ad immaginare l'immagine prospettica del cubo unitario quando l'osservatore si trova dentro il cubo! Per questo motivo, in Computer Graphics, il centro di prospettiva, ossia l'osservatore, si trova sempre da un lato del piano di visuale e la scena osservata dall'altro lato. □

16.3. Proiezione prospettica ortogonale (o ortografica)

La proiezione ortogonale (detta anche ortografica) è la trasformazione prospettica data dalla proiezione perpendicolare al piano di visuale, che potrebbe essere un piano che non contiene l'origine, quindi non un sottospazio vettoriale ma un suo traslato (ed in tal caso l'origine non può essere preservata dalla proiezione, che pertanto non è una applicazione lineare). Però manteniamo la scelta di localizzazione del piano di visuale fatta alla fine della precedente Sezione 16.5, e quindi il piano di visuale passa per l'origine: è il piano $\{z = 0\}$, quindi un sottospazio. La proiezione ortogonale all'asse z è quindi la trasformazione che manda il punto $\mathbf{p} = (x, y, z)$ nel punto di tale piano ottenuto ponendo uguale a zero la componente z, cioè $(x, y, 0)$. In coordinate omogenee stiamo ponendo

$$\begin{aligned} x_0 &= x \\ y_0 &= y \\ z_0 &= 0 \\ w_0 &= w \end{aligned}$$

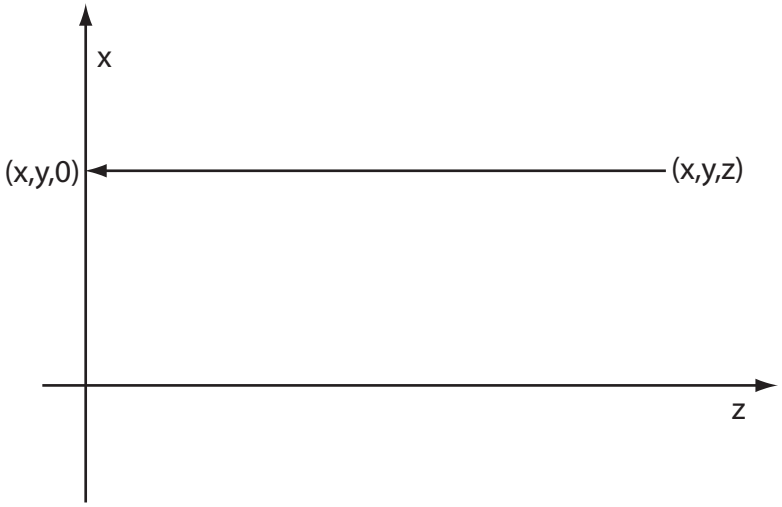


FIGURA 16.3.1. Proiezione della prospettiva ortogonale sul piano $z = 0$

Pertanto la "matrice" prospettica ora è

$$M^{(o)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \tag{16.3.1}$$

NOTA 16.3.1. Si osservi che la matrice $M^{(o)}$ si ottiene facendo tendere d ad infinito nell'espressione per $M_d^{(c)}$ (si veda anche come, nell'Esempio 16.2.3, il trasformato prospettico del cubo unitario in $\mathbb{R}^3$ tende al cubo unitario nel piano di proiezione $\mathbb{R}^2$ quando $d \to \infty$, e quindi tende alla proiezione ortogonale del cubo). In effetti questo fatto è naturale, perchè i raggi prospettici in questo tipo di prospettiva sono tutti paralleli all'asse di proiezione (nel nostro caso l'asse z, e quindi non convergono verso un centro di proiezione (punto di fuga) al finito, bensì verso un punto di fuga all'infinito (la direzione proiettiva dell'asse z, cioè, in coordinate omogenee, $[0, 0, 1, 0]$). Questo equivale a dire che la distanza d fra il centro di proiezione e l'origine tende ad infinito. □

16.4. Un'unica matrice per prospettiva centrale e ortogonale

Nelle due trasformazioni prospettiche viste nelle Sezioni precedenti, la matrice della prospettiva centrale $M_d^{(c)}$ è quella che si applica quando il centro di proiezione è a distanza $d < \infty$ dal piano di visuale, mentre la matrice della proiezione ortogonale M^o si applica quando il centro di proiezione è all'infinito. È possibile unificare questi due casi nelle seguente formulazione più generale di una trasformazione prospettica con un unico punto di fuga (al finito o all'infinito).

Sia $\mathbf{p} = (x, y, z)$ un punto generico da trasformare prospetticamente, sia $\mathbf{c}$ il centro di proiezione, ubicato in un punto arbitrario dello spazio, e collochiamo il piano di visuale in posizione $z = z_p$ (questa è la situazione abituale della Computer Graphics, nella quale il piano di visuale è ortogonale all'asse z). Consideriamo il punto $\mathbf{b}_p = (0, 0, z_p)$ dove questo piano interseca l'asse z, e sia $q = \|\mathbf{c}-\mathbf{b}_p\|$ e $\mathbf{d}$ il versore $\mathbf{d} = \frac{1}{q}(\mathbf{c} - \mathbf{b}_p)$. Come prima, denotiamo con $\mathbf{p}_p = (x_p, y_p, z_p)$ il punto sul piano di visuale ottenuto proiettando prospetticamente (con centro in $\mathbf{c}$) il generico punto $\mathbf{x} = (x, y, z)$. Allora $\mathbf{p}_p$ appartiene alla retta di equazione parametrica

$$\mathbf{r}(t) = (x'(t), y'(t), z'(t)) = \mathbf{c} + t(\mathbf{x} - \mathbf{c}) \tag{16.4.1}$$

per qualche $0 \leqslant t \leqslant 1$.

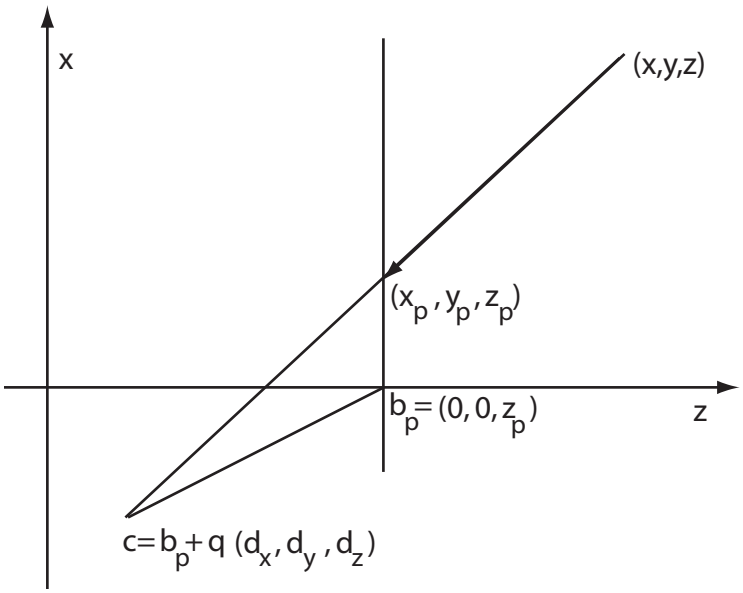


FIGURA 16.4.1. La proiezione prospettica: caso generale di punto di fuga al finito

Usando il fatto che $\mathbf{c} = \mathbf{b}_p + q\mathbf{d}$ ricaviamo da (16.4.1) che i punti del segmento parametrico verificano

$$\begin{aligned} x'(t) &= qd_x + (x - qd_x)t \\ y'(t) &= qd_y + (x - qd_y)t \\ z'(t) &= z_p + qd_z + (z - (z_p + qd_z))\, t \end{aligned}$$

Ora ricaviamo $\mathbf{p}_p$ ponendo $z' = z_p$ nell'ultima uguaglianza. In tal modo si ottiene il valore appropriato di t:

$$t = -\frac{qd_z}{z - (z_p + qd_z)}\,,$$

che sostituito nelle uguaglianze precedenti dà

$$x_p = x'(t) = qd_x + \frac{(x - qd_x)qd_z}{qd_z + z_p - z} = qd_x + \frac{x - qd_x}{1 + \frac{z_p - z}{qd_z}}$$

$$= \frac{x + \frac{z_p - z}{qd_z}}{1 + \frac{z_p - z}{qd_z}} = \frac{x - \frac{z}{qd_z} + \frac{z_p}{qd_z}}{1 + \frac{z_p - z}{qd_z}}$$

ed analogamente per la coordinata y, mentre, ovviamente, risulta $z'(t) = z_p$. Quindi alla fine si ottiene:

$$x_p = \frac{x - \frac{z}{qd_z} + \frac{z_p}{qd_z}}{1 + \frac{z_p - z}{qd_z}}$$

$$y_p = \frac{y - \frac{z}{qd_z} + \frac{z_p}{qd_z}}{1 + \frac{z_p - z}{qd_z}}$$

$$z_p = z_p \frac{1 + \frac{z_p - z}{qd_z}}{1 + \frac{z_p - z}{qd_z}} = \frac{-z\frac{z_p}{qd_z} + \frac{z_p(1 + qd_z)}{qd_z}}{1 + \frac{z_p - z}{qd_z}}\,.$$

16.5. Forma matriciale generale della prospettiva centrale

Nella precedente Sezione 16.4 abbiamo visto la forma della matrice prospettica quando il centro di prospettiva è arbitrario ma, come di consuetudine in Computer Graphics, il piano di visuale è ortogonale all'asse z. Anche se questa è la consuetudine, in una animazione nella quale l'osservatore si sposta può essere necessario spostare anche il piano di visuale al fine di vedere parti interessanti della scena. Si pensi ad esempio ad un videogioco nel quale l'osservatore (ossia il centro di proiezione) si muove in un labirinto, o anche in un appartamento dove attraversa una porta, ed ha bisogno di guardare a sinistra e a destra per vedere se ci sono nemici: mentre attravesra la porta, i suoi lati sinistro e destro giacciono nel piano di visuale o nelle sue immediate vicinanze, e quindi (Esempio 16.2.3) non sono visibili (sono mandati all'infinito dalla proiezione) oppure sono estremamente distorti. È come se l'osservatore guardasse con la coda dell'occhio: per vedere bene deve gitrare la testa, o, nel nostro caso, il piano di visuale. In questa Sezione deriviamo la forma generale della trasformazione prospettica centrale nella quale anche il piano di visuale è scelto arbitrariamente.

Cominciamo con un caso particolare che viola il nostro precedente proposito (legato al confronto con la prospettiva ortogonale) di collocare il centro di prospettiva in $(0, 0, -d)$, ed invece lo ricolloca nell'origine: ma siccome alla fine dovremo traslarlo ad un punto arbitrario, questa scelta ormai non comporta svantaggi, ed anzi rende più semplice effettuare in seguito la traslazione.

PROPOSIZIONE 16.5.1. **(Prospettiva centrale con centro nell'origine e piano di proiezione arbitrario.)** *La trasformazione prospettica con centro nell'origine e piano di visuale P a distanza*

d_0 dall'origine e con versore normale $\mathbf{n} = (n_x, n_y, n_z)$ è espressa, in coordinate proiettive (ovvero omogenee), dalla matrice

$$M_{\mathbf{0}} = \begin{pmatrix} d_0 & 0 & 0 & 0 \\ 0 & d_0 & 0 & 0 \\ 0 & 0 & d_0 & 0 \\ n_x & n_y & n_z & 0 \end{pmatrix} .$$

*Dimostrazione*Poiché il centro di proiezione è l'origine, la trasformazione manda il punto generico $\mathbf{p} = (x, y, z)$ in

$$\mathbf{p}' = \alpha\mathbf{p} \in P\,. \tag{16.5.1}$$

Osserviamo che la distanza del piano P dall'origine è, a parte il segno, $d_0 = \mathbf{p}' \cdot \mathbf{n}$. Quindi $\alpha = d_0/\mathbf{p} \cdot \mathbf{n} = d_0/(n_x x + n_y y + n_z z)$. La divisione per $n_x x + n_y y + n_z z$ non è altro che la divisione prospettica per la distanza (in questo caso fra P e l'origine) se, in coordinate omogenee, si scrive la matrice $M_{\mathbf{0}}$ come nell'enunciato. □

Ora trattiamo il caso generale in cui anche il centro di prospettiva è arbitrario. A questo scopo è utile la seguente Nota, che mette in rilievo la mancanza di linearità delle trasformazioni proiettive pensate come trasformazioni dello spazio tridimensionale visto in coordinate omogenee quadridimensionali.

NOTA 16.5.2. Siano $\mathbf{p} = (p_x, p_y, p_z) \in \mathbb{R}^3$, $\mathbf{q} = (q_x, q_y, q_z)$, $\mathbf{r} = (r_x, r_y, r_z)$ e si consideri la classe di equivalenza proiettiva $\overline{\mathbf{p}}$ di $(p_x, p_y, p_z, 1)$. Se due matrici M_1 e M_2 a quattro dimensioni mandano, rispettivamente $(p_x, p_y, p_z, 1)$ in (q_x, q_y, q_z, a) e (r_x, r_y, r_z, b), allora la matrice $M_1 + M_2$ manda $(p_x, p_y, p_z, 1)$ in $(q_x + r_x, q_y + r_y, q_z + r_z, a + b)$. Se a, b , $a + b \neq 0$ possiamo considerare i rappresentanti stereografici, ossia i rappresentanti speciali con ultima coordinata 1, e quanto appena visto corrisponde a dire che le corrispondenti trasformazioni proiettive mandano la classe di equivalenza di $\mathbf{p}$ rispettivamente nelle classi di $\mathbf{q}/a$ e $\mathbf{r}/b$, e la loro somma manda la classe di $\mathbf{p}$ nella classe di $(\mathbf{q}+\mathbf{r})/(a+b)$ (in questo caso, come d'abitudine, le traasformazioni sullo spazio proiettivo non sono lineari, nel senso che non mandano la classe della somma di due vettori nella somma delle ripsettive classi). Ma se in particolare le due matrici M_1 e M_2 hanno la stessa ultima riga, e quindi $a = b$ (e continuiamo as assumere $a \neq 0$), allora la matrice proiettiva con la stessa ultima riga di M_1 e M_2 e con le prime tre righe date dalla somma termine a termine di M_1 e M_2 manda la classe di $(p_x, p_y, p_z, 1)$ nella somma delle classi di $(q_x/a, q_y/a, q_z/a, 1)$ e $(r_x/a, r_y/a, r_z/a, 1)$ (in particolare, se $a = 1$, la matrice rispetta la somma sullo spazio proiettivo). □

NOTA 16.5.3. ***(Il significato geometrico del fattore di distanza.)*** Abbiamo ripetutamente incontrato, ad esempio nella Proposizione 16.5.1, il termine $d_0 = \mathbf{p}' \cdot \mathbf{n}$, dove $\mathbf{p}'$ è un generico punto del piano di proiezione P. Chiaramente, $|d_0|$ è la distanza fra P e l'origine, ed il segno di d_0 è positivo se il versore normale $\mathbf{n}$ di P punta nel semispazio opposto all'origine. Nella suddetta Proposizione, l'origine è il centro di proiezione, e quindi la consuetudine della Computer Graphics è che la scena giaccia nel semispazio opposto, per evitare drastiche distorsioni prospettiche, come osservato alla fine dell'Esempio 16.2.3. Quindi in questo caso d_0 è positivo e misura la distanza dall'origine al piano.
Nei prossimi enunciati sposteremo il centro di proiezione dall'origine ad un punto arbitrario $\mathbf{c}$ ed introdurremo la costante $d_1 = (\mathbf{c}\,, \mathbf{n})$. Chiaramente, d_1 è la distanza fra i piani paralleli a P che passano per l'origine e per $\mathbf{c}$. Chiamiamo questi due piani paralleli P_0 e P_c, rispettivamente. Il segno di d_1 è positivo se il centro di proiezione $\mathbf{c}$ giace in quello dei due semispazi determinati da P_0 verso cui punta il versore normale $\mathbf{n}$ (qualche volta si parla del *semispazio positivo* determinato dalla scelta del versore normale).
Infine, introdurremo la quantità $d = d_0 - d_1$, che ha un ruolo significativo nella matrice della prospettiva centrale nel caso generale. Il significato geometrico di questa costante è comprensibile se

consideriamo due casi. Il primo caso è quello in cui il piano P ed il centro $\mathbf{c}$ giacciono in semispazi opposti rispetto a P_0. In tal caso i segni di d_0 e d_1 sono opposti, e quindi d, a parte il segno, misura la somma delle distanze fra P e P_0 e fra P_0 e P_c, ossia la distanza fra P ed il centro di proiezione $\mathbf{c}$ (esattamente come succedeva per d_0 nel caso della Proposizione 16.5.1). Nel secondo caso, i segni di d_0 e d_1 sono uguali, e quindi d misura la differenza delle suddette distanze. Però ora il piano P_c giace fra P e P_0, e quindi d misura nuovamente, a parte il segno, la distanza fra P ed il centro di proiezione $\mathbf{c}$. Il segno di d è positivo o negativo a seconda della scelta di verso del versore $\mathbf{n}$, e più precisamente è positivo se il centro di proiezione (ossia l'osservatore) sta nel semispazio *negativo* deteminato dal piano di visuale P. □

TEOREMA 16.5.4. **(Prospettiva centrale con centro e piano di proiezione arbitrari.)** *Consideriamo la trasformazione prospettica con centro in un punto $\mathbf{c} = (c_x, c_y, c_z)$ e piano di visuale P con versore normale $\mathbf{n} = (n_x, n_y, n_z)$ a distanza con segno d_0 dall'origine (ossia $d_0 = \mathbf{p_0} \cdot \mathbf{n}$ per qualsiasi punto $\mathbf{p_0} \in P$). Sia d_1 la distanza con segno fra i piani paralleli a P che passano per l'origine e per $\mathbf{c}$, rispettivamente (ossia $d_1 = \mathbf{c} \cdot \mathbf{n}$), e sia $d = d_0 - d_1$. Allora la trasformazione è espressa, in coordinate proiettive, dalla matrice $M_\mathbf{c}$, che in questo Teorema scriviamo brevemente M, data da*

$$M = M_\mathbf{c} = \begin{pmatrix} d + c_x n_x & c_x n_y & c_x n_z & -c_x d_0 \\ c_y n_x & d + c_y n_y & c_y n_z & -c_y d_0 \\ c_z n_x & c_z n_y & d + c_z n_z & -c_z d_0 \\ n_x & n_y & n_z & -d_x \end{pmatrix} .$$

*Dimostrazione*Potremmo svolgere la dimostrazione riportando il centro di proiezione all'origine mediante una traslazione, applicando allora la trasformazione prospettica la cui matrice è stata determinata nella Proposizione 16.5.1 ed infine ritornando indietro con la traslazione opposta. Questi calcoli li svolgeremo nel successivo Esercizio 16.5.5 qui invece preferiamo sviluppare il calcolo diretto in analogia alla dimostrazione della Proposizione 16.5.1.

Poiché ora la trasformazione prospettica proietta radialmente verso $\mathbf{c}$ il punto generico $\mathbf{p} = (x, y, z)$ in un punto $\mathbf{p}'$, l'identità (16.5.1) diventa

$$\mathbf{p}' - \mathbf{c} = \alpha(\mathbf{p} - \mathbf{c}) . \tag{16.5.2}$$

Pertanto $\mathbf{n} \cdot (\mathbf{p}' - \mathbf{c}) = \alpha \mathbf{n} \cdot (\mathbf{p} - \mathbf{c})$, da cui

$$\alpha = \frac{\mathbf{n} \cdot \mathbf{p}' - \mathbf{n} \cdot \mathbf{c}}{\mathbf{n} \cdot \mathbf{p} - \mathbf{n} \cdot \mathbf{c}} = \frac{d_0 - d_1}{\mathbf{n} \cdot \mathbf{p} - d_1} = \frac{d}{\mathbf{n} \cdot \mathbf{p} - d_1} , \tag{16.5.3}$$

perché, per definizione, $d_0 = \mathbf{n} \cdot \mathbf{p}'$, $d_1 = \mathbf{n} \cdot \mathbf{c}$ e $d = d_0 - d_1$.
Riscriviamo la trasformazione prospettica (16.5.2) come

$$\mathbf{p}' = T\mathbf{p} = \mathbf{c} + \alpha(\mathbf{p} - \mathbf{c}) = \alpha\mathbf{p} + (1 - \alpha)\mathbf{c}$$

ovvero $T = T_1 + T_2$, dove $T_1\mathbf{p} = \alpha\mathbf{p}$ e $T_2\mathbf{p} = (1 - \alpha)\mathbf{c}$. Consideriamo le trasformazioni proiettive $\widetilde{T}_1$ e $\widetilde{T}_2$ indotte da T_1 e T_2 pensate in coordinate omogenee. Scriviamo le matrici (a dimensione 4) M_1 e M_2 associate alle trasformazioni proiettive $\widetilde{T}_1$ e $\widetilde{T}_2$. È chiaro da (16.5.3) che la prima di esse è

$$M_1 = \begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ n_x & n_y & n_z & -d_1 \end{pmatrix} .$$

Per scrivere M_2 osserviamo che da (16.5.3) segue

$$1 - \alpha = 1 - \frac{d}{\mathbf{n} \cdot \mathbf{p} - d_1} = \frac{\mathbf{n} \cdot \mathbf{p} - d_0}{\mathbf{n} \cdot \mathbf{p} - d_1} .$$

Pertanto $\widetilde{T}_2$ diventa

$$\begin{aligned}\widetilde{T}_2(x,y,z,1) &= ((1-\alpha)c_x,(1-\alpha)c_y,(1-\alpha)c_z,1)\\ &\sim ((\mathbf{n}\cdot\mathbf{p}-d_0)c_x,(\mathbf{n}\cdot\mathbf{p}-d_0)c_y,(\mathbf{n}\cdot\mathbf{p}-d_0)c_z,\mathbf{n}\cdot\mathbf{p}-d_1)\\ &= ((n_xx+n_yy+n_zz)c_x-d_0c_x,(n_xx+n_yy+n_zz)c_y-d_0c_y,\\ &\quad (n_xx+n_yy+n_zz)c_z-d_0c_z,n_xx+n_yy+n_zz-d_1)\,,\end{aligned}$$

e quindi

$$M_2 = \begin{pmatrix} c_xn_x & c_xn_y & c_xn_z & -c_xd_0\\ c_yn_x & c_yn_y & c_yn_z & -c_yd_0\\ c_zn_x & c_zn_y & c_zn_z & -c_zd_0\\ n_x & n_y & n_z & -d_1\end{pmatrix}\,.$$

L'enunciato ora segue dalla Nota 16.5.2. □

Esercizio 16.5.5. . Come annunciato all'inizio della dimostrazione del Teorema 16.5.4, ora diamo una formulazione alternativa di quella dimostrazione riportando all'origine il centro di proiezione $\mathbf{c}$ mediante la traslazione $\Lambda(x,y,z)=(x-c_x,y-c_y,z-c_z)$, poi applicando la trasformazione prospettica con centro l'origine della Proposizione 16.5.1 ed infine ritornando indietro con la traslazione opposta Λ^{-1}.

Svolgimento. La matrice della trasformazione affine associata alla traslazione Λ è

$$\Lambda = \begin{pmatrix} 1 & 0 & 0 & -c_x\\ 0 & 1 & 0 & -c_y\\ 0 & 0 & 1 & -c_z\\ 0 & 0 & 0 & 1\end{pmatrix}$$

e la sua inversa ovviamente è la traslazione opposta,

$$\Lambda^{-1} = \begin{pmatrix} 1 & 0 & 0 & c_x\\ 0 & 1 & 0 & c_y\\ 0 & 0 & 1 & c_z\\ 0 & 0 & 0 & 1\end{pmatrix}\,.$$

Per scrivere la matrice $M_{\mathbf{0}}$ abbiamo bisogno della distanza con segno fra il piano di proiezione e l'origine *dopo aver applicato la traslazione* Λ. Nella Proposizione 16.5.1 questa distanza, chiaramente, era $d_0=\mathbf{n}\cdot\mathbf{p}'$, dove $\mathbf{p}'$ è un punto qualsiasi del piano di proiezione. La traslazione manda $\mathbf{p}'$ in $\mathbf{p}'-\mathbf{c}$, e quindi d_0 in

$$\mathbf{n}\cdot\mathbf{p}'-\mathbf{c}=\mathbf{n}\cdot\mathbf{p}'-\mathbf{n}\cdot\mathbf{c}=d_0-d_1=d$$

con la notazione del precedente Teorema 16.5.4. Pertanto, la matrice della Proposizione 16.5.1 ora diventa

$$M_{\mathbf{0}} = \begin{pmatrix} d & 0 & 0 & 0\\ 0 & d & 0 & 0\\ 0 & 0 & d & 0\\ n_x & n_y & n_z & 0\end{pmatrix}\,.$$

È ora immediato verificare che la composizione di matrici $\Lambda^{-1}\,M_{\mathbf{0}}\,\Lambda$ (si badi all'ordine!) è esattamente la matrice $M_{\mathbf{c}}$ del Teorema 16.5.4. □

16.6. Punti di fuga della prospettiva centrale

Abbiamo introdotto il concetto di punto di fuga per la proiezione standard nella Nota 16.2.2. Rammentiamo che si tratta del punto in cui si incontrano le immagini sotto la trasformazione prospettica di un fascio di rette parallele, in quel caso ortogonali al piano di visuale, e quindi orientate secondo il versore normale dell'azze z. Ora, grazie al Teorema 16.5.4, possiamo estendere quel risultato al caso della prospettiva centrale generale, per un fascio di rette parallele diretto secondo un versore arbitrario $\mathbf{u} = (u_x, u_y, u_z)$, non necessariamente ortogonale al piano di visuale P. Come prima, sia $\mathbf{c} = (c_x, c_y, c_z)$ il centro di proiezione, e $\mathbf{n}$ il versore normale a P. Per qualsiasi punto $\mathbf{p_0} = (x_0, y_0, z_0) \in P$, la distanza (con segno) di P dall'origine è $d_0 = \mathbf{n} \cdot \mathbf{p_0}$, e la distanza (con segno) fra i piani paralleli a P che passano, rispettivamente, per l'origine e per $\mathbf{c}$ è $d_1 = \mathbf{n} \cdot \mathbf{c}$. Poniamo ancora $d = d_0 - d_1$.

Teorema 16.6.1. *Chiamiamo* $\mathbf{r}$ *le rette parallele dirette secondo il versore* $\mathbf{u}$*:* $\mathbf{r}(t) = \mathbf{r^0} + t\mathbf{u}$*, con* $\mathbf{r^0}$ *arbitrario: ma senza perdita di generalità (come già nella Nota 16.2.2), se il fascio non è diretto parallelamente al piano di proiezione* P*, scegliamo* $\mathbf{r^0}$ *come la intersezione della retta* $\mathbf{r}$ *con* P*; se il fascio è parallelo a* P *possiamo considerare una retta del fascio che giace in* P *e quindi, anche in questo caso scegliere (non più in modo unico)* $\mathbf{r^0} \in P$*.*
Se il fascio non è diretto parallelamente al piano di proiezione, ossia se $\mathbf{n} \cdot \mathbf{u} \neq 0$*, allora la proiezione della prospettiva centrale generale del Teorema 16.5.4 manda questo fascio di rette in un insieme di rette* $\mathbf{s}$ *radiali rispetto al (ossia che si incontrano nel) punto di fuga*

$$\mathbf{f_u} = \mathbf{c} + \frac{d}{\mathbf{n} \cdot \mathbf{u}} \mathbf{u}\,.$$

Altrimenti, se $\mathbf{n} \cdot \mathbf{u} = 0$*, il fascio di rette rimane parallelo anche dopo la trasformazione prospettica, e non c'è alcun punto di incontro al finito (per queste direzioni il punto di fuga corrisponde al punto proiettivo all'infinito dato dalla direzione del versore* $\mathbf{u}$*, ossia dalla clase di equivalenza di* $(u_x, u_y, u_z, 0)$*: diciamo che* il punto di fuga è all'infinito*).*

*Dimostrazione*Scriviamo $\mathbf{r}(t) = (x(t), y(t), z(t))$, e la retta immagine $\mathbf{s}(t) = (x'(t), y'(t), z'(t))$. Sia T la trasformazione della prospettiva centrale con piano di proiezione P e centro di proiezione $\mathbf{c}$: essa agisce in coordinate omogenee tramite la matrice $M_\mathbf{c}$ del Teorema 16.5.4. Scriviamo rispettivamente $\mathbf{R}(t)$, $\mathbf{S}(t)$ le classi di equivalenza proiettiva $[\mathbf{r}(t)]$, $[\mathbf{s}(t)]$, ossia in coordinate omogenee. Allora la trasformazione $T\mathbf{r} = \mathbf{s}$ diventa $M_\mathbf{c}\mathbf{R}(t) = \mathbf{S}(t)$. Scegliamo i rappresentanti speciali della classe di equivalenza omogenea $\mathbf{R}(t)$, ovvero quelli del tipo $(x(t), y(t), z(t), 1)$. L'azione di $M_\mathbf{c}$ manda questo vettore in un rappresentante di $\mathbf{S}$ che scriviamo $(x''(t), y''(t), z''(t), g(t))$. Dal Teorema 16.5.4 ora abbiamo

$$\begin{aligned}
x''(t) &= (d + c_x n_x)(r_x^0 + tu_x) + c_x n_y (r_y^0 + tu_y) + c_x n_z (r_z^0 + tu_z) - c_x d_0 \\
y''(t) &= c_y n_x (r_x^0 + tu_x) + (d + c_y n_y)(r_y^0 + tu_y) + c_y n_z (r_z^0 + tu_z) - c_y d_0 \\
z''(t) &= c_z n_x (r_x^0 + tu_x) + c_z n_y (r_y^0 + tu_y) + (d + c_z n_z)(r_z^0 + tu_z) - c_z d_0 \\
g(t) &= n_x (r_x^0 + tu_x) + n_y (r_y^0 + tu_y) + n_z (r_z^0 + tu_z) - d_1\,.
\end{aligned}$$

Oserviamo che l'ultima identità si traduce in

$$g(t) = \mathbf{n} \cdot \mathbf{r^0} + \mathbf{tn} \cdot \mathbf{u} - \mathbf{d_1} = \mathbf{tn} \cdot \mathbf{u}$$

perché $\mathbf{n} \cdot \mathbf{r^0} = \mathbf{d_1}$ dal momento che abbiamo scelto $\mathbf{r^0}$ in P e d_1 è la distanza di P dall'origine in base alla Nota 16.5.3. Da qui passiamo ai valori tridimensionali delle componenti di $\mathbf{s}(t)$ dividendo x'', y'', z'' per g e poi facciamo tendere t ad infinito per trovare il punto di fuga $\mathbf{f_u}$. Il limite esiste; esso dà luogo alla classe di equivalenza proiettiva di un punto al finito se il limite del denominatore della divisione prospettica (ossia il limite della quarta coordinata $g(t)$) non si annulla. In tal caso

(estraendo come al solito le prime tre coordinate del suo rappresentante speciale) il limite corrisponde al punto $\mathbf{f_u}$ dell'enunciato, ed esso giace in P (perché $\frac{d}{\mathbf{n}\cdot\mathbf{u}}$ è la lunghezza del segmento che parte da $\mathbf{c}$ in direzione $\mathbf{u}$ e termina al punto di intersezione con il piano P la cui distanza da $\mathbf{c}$ vale d). Per maggior chiarezza, quest'ultimo argomento viene ripetuto più in dettaglio alla fine della prossima Nota 16.6.2.
Se invece $\mathbf{n}\cdot\mathbf{u}=0$ il termine in t viene moltiplicato per zero e scompare, quindi le rette trasformate non si incontrano per alcun valore di t. In tal caso la quarta coordinata omogenea $g(t)$ vale 0 per tutti i t, e $\mathbf{f_u}$ corrisponde al punto proiettivo all'infinito $\mathbf{u}$, 0). □

NOTA 16.6.2. La determinazione del punto di fuga di un fascio di rette in direzione $\mathbf{u}$, ricavata nel Teorema 16.6.1, si può ottenere senza calcoli con un argomento geometrico alternativo ed illuminante, che ora presentiamo.
Per prima cosa si osserva che il punto di fuga deve esistere al finito se il fascio non è parallelo al piano di proiezione, perchè in tal caso la proiezione centrale diminuisce le distanze di punti che si muovono parallelamente allontanandosi dal piano di proiezione, ossia restando a distanza costante prima della proiezione (questo fatto geometricamente evidente equivale alla divisione prospettica per la distanza).
In secondo luogo, l'immagine della proiezione prospettica centrale è il piano di proiezione P. In particolare, il punto di fuga del fascio di rette parallele a $\mathbf{u}$ è un punto di P. In questo fascio basta allora considerare la retta che passa per il centro di proiezione $\mathbf{c}$, ossia $\mathbf{r}(t)=\mathbf{c}+t\mathbf{u}$: il punto di fuga del fascio è l'intersezione di questa retta con P (esso esiste se e solo se $\mathbf{u}$ non è un vettore parallelo a P: se invece lo è, allora le rette del fascio sono tutte parallele al piano P, e poiché $\mathbf{c}$ si sceglie fuori di questo piano la retta $\mathbf{s}$ non lo interseca: in tal caso si dice che il punto di fuga è all'infinito, come abbiamo visto nella dimostrazione del precedente Teorema 16.6.1 e vedremo geometricamente nella successiva Nota 16.6.3).
Se il fascio non è parallelo a P, mostriamo in dettaglio ciò che abbiamo già accennato alla fine della dimostrazione del precedente Teorema 16.6.2, ossia che il punto $\mathbf{f_u}$ giace in P. Sia come sempre d la distanza fra $\mathbf{c}$ e P. Poiché $\mathbf{u}$ ha norma 1, il valore di t per il quale la retta $\mathbf{s}(t)=\mathbf{c}+t\mathbf{u}$ interseca P è la lunghezza $d_\mathbf{u}$ del segmento da $\mathbf{c}$ a P nella direzione di $\mathbf{u}$. La lunghezza della proiezione di questa segmento sulla direzione normale a P (quella del versore $\mathbf{n}$) è quindi d: pertanto $d_\mathbf{u}=d_1/\mathbf{n}\cdot\mathbf{u}=d_1/\cos\theta$, dove θ è l'angolo sotteso da $\mathbf{u}$ e $\mathbf{n}$. In altre parole, il punto di fuga è precisamente $\mathbf{f_u}=\mathbf{c}+\mathbf{u}\,d/\cos\theta$, come trovato con i calcoli della dimostrazione del Teorema 16.6.1, e giace in P. □

NOTA 16.6.3. È interessante vedere come cambia la posizione del punto di fuga al variare del versore $\mathbf{u}$ del fascio nella sfera unitaria. Se $\mathbf{u}$ varia nella sfera mantenendo costante la propria deviazione angolare θ da $\mathbf{n}$ (ossia se si muove sul parallelo C_θ rispetto al polo $\mathbf{n}$), allora il punto di fuga descrive anch'esso una circonferenza nello spazio, ed esattamente il parallelo C_θ della sfera con centro $\mathbf{c}$ e raggio $d/\cos\theta$, che tende ad infinito quando θ tende a $\pi/2$. Questo è vero fin tanto che $\theta\neq\pi/2$, ossia purché $\mathbf{u}$ non sia perpendicolare a $\mathbf{n}$, ovvero sull'equatore. Il caso di $\mathbf{u}$ sull'equatore è il caso limite in cui il raggio di queste circonferenze diverge, e diciamo allora che i punti di fuga sono all'infinito: in tal caso le rette rimangono parallele anche dopo la trasformazione prospettica.

Quindi punti di fuga a distanza piccola dal centro di proiezione si ottengono quando il fascio si avvicina ad essere perpendicolare al piano di proiezione (ossia l'angolo θ vicino a zero, $\mathbf{u}$ vicino ad essere allineato con $\mathbf{n}$), ed in aggiunta quando d è piccolo. Per quanto visto nella Nota 16.5.3, l'ultima condizione significa che la distanza fra il centro di proiezione (ovvero l'osservatore) ed il piano di proiezione P è piccola, ossia la prospettiva è accentuata. In altre parole, punti di fuga ravvicinati corrispondono a un osservatore vicino alla scena, con l'accentuazione (o distorsione) prospettica tipica di ciò che si ottiene quando si fotografa una scena da vicino. In questo caso, perché si riesca a

riprendere la scena nella sua interezza, bisogna usare un grandangolo, e questa distorsione prospettica si chiama appunto *distorsione grandangolare.* □

Dal Teorema 16.6.1 ottieniamo immediatamente il risultato seguente. Si noti che la terza identità è coerente con quella trovata nella Nota 16.2.2.

COROLLARIO 16.6.4. *Scriviamo* $\mathbf{f}_i$ *invece che* $\mathbf{f}_{\mathbf{e}_i}$ *(i=1,2,3) per i* punti di fuga principali *della prospettiva centrale, ossia i punti di fuga dei fasci di rette paralleli, rispettivamente, ai tre versori canonici di base. Essi sono, rispettivamemte,*

$$\mathbf{f_1} = (c_x + \frac{d}{n_x}, c_y, c_z) \tag{16.6.1a}$$

$$\mathbf{f_2} = (c_x, c_y + \frac{d}{n_y}, c_z) \tag{16.6.1b}$$

$$\mathbf{f_3} = (c_x, c_y, c_z + \frac{d}{n_z}) \,. \tag{16.6.1c}$$

In queste identità, si noti che, per ogni coppia di versori canonici, i corrispondenti punti di fuga principali hanno componente lungo il restante versore uguale a quella del centro di proiezione: ad esempio, i punti di fuga principali nelle direzioni degli assi x e y hanno la stessa altezza sul piano $\{z = 0\}$ *del centro di proiezione, e così via ruotando gli indici.*

ESERCIZIO 16.6.5. Mostrare che i punti di fuga principali, se sono punti al finito, sono esattamente le intersezioni con il piano di proiezione delle tre semirette uscenti dal centro di proiezione $\mathbf{c}$ nelle direzioni dei tre versori canonici di base. □

ESERCIZIO 16.6.6. (*i*) Un pittore sceglie la prospettiva per un quadro nel quale, oltre ad altri dettagli, dipinge due edifici a forma di parallelepipedo. Il primo edificio è disposto in modo frontale, diciamo come il cubo di lato 1 nel primo ottante che contiene $(0, 0, 3)$ ed ha la faccia più vicina al piano di proiezione parallela ad esso e disposta sul piano $\{z = 3\}$, e le altre due facce che contengono $(0, 0, 3)$ disposte rispettivamente sui piani $\{x = 0\}$ e $\{y = 0\}$. Per dipingere questo edificio il pittore sceglie la distanza d del proprio punto di osservazione dal piano della tela (supponiamo che il piano sia $\{z = 2\}$ e sia $d = 1$), ed un punto di fuga per i lati del cubo diretti parallelamente all'asse z (ossia ortogonalmente alla tela). Supponiamo che come punto di fuga scelga $\mathbf{f_3} = (1, 1, 2)$.
L'altro edificio è anch'esso a forma cubica, con angolo di $\pi/4$ (45 gradi) rispetto alla direzione di osservazione: ad esempio il cubo di altezza $\sqrt{2}$ sopra il quadrato sul piano $\{y = 0\}$ di vertici $(x = 4\,,\,,z = 3)$, $(5\,,\,4)$, $(4\,,\,5)$, $(3\,,\,4)$. Il pittore come disegna in maniera prospetticamente corretta i lati di questo secondo edificio?

(*ii*) La stessa scena dipinta dal pittore viene fotografata con una macchina fotografica disposta dove era il pittore, con la stessa angolazione (quindi frontalmente rispetto al primo edificio, ossia con le stesse angolazioni orizzontale e verticale). Quella che era la distanza del pittore dalla tela ora diventa la lunghezza focale dell'obiettivo, che, quando guardiamo la fotografia, non ci è nota. Come possiamo ricostruire dalla fotografia l'angolazione nello spazio tridimensionale del secondo edificio?

Svolgimento. (*i*). Sappiamo che $\mathbf{n} = (0, 0, 1)$, quindi $n_z = 1$. Dall'equazione (16.6.1c) del punto di fuga principale $\mathbf{f_3}$ ricaviamo $c_z + d = 2$, quindi $c_z = 1$, ed anche le altre coordinate del centro di proiezione valgono 1. In altre parole $\mathbf{c} = (1, 1, 1)$.

Le due facciate visibili del secondo edificio sono dirette come $\mathbf{u}^- = (-1/\sqrt{2}, 0, -1/\sqrt{2})$ e $\mathbf{u}^+ = (1/\sqrt{2}, 0, -1/\sqrt{2})$. I punti di fuga per questi versori si ottengono dal Teorema 16.6.1:

$$\mathbf{f}_- = (1,1,1) - \sqrt{2}(-1/\sqrt{2}, 0, -1/\sqrt{2}) = (2,1,2)$$
$$\mathbf{f}_+ = (1,1,1) - \sqrt{2}(1/\sqrt{2}, 0, -1/\sqrt{2}) = (0,1,2)\,.$$

Il pittore disegna quindi i bordi del secondo edificio facendo convergere le linee parallele ai rispettivi punti di fuga appena trovati.
(*ii*). Dalla fotografia misuriamo le coordinate del punto di fuga dei lati del primo edificio ortogonali al piano della pellicola, che abbiamo chiamato $\mathbf{f_3}$. Grazie all'equazione (16.6.1c) questa misura ci dà il vettore $\mathbf{c} + d\mathbf{e_3}$, ossia le prime due coordinate di $\mathbf{c}$ ed il valore della quantità $c_z + d$, che è la terza componente di $\mathbf{f_3}$ e vale 2. Ora consideriamo i punti di fuga associati agli spigoli non verticali del secondo edificio (quelli verticali rimangono paralleli al piano della pellicola, perché l'edificio e questo piano sono entrambi verticali, quindi paralleli: il terzo punto di fuga è all'infinito). Vogliamo determinare i versori $\mathbf{u_\pm}$ delle due famiglie ortogonali di tali spigoli. Osserviamo che $\mathbf{n} = \mathbf{e_3}$, e pertanto $\mathbf{u}^\pm \cdot \mathbf{n} = u_z^\pm$. Ora dalle equazioni (16.6.1a) e (16.6.1c) ricaviamo i vettori $\mathbf{c} + \frac{d}{u_z^\pm}\mathbf{u}^\pm$. Poiché le prime due componenti di $\mathbf{c}$ sono già note, questo ci restituisce i valori di $du_x^\pm/u_z^\pm$ e $du_y^\pm/u_z^\pm$ (ma attenzione: sappiamo già che $u_y^\pm = 0$, perché l'edificio è verticale, non inclinato, quindi l'ultima informazione non ci serve). Per le terze componenti, troviamo in entrambi i casi il valore di $c_z + d$, ma sapevamo già che questo valore è 2, quindi anche questa informazione non ci dà nulla di nuovo. Infine sappiamo che $\mathbf{u}^+$ e $\mathbf{u}^-$ sono ortogonali e di norma 1. Ricapitolando tutte queste informazioni nell'ordine dato, abbiamo:

$$c_z + d = 2$$
$$d\,\frac{u_x^+}{u_z^+} = \text{costante nota}$$
$$d\,\frac{u_x^-}{u_z^-} = \text{costante nota}$$
$$u_x^+ u_x^- + u_z^+ u_z^- = 0$$
$$(u_x^+)^2 + (u_z^+)^2 = 1$$
$$(u_x^-)^2 + (u_z^-)^2 = 1\,.$$

Si tratta di sei equazioni quadratiche nelle sei incognite d, c_z, $u_x^\pm$, $u_z^\pm$ (si rammenti che $u_y^\pm = 0$). Se il sistema è risolvibile, in tal modo si determina d ed i due versori $\mathbf{u}^\pm$ richiesti. Lasciamo al lettore considerare le condizioni di risolubilità e sviluppare i calcoli numerici dopo aver fissato a proprio piacere le posizioni dei tre punti di fuga utilizzati (ma ci si rammenti che questi tre punti debbono essere scelti tutti alla stessa altezza y, perchè giacciono sulla retta che rappresenta l'orizzonte sul piano della pellicola, e questa retta è orizzontale perché la macchina fotografica non è inclinata di lato).

Si noti anche che ora conosciamo anche $\mathbf{c}$ e d, e quindi l'intera matrice della trasformazione prospettica. Il lettore è invitato a scriverla esplicitamente. □

Esempio 16.6.7. La prospettiva centrale con un solo punto di fuga è quella in cui il piano di proiezione è parallelo a due dei versori canonici, ossia perpendicolare al terzo. Ad esempio, se come d'abitudine in Computer Graphics la direzione perpendicolare è quella dell'asse z, si ottiene la prospettiva del tipo della proiezione standard della Sezione 16.2. Abbiamo già determinato nell'Esempio 16.2.3 come appare il cubo unitario in questa prospettiva: lo disegniamo in Figura 5.

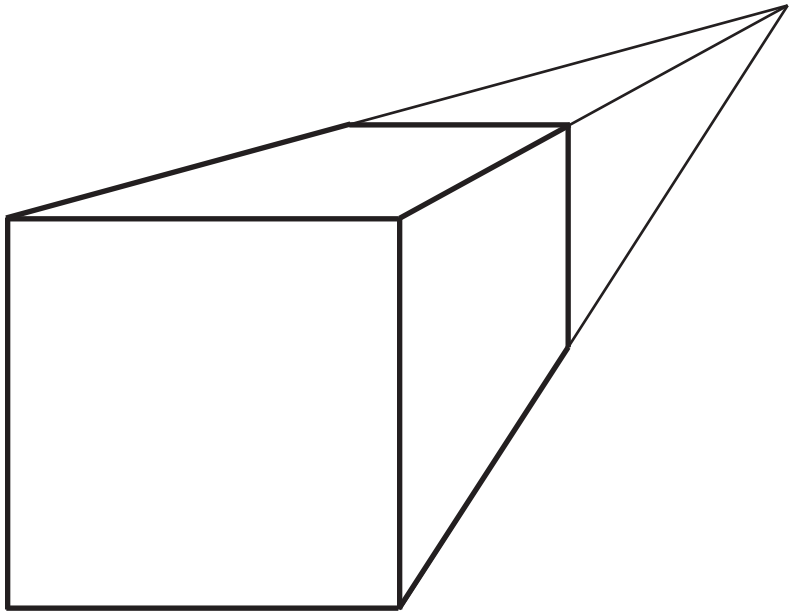


FIGURA 16.6.1. Prospettiva centrale con un solo punto di punto di fuga al finito.

Il caso di due punti di fuga è illustrato nella Figura 6. La retta a cui appartengono i due punti di fuga si chiama l'*orizzonte*. Si osservi che, in pittura, è il pittore a decidere a che altezza collocare la retta dell'orizzonte, esattamente come, nel caso di un solo punto di fuga, l'altezza di tale punto nel disegno può essere scelta arbitrariamente. In effetti, la precedente Figura 5 illustra il caso di orizzonte a media altezza (e centro di prospettiva alto, ovvero punto di osservazione dall'alto: la faccia superiore del cubo è visibile, il centro di prospettiva (ossia l'osservatore) è più alto del cubo). In Figura 6 disegniamo il cubo con orizzonte e centro alti, in Figura 7 lo ridisegniamo con orizzonte e centro più bassi (la proiezione sul piano di visuale dell'orizzonte è la linea tratteggiata). Se abbassassimo ancora di più il centro di prospettiva, vedremmo anche la faccia inferiore del cubo, dal di sotto.

□

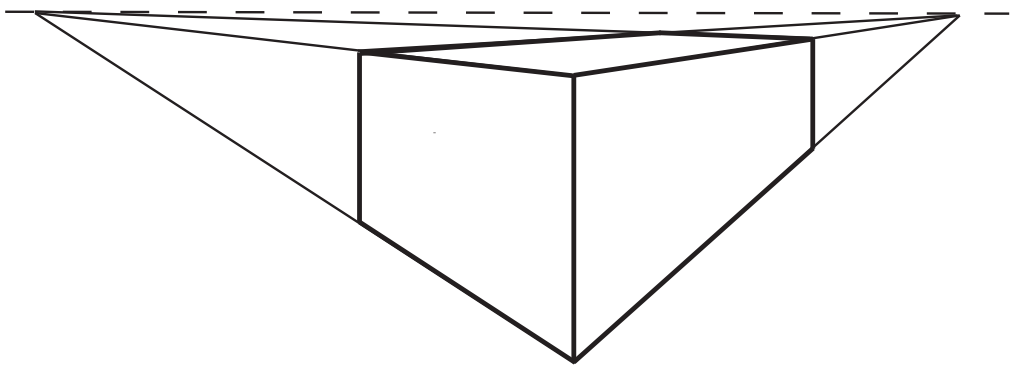


FIGURA 16.6.2. Prospettiva centrale con due punti di fuga, centro di proiezione alto, e quindi orizzonte alto.

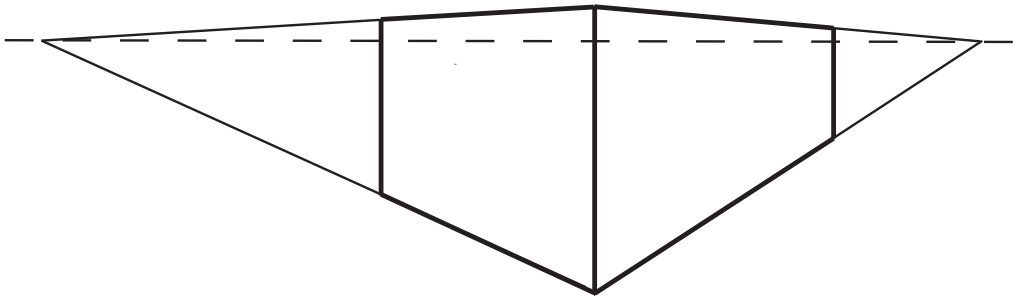


FIGURA 16.6.3. Prospettiva centrale con due punti di fuga e centro di proiezione ad altezza media, e quindi orizzonte ad altezza media.

CAPITOLO 17

Appendice: spazi di colore

17.1. Distribuzione spettrale della luce e lunghezza d'onda dominante

La luce visibile consiste di onde elettromagnetiche di lunghezza d'onda compresa fra $400nm$ (violetto) e $700nm$ (rosso).
Quando percepiamo un colore, la luce che vediamo non è di solito monocromatica, bensì ha una distribuzione di energia al variare della lunghezza d'onda.
Questa distribuzione viene chiamata *distribuzione spettrale* ed indicata con $P(\lambda)$.

FIGURA 17.1.1. Distribuzione spettrale della luce monocromatica

Nella Figura 17.1.1, $P(\lambda)$ è una distribuzione, la delta di Dirac, a $\lambda = \lambda_0$: vediamo luce monocromatica a lunghezza d'onda λ_0.

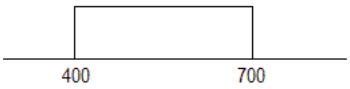


FIGURA 17.1.2. Distribuzione spettrale della luce bianca o grigia

Nella Figura 17.1.2,

$$P(\lambda) = \begin{cases} 1 & \text{se } 400 \text{ nm } \leqslant \lambda \leqslant 700 \text{ nm}, \\ 0 & \text{altrimenti.} \end{cases}$$

Qui 1=100% sta simbolicamente per la luminosità massima, su un monitor con 255 livelli di grigio avremmo forse scelto 255. Vediamo luce bianca (o grigia, a seconda dell'intensità). La luce bianca (o grigia), cioè senza dominanti cromatiche, ha una distribuzione spettrale equidistribuita fra le varie frequenze.

Se sommiamo le distribuzioni in questi due esempi, come nella Figura 17.1.3, otteniamo luce colorata (lunghezza d'onda λ_0) con una componente addizionale di grigio: cioè vediamo il colore corrispondente alla lunghezza d'onda λ_0 ma desaturato.
Si dice che λ_0 è la lunghezza d'onda dominante.

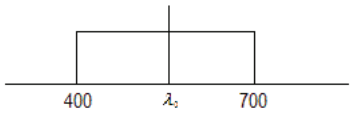


FIGURA 17.1.3. Distribuzione spettrale di un colore dominante desaturato

17.2. Lo stesso colore può essere ottenuto da distribuzioni spettrali differenti

DEFINIZIONE 17.2.1. Due colori la cui somma è il bianco si dicono *complementari*.

Sia $P_w(\lambda)$ la distribuzione spettrale della luce bianca che scegliamo come bianco standard. Ad esempio, supponiamo che $P_w(\lambda)$ sia come alla figura 17.1.2 (di solito però la distribuzione spettrale del bianco è piatta, perché dipende anche dalla curva di risposta del nostro sistema visivo: si veda la Figura 17.5.1) nel seguito.
Ora sia $P(\lambda)$ la distribuzione spettrale di un altro colore. Allora il colore complementare a $P(\lambda)$ ha distribuzione $Q(\lambda) = P_w(\lambda) - P(\lambda)$.

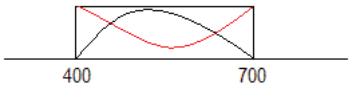


FIGURA 17.2.1. Colori complementari

Se consideriamo una distribuzione spettrale che approssima quelle della Figura 17.1.3, ma senza componenti discrete monocromatiche, percepiamo lo stesso colore della distribuzione in Figura 17.1.3.

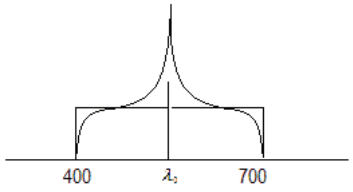


FIGURA 17.2.2. Distribuzione spettrale con una componente discreta monocromatica

Cautela: la lunghezza d'onda dominante potrebbe non coincidere esattamente col punto di massimo di $P(\lambda)$. Si veda il grafico in Figura 17.2.3.
La lunghezza d'onda dominante è λ_0, i punti di massimo di $P(\lambda)$: λ_+, λ_-. Pertanto, differenti distribuzioni spettrali $P(\lambda)$, possono corrispondere alla percezione dello stesso colore.
Tutti i colori (oppure i loro complementari: si veda la sezione 17.11 nel seguito) si possono classificare in termini di tre parametri:

(1) croma (o tinta): la lunghezza d'onda dominante
(2) luminosità (l'intensità del colore)
(3) saturazione o purezza (una bassa saturazione significa che diluizione con componenti di grigio è elevata).

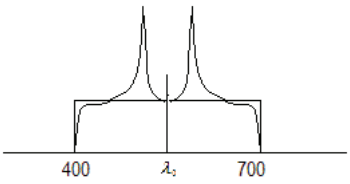


FIGURA 17.2.3. Distribuzione spettrale con più componenti discrete monocromatiche

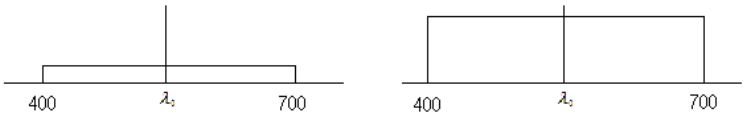


FIGURA 17.2.4. Distribuzione spettrale per la stessa tinta a due diversi livelli di saturazione

17.3. Croma, Saturazione e Luminosità: tre parametri per classificare i colori

Il croma è il colore monocromatico (o il suo complementare) alla massima saturazione e luminosità. Può essere messo in corrispondenza con un angolo fra 0° e 360° (scala del croma). Si ottiene quindi il seguente diagramma croma-saturazione.

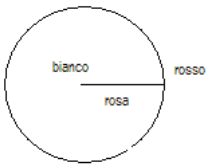


FIGURA 17.3.1. Diagramma croma-saturazione

La circonferenza esterna corrisponde alla massima saturazione. Per saturazione uguale a zero si ottiene al centro il grigio (il bianco se siamo alla massima intensità). Facendo variare anche l'intensità si ottiene un cono.

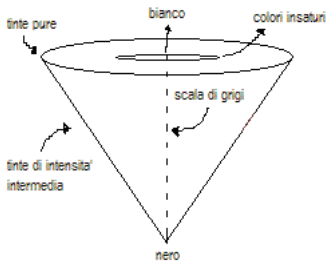


FIGURA 17.3.2. Diagramma croma-saturazione-intensità

17.4. Percezione del colore: teoria del tristimolo

Una teoria della biofisica della percezione ottica, confermata sperimentalmente, asserisce che la retina dispone di tre tipi di sensori, detti coni, con sensibilità centrata sulle bande di lunghezza d'onda corrispondenti rispettivamente al blu al verde e al rosso. Questa teoria è comoda per la modellazione del colore per emissione (come quello dei fosfori di un monitor) nel quale il colore si parametrizza con le intensità nei tre canali primari rosso, verde e blu (RGB).

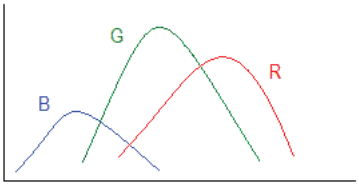


FIGURA 17.4.1. Curve di sensibilità ai colori primari RGB

Si noti come i sensori "blu" assorbano meno luce: siamo meno sensibili alla luce blu, e più in generale alla banda a bassa lunghezza d'onda (alta frequenza) dal violetto al blu.

17.5. Curva di sensibilità al colore

Dalle curve di sensibilità ai colori RGB ricaviamo la curva di sensibilità globale del nostro sistema visivo al colore, cioè la sua distribuzione di sensibilità relativa al variare della lunghezza d'onda; il massimo della sensibilità relativa si ha in corrispondenza di $\lambda = 550nm$ (giallo-verde). La curva della sensibilità relativa nella Figura 5.1 è la somma dei tre grafici nella Figura 17.4.1.

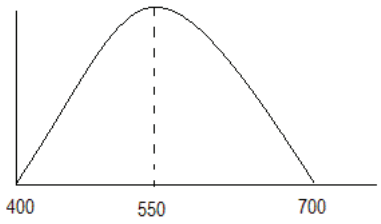


FIGURA 17.5.1. Curva di sensibilità al colore

17.6. Curve di corrispondenza al colore *(color matching curves)*

Segue dalla teoria del tristimolo che il sistema visivo percepisce ogni colore come una distribuzione lineare a coefficienti (cioè intensità) positivi delle componenti primarie rosso, verde e blu.
In realtà questa asserzione non è precisa. Per capire perché è necessario determinare sperimentalmente le curve di corrispondenza al colore.

DEFINIZIONE 17.6.1. Si chiamano curve di corrispondenza al colore i grafici delle tre funzioni $\bar{r}$, $\bar{g}$, $\bar{b}$ di λ il cui valore è l'intensità delle tre primarie rossa, verde e blu che corrisponde alla percezione del colore saturo monocromatico di lunghezza d'onda λ (o più in generale il colore saturo di lunghezza d'onda dominante λ) di luminosità costante massima al variare di λ.

Denotiamo queste curve con $\bar{r}_\lambda$, $\bar{g}_\lambda$, $\bar{b}_\lambda$. Esse sono illustrate nella figura 6.1. Questi grafici rappresentano le intensità percentuali di rosso, verde e blu tali che il nostro sistema visivo veda il colore di lunghezza d'onda λ alla massima purezza e ad intensità massima e costante.

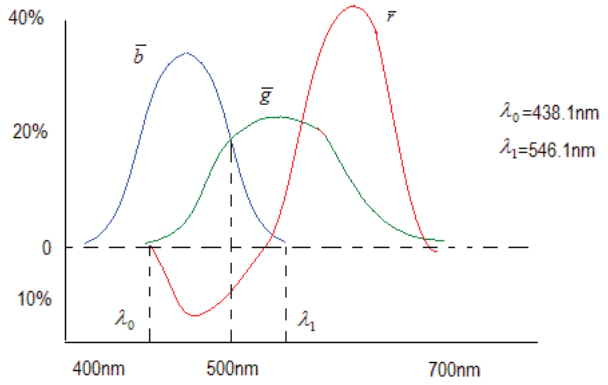


FIGURA 17.6.1. Le curve RGB di corrispondenza del colore *(color matching curves)*

Come si vede, la curva del rosso è al di sotto dell'asse delle ascisse nell'intervallo tra λ_0 e λ_1 . Questo significa che i colori in questa banda alla massima saturazione e luminosità non sono sintetizzabili sommando le componenti rosse, verdi e blu; però, se ad uno di questi colori si aggiunge una opportuna quantità di rosso (quella che compensa esattamente il valore negativo) allora il colore risultante è sintetizzabile (è in questo modo che sono stati misurati i valori negativi del grafico). Quindi non tutti i colori visibili sono riproducibili nel modello RGB. Questo fatto non dipende dalla qualità del monitor usato, ma dalla psicofisica del sistema visivo.

17.7. Risoluzione del colore da parte del nostro sistema visivo al variare della lunghezza d'onda

La capacità del sistema visivo di riconoscere colori della stessa luminosità e saturazione che differiscono solo per la tinta (cioè per la lunghezza d'onda) è stata determinata sperimentalmente su campioni di osservatori. Le differenze minime di lunghezza d'onda a cui il sistema visivo riconosce colori diversi cambiano con la lunghezza d'onda. La risoluzione è peggiore in prossimità del violetto e del rosso, come mostrato nella Figura **??**.

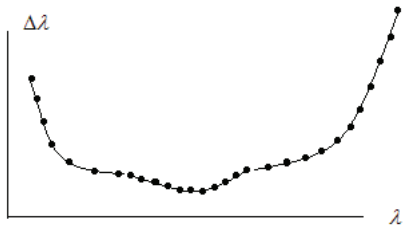


FIGURA 17.7.1. Risoluzione del sistema visivo umano al variare della lunghezza d'onda

Questo risultato sperimentale è in accordo con l'altro fatto sperimentale visto nell' Esempio 10, cioè che il sistema visivo è meno sensibile alla componente blu che alle altre. Al diminuire della

saturazione e della luminosità i colori si appiattiscono verso un colore costante (bianco, grigio o nero) e la risoluzione del sistema visivo, ovviamente, decresce.

17.8. Le curve di corrispondenza CIE (modello CIE-XYZ, detto anche CIE-LAB)

Un modello di colore nel quale si può ottenere ogni colore visibile sommando tre primarie è stato proposto nel 1931 dalla Commission Internazionale de l'Eclairage (CIE). Questo modello, che usa tre curve di corrispondenza che indichiamo con $\bar{x}$, $\bar{y}$, $\bar{z}$ risolvere il problema dei valori negativi della Figura 17.6.1. Il problema è dovuto alla incapacità del nostro sistema visivo a sintetizzare i colori alla massima luminosità e saturazione in certe bande di frequenza partendo da componenti R, G, B a causa della insufficiente curva di risposta dei suoi coni. Per evitare questo problema, la primaria CIE che misura la luminosità viene scelta come la sensibilità del sistema visivo alla luminosità del colore. Quindi la curva $\bar{y}$ è la stessa della Figura 17.5.1. Le altre curve sono definite in maniera appropriata (come differenze di opportuni canali di colori complementari, rispettivamente verde-magenta e blu-giallo, o, più precisamente delle corrispondenti color-matching curves del modello RGB o CMY). Si ottengono le curve seguenti:

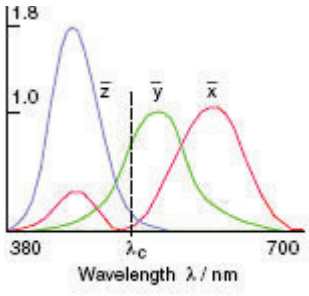


FIGURA 17.8.1. Le primarie CIE-XYZ

Si noti che $\bar{x}$ si annulla alla lunghezza d'onda per cui le color-matching curves blu e verde coincidono, cioè quella dove si annulla la curva $\bar{z}$. Per tale lunghezza d'onda, pari a circa 500 nm, la sensibilità del sistema visivo alla luminosità del colore è tutta basata sul canale relativo a $\bar{x}$ (cioè verde-magenta). Però, anche se $\bar{x}$ si annulla per $\lambda = 500nm$, le tre color-matching curves non diventano mai negative. Con queste primarie è quindi è possibile ricostruire qualsiasi colore. La componente nel canale di rappresenta la luminosità del colore ricostruito. Indichiamo con $\underline{X}$, $\underline{Y}$, $\underline{Z}$ i tre canali del modello CIE. Le curve di corrispondenza dell' Esempio 14 ci dicono in che percentuali $\bar{x}_\lambda$, $\bar{y}_\lambda$, $\bar{z}_\lambda$ si devono mescolare le intensità di questi canali per ricostruire colori che appaiano di luminosità costante.

17.9. Ricostruzione del colore nel modello CIE-XYZ

Consideriamo un colore di distribuzione spettrale $P(\lambda)$. Quali sono le sue coordinate X, Y, Z nel modello CIE? Dobbiamo considerare i prodotti scalari con le tre funzioni di ricostruzione $\bar{x}$, $\bar{y}$, $\bar{z}$

opportunamente normalizzati:

$$X = c\int_{-\infty}^{+\infty} P(\lambda)\bar{x}(\lambda)\,d\lambda$$
$$Y = c\int_{-\infty}^{+\infty} P(\lambda)\bar{y}(\lambda)\,d\lambda$$
$$Z = c\int_{-\infty}^{+\infty} P(\lambda)\bar{z}(\lambda)\,d\lambda$$

La costante di normalizzazione è scelta in modo da avere il valore convenzionale 100 al punto di bianco di una immagine stampata, cioè

$$c = \frac{1}{\int_{-\infty}^{+\infty} P_{bianco}(\lambda)\bar{y}(\lambda)\,d\lambda}$$

dove $P_{bianco}(\lambda)$ è la distribuzione spettrale del tipo di luce bianca che scegliamo come luce standard. Il prodotto scalare al denominatore è la luminosità complessiva percepita dal sistema visivo quando esso guarda questa luce bianca (massima intensità d grigio).
Per oggetti che emettono luce, come i monitor, non ha senso scegliere la costante di normalizzazione basandosi sulla luce bianca standard. Infatti la distribuzione spettrale del bianco su un monitor è regolabile, e dipende dal singolo monitor. Per convenzione in tal caso si sceglie di fissare $c = 600$ *lumieres/watt*.

Le coordinate X, Y, Z del colore $\mathbf{C}$ nel modello CIE sono i pesi con cui si ricostruisce (o meglio, si sintetizza) il colore $\mathbf{C}$ come combinazione lineare delle tre funzioni i cui grafici sono le curve di corrispondenza al colore dei tre canali $\underline{X}$, $\underline{Y}$, $\underline{Z}$:

$$\mathbf{C} = X\underline{X} + Y\underline{Y} + Z\underline{Z}$$

17.10. Coordinate di cromaticità

Queste coordinate si definiscono a partire dalle coordinate CIE X, Y, Z come i valori percentuali rispetto a $X+Y+Z$ (per chiarezza, diamo un nome alla quantità $X+Y+Z$: chiamiamola *contenuto cromatico*). In altre parole, poniamo:

$$x = \frac{X}{X+Y+Z} \tag{17.10.1}$$
$$y = \frac{Y}{X+Y+Z} \tag{17.10.2}$$
$$z = \frac{Z}{X+Y+Z} \tag{17.10.3}$$

Perciò $x+y+z = 1 = 100\%$ Nell'ottante $X+Y+Z \geqslant 0$ del modello di colore CIE le coordinate x, y, z corrispondono alla proiezione radiale sul piano $X+Y+Z = 1$.
Queste sono le coordinate dei colori di massima luminosità.
Poiché il sistema visivo vede solo i colori tra $\lambda = 400nm$, e $\lambda = 700nm$, e con una curva di risposta che si avvicina a zero a questi estremi, non tutto il triangolo sotteso dai vertici $(1,0,0)$, $(0,1,0)$, $(0,0,1)$ corrisponde ai colori visibili di massima luminosità.
La Figura 17.11 rappresenta lo spazio dei colori visibili:

Si tratta quindi di un cono contenuto nell'ottante positivo, con vertice nell'origine. La base del cono, cioè la faccia piatta, quella che non tocca l'origine, si chiama il *diagramma di cromaticità*.

È chiaro che per determinare un punto in questa faccia primaria bastano due parametri.
Decidiamo di scegliere come parametri x e y.

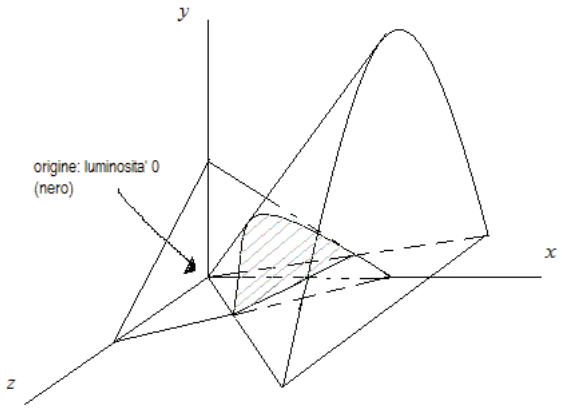


FIGURA 17.10.1. Spazio dei colori visibili

![h]

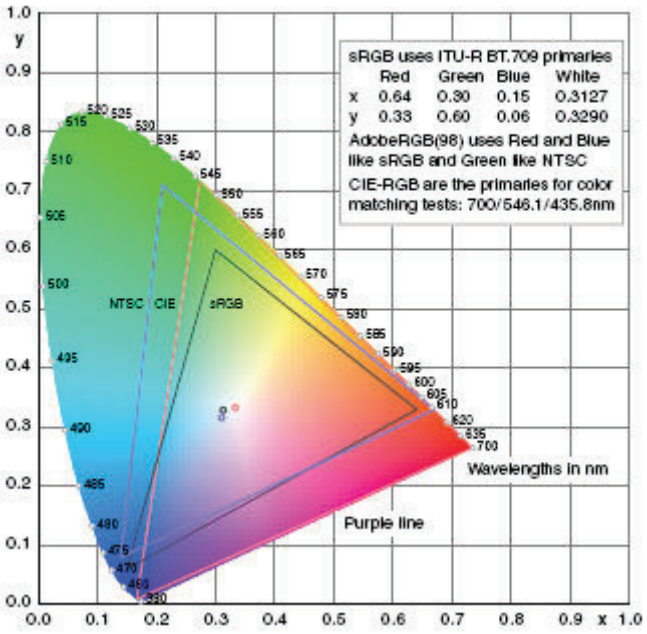


FIGURA 17.10.2. Diagramma di cromaticità

Infatti, poiché , $x + y + z = 1$ si ha $z = 1 - x - y$; pertanto possiamo ricostruire ogni colore $\mathbf{C} = X\underline{X} + Y\underline{Y} + Z\underline{Z}$, con questi due parametri.
In effetti, essi corrispondono alla versione di $\mathbf{C}$ di massimo contenuto cromatico $X + Y + Z = 1$. Per ricostruire $\mathbf{C}$ si specifica, ad esempio, anche la sua luminosità, cioè la coordinata Y. Così abbiamo costruito una corrispondenza

$$\mathbf{C} \to (X, Y, Z) \to (x, y, Y) \; ; \tag{17.10.4}$$

la regola di corrispondenza si ricava dall'equazione $x+y+z = 1 = 100\%$. Nell'ottante $X+Y+Z \geqslant 0$ del modello di colore CIE le coordinate x, y, z corrispondono alla proiezione radiale sul piano

$X + Y + Z = 1$ da cui si ottiene

$$\frac{x}{y} = \frac{X}{Y}$$
$$\frac{z}{y} = \frac{1 - x - y}{y} = \frac{Z}{Y}$$

Perciò

$$X = \frac{x}{y} Y$$
$$Y = y$$
$$Z = \frac{1 - x - x}{y} Y$$

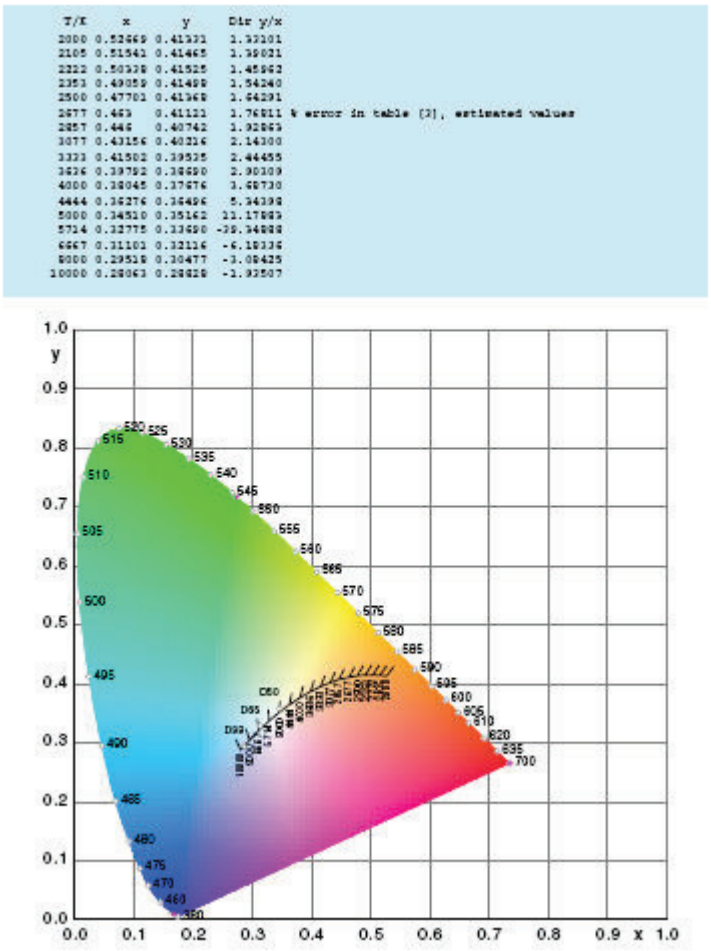
17.11. Diagramma di cromaticità

La parte curva del bordo del diagramma di cromaticità nella Figura corrisponde a tinte sature di massimo contenuto cromatico. Le loro distribuzioni spettrali sono percepite dal sistema visivo come colori monocromatici, che variano dal violetto al rosso.
All'interno del diagramma ci sono varianti meno sature degli stessi colori.
Un punto centrale corrisponde al bianco (saturazione zero, luminosità massima). Questo punto è scelto convenzionalmente come il colore corrispondente ad una fissata distribuzione spettrale vicina a quella della luce solare, ad esempio a mezzogiorno ($5600K$). Esso ha coordinate vicine a $x = \frac{1}{3}$, $y = \frac{1}{3}$, $z = \frac{1}{3}$. Indichiamo questo punto con w. Si possono scegliere punti di bianco differenti, ad esempio in corrispondenza a temperature di corpo nero diverse. Una temperatura di corpo nero più elevata si avvicina all'azzurro, una più bassa si avvicina al rosso. Spesso i monitor vengono calibrati perché il. loro punto di bianco corrisponda alla temperatura di $9300K$: nella Figura 17.11.1 riportiamo la traiettoria nel diagramma di cromaticità dei punti di bianco al variare della loro temperatura, ed indichiamo il corrispondente punto del diagramma di cromaticità con $D93$. Analogamente indichiamo il bianco corrispondente, ad esempio, alla temperatura di $4000K$ (un bianco arancio) con $D40$.
Il segmento da w ad un punto del bordo corrispondente al colore di lunghezza d'onda λ_0, percorre tutta la scala della saturazione di quel colore, dal bianco al colore puro (completamente saturo).
Il colore $\mathbf{C}$ di luminosità massima e di distribuzione spettrale $P(\lambda)$, corrisponde al punto interno al diagramma di cromaticità ottenuto dalla combinazione convessa (cioè l'integrale normalizzato) con peso $P(\lambda)$ dei punti del bordo del diagramma a lunghezza d'onda λ (quindi quelli corrispondenti alla parte curva del bordo).
A questo punto le coordinate x e y di $\mathbf{C}$ si calcolano come nella sezione precedente.
Questo identifica $\mathbf{C}$ (o meglio la sua proiezione a massimo contenuto cromatico) come un punto del diagramma di cromaticità.
Ad esempio nella Figura 17.11.2 percorrendo la corda da w a $\mathbf{C}$ si tocca il bordo in un punto che corrisponde alla lunghezza d'onda λ_0 . Questa lunghezza d'onda è la lunghezza d'onda dominante di $\mathbf{C}$.

Questo però è vero se la corda da w a $\mathbf{C}$ interseca il bordo nella parte curva. Se lo interseca nella parte piatta si ottengono i colori complementari, che ora definiamo.

17.12. Visualizzazione geometrica dei colori complementari nel diagramma CIE

Rivediamo la definizione di colore complementare alla luce della parametrizzazione con le coordinate x, y e z.
Un colore di massimo contenuto cromatico corrisponde ad un punto sul piano $X + Y + Z = 1$, X, Y, $Z \geqslant 0$.



T/K	x	y	Dir y/x
2000	0.52669	0.41331	1.32101
2105	0.51541	0.41465	1.35021
2222	0.50328	0.41525	1.45963
2353	0.49059	0.41499	1.54240
2500	0.47701	0.41369	1.64291
2677	0.463	0.41121	1.76911 % error in table [3], estimated values
2857	0.446	0.40742	1.92863
3077	0.43156	0.40216	2.14300
3333	0.41502	0.39535	2.44455
3636	0.39792	0.38690	2.90309
4000	0.38045	0.37676	3.48730
4444	0.36276	0.36496	5.34198
5000	0.34510	0.35162	11.17883
5714	0.32775	0.33690	-28.34888
6667	0.31101	0.32116	-6.18336
8000	0.29518	0.30477	-3.08425
10000	0.28063	0.28828	-1.92507

FIGURA 17.11.1. Diagramma di cromaticità e lunghezza d'onda dominante

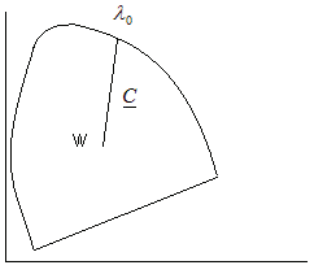


FIGURA 17.11.2. Diagramma di cromaticità e lunghezza d'onda dominante

Se sommiamo due colori $\mathbf{C}_1 = (x_1, y_1, z_1)$ e $\mathbf{C}_2 = (x_2, y_2, z_2)$ otteniamo il colore corrispondente alle coordinate $\mathbf{C}_1 + \mathbf{C}_2$.
Se i contenuti cromatici $X_1 + Y_1 + Z_1$ di $\mathbf{C}_1$ e $X_2 + Y_2 + Z_2$ di $\mathbf{C}_2$ sono bassi, si ottiene un colore di basso contenuto cromatico, ed il colore corrispondente di massimo conntenuto cromatico è

$\frac{1}{L}(X_1 + X_2, Y_1 + Y_2, Z_1 + Z_2$, dove $L = \|(X_1 + X_2, Y_1 + Y_2, Z_1 + Z_2)\|$.
In tal modo, quando succede che la somma dei due colori è il colore bianco?
Per linearità, sul piano di massimo contenuto cromatico $X + Y + Z = 1$, la somma di $\mathbf{C}_1$ e $\mathbf{C}_2$ è esattamente il punto di mezzo della corda che li congiunge nel diagramma di cromaticità.
Abbiamo quindi la seguente visualizzazione geometrica:

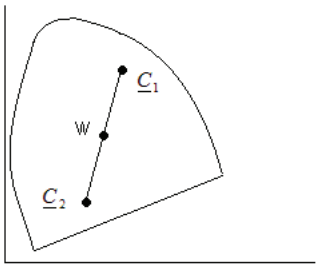


FIGURA 17.12.1. Diagramma di cromaticità e colori complementari

Il complementare di $\mathbf{C}_1$ è il punto opposto a $\mathbf{C}_1$ rispetto al punto di bianco w (alla stessa distanza da w ma dall'altro lato); w è il punto di mezzo della corda che congiunge $\mathbf{C}_1$ e $\mathbf{C}_2$.

Nel cono sotteso dal tratto piatto del bordo del diagramma di cromaticità e con centro in w si trovano i colori complementari dei colori dello spettro visibile (colori monocromatici). Si tratta della banda dal porpora al magenta.

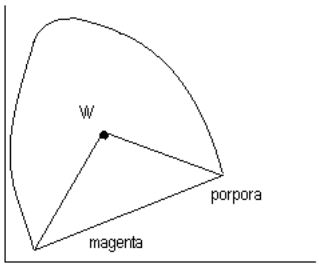


FIGURA 17.12.2. La banda dal porpora al magenta nel diagramma di cromaticità

17.13. Diagramma di cromaticità a luminosità

Il diagramma di cromaticità riguarda solo la sezione del cono di colori visibili che corrisponde al piano di massimo contenuto cromatico $X + Y + Z = 1$. Se fossimo interessati a un contenuto cromatico inferiore dovremmo scegliere piani di equazione $X + Y + Z = \ell$ con $0 \leqslant \ell \leqslant 1$.
I colori puri (quelli con valori massimi di X e Z) sono percepiti in maniera differente a seconda della luminosità Y. Ad esempio, la sovrapposizione di rosso e giallo, in dosi opportune, ad alta luminosità è un colore simile a quello della pelle di un individuo di razza bianca, mentre a bassa luminosità è il marrone.

17.14. Gamma dei colori riproducubili

Quali sono i colori riproducibili da un monitor?
Il monitor ha tre tipi di fosfori di emissione con distribuzione spettrale centrata rispettivamente sulle bande del rosso, del verde e del blu. Queste distribuzioni corrispondono a colori, quindi a coordinate di cromaticità x e y.
Ecco le coordinate corrispondenti ad un monitor tipico, che adotta lo spazio di colore sRGB (per ciascun monitor esse sono fornite dal costruttore, ma non si discostano mai molto da queste, a meno che il monitor non adotti un altro spazio di colore, magari con gamut più ampio, come Adobe RGB (98) - nella successiva Figura 17.14.1 confronteremo questi due gamuts).

	fosfori rossi	fosfori verdi	forfori blu
x	0.64	0.30	0.15
y	0.33	0.60	0.06

Di nuovo per linearità, la gamma di colori di massima grandezza cromatica riprodotta dal monitor (cioè nel piano $X + Y + Z = 1$) è il triangolo nel diagramma di cromaticità con vertice in questi tre punti.
Nota: poiché la gamma riprodotta dal monitor corrisponde ad un triangolo mentre i bordi del diagramma di cromaticità sono curvi, nessun monitor può visualizzare l'intera gamma di colori visibili. Lo stesso accade per le stampanti, la cui gamma riproducibile è il triangolo generato dai tre colori alle distribuzioni spettrali dei tre inchiostri ciano, magenta e giallo usati dalla stampante.
Se si usano più inchiostri questo triangolo diventa un poligono, ed approssima meglio l'intero diagramma di cromaticità, ma non può esaurirlo. Per le stampanti, i tre colori corrispondenti ai tre inchiostri sono percepiti in maniera diversa a seconda del tipo di luce bianca usata per generare le immagini e a seconda del tipo di carta per stamparle, quindi la gamma dipende da questi fattori.

FIGURA 17.14.1. La gamma dei colori riproducibili

17.15. Il modello LUV

A causa della differente risoluzione del colore da parte del sistema visivo alle diverse lunghezze d'onda, la percezione delle variazioni del colore in seguito ad uno spostamento dei parametri x e y del modello CIE a partire dal colore $\mathbf{C}_1$ per arrivare al colore $\mathbf{C}_2$ non dipende solo dalla lunghezza dello spostamento, cioè dalla distanza dei punti $\mathbf{C}_1$ e $\mathbf{C}_2$, ma anche dalla posizione di $\mathbf{C}_1$ e $\mathbf{C}_2$.
Per lo stesso motivo, se i parametri x e y variano in linea retta con velocità costante da (x_0, y_0) a (x_1, y_1), il sistema visivo può percepire una variazione di colore non uniforme.
Un modello di colore in cui la variazione sia percettivamente uniforme è stato sviluppato dalla Commissione Internazionale de L'Eclairage nel 1976.
Esso utilizza una variante non lineare, detta LUV, del modello XYZ. In altre parole, le coordinate LUV dipendono tramite espressioni non lineari (che includono potenze) da quelle XYZ. Qui ci limitiamo a presentare queste espressioni senza giustificazione.
e nuove coordinate sono chiamate ℓ, u, v e sono definite rispetto al punto di bianco.
Scriviamo x_w, y_w, z_w le coordinate CIE-XYZ del punto di bianco, e consideriamo due nuove quantità:

$$u' = \frac{X}{4X + 15Y + 3Z}$$
$$v' = \frac{Y}{X + 15Y + 3Z}$$

Ecco le espressioni corrispondenti per il punto di bianco:

$$u'_w = \frac{4X_w}{X_w + 15Y_w + 3Z_w}$$
$$v'_w = \frac{4Y_w}{X_w + 15Y_w + 3Z_w}$$

Allora ℓ^*, u^*, v^* sono date da:

$$\ell^* = 116\Big(\frac{Y}{Y_w}\Big)^{\frac{1}{3}} - 16$$
$$u^* = 13\ell^*(u' - u'_w)$$
$$v^* = 13\ell^*(v' - v'_w)$$

17.16. Uniformizzazione delle coordinate di colore di due monitor

Come visto nella Figura 17.14.1, la gamma di colore di un monitor dipende dalle coordinate di cromaticità dei suoi fosfori, che non sono le stesse per tutti i monitor. Questo significa che un'immagine che ai vari pixel ha determinati valori di luminosità dei tre canali RGB appare leggermente diversa su diversi monitor, a meno che non si applichi una trasformazione di coordinate per correggere le discrepanze. Questa trasformazione di coordinate nel software corrente di trattamento delle immagini (ad esempio nelle ultime versioni di Adobe Photoshop) viene effettuata nella fase di lettura dell'immagine; i dati necessari (ad esempio le coordinate di cromaticità dei fosfori) vengono inseriti una volta per tutte quando si sceglie il *profilo del monitor* in Photoshop.

La trasformazione di coordinate è basata sulla identificazione della gamma riproducibile dal monitor dentro lo spazio di colore CIE. Quindi dobbiamo scrivere la matrice J di trasformazione dalle coordinate R, G, B alle coordinate X, Y, Z:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = J \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{17.16.1}$$

Osserviamo che i coefficienti X_r, Y_r, Z_r sono le coordinate CIE del rosso più intenso riproducibile dal monitor. Infatti questo rosso di massima luminosità corrisponde alle coordinate $(R, G, B) = (1, 0, 0)$ e quindi alle coordinate X, Y, Z seguenti:

$$\begin{pmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} X_r \\ Y_r \\ Z_r \end{pmatrix} \tag{17.16.2}$$

Analogamente per gli altri coefficienti di J.

Una volta determinata J (lo faremo in seguito) per trasformare le coordinate R, G, B del primo monitor al secondo monitor basta usare le matrici rispettive J_1 e J_2 di conversione alle coordinate CIE e formare la matrice di "uniformizzazione" $J_2^{-1}J_1$.
Naturalmente, ogni colore $\mathbf{C}_1$ che è nella gamma del primo monitor, ma non in quella del secondo monitor, viene così trasformato in coordinate $\mathbf{C}_2 = J_2^{-1}J_1\mathbf{C}_1$ che sono al di fuori del cubo unitario del modello RGB, e quindi non riproducibili dal secondo monitor.
In tal caso approssimiamo con il punto ad esso più vicino del cubo unitario.

17.17. La trasformazione di coordinate per un monitor

Calcoliamo ora la matrice J di trasformazione da R, G, B al modello CIE per un dato monitor. Come visto nella (17.16.1) basta determinare le coordinate CIE dell'emissione dei suoi fosfori.
Per questo scriviamo E_r per il contenuto cromatico del rosso $E_r = X_r + Y_r + Z_r$, ed usiamo le relazioni (17.10.1), (17.10.2), (17.10.3) per passare alle coordinate di cromaticità:

$$x_r = \frac{X_r}{E_r} \tag{17.17.1}$$

$$y_r = \frac{Y_r}{E_r} \tag{17.17.2}$$

$$z_r = \frac{Z_r}{E_r} = \frac{1 - x_r - y_r}{E_r}\,. \tag{17.17.3}$$

Da qui si ha

$$\begin{aligned} X_r &= x_r E_r \\ Y_r &= y_r E_r \\ Z_r = z_r E_r &= (1 - x_r - y_r) E_r\,. \end{aligned}$$

Si definiscono analogamente E_g ed E_b .
Perciò la matrice J nella (17.16.1) diventa

$$J = \begin{pmatrix} x_r E_r & x_g E_g & x_b E_b \\ y_r E_r & y_g E_g & y_b E_b \\ (1 - x_r - y_r) E_r & (1 - x_g - y_g) E_g & (1 - x_b - y_b) E_b \end{pmatrix} \tag{17.17.4}$$

Le coordinate di cromaticità dei fosfori sono di solito specificate dal costruttore del monitor; altrimenti, esse si possono misurare con un colorimetro. Per trovare J basta quindi determinare E_r, E_g, E_b.
Per far questo si può procedere in uno dei due modi seguenti:

(1) Si misura con un fotometro di alta qualità la luminosità massima Y_r, Y_g, Y_b dell'emissione dei fosfori (cioè dei colori rosso, verde e blu più intensi emessi dal monitor; la qualità dello strumento deve essere sufficientemente alta da garantire buona precisione anche per il blu, colore per il quale la sensibilità è minore). Cos
'ı dalle (17.17.1), (17.17.2), (17.17.3) si ricava

$$E_r = \frac{Y_r}{y_r}$$
$$E_g = \frac{Y_g}{y_g}$$
$$E_b = \frac{Y_b}{y_b}\,.$$

Abbiamo già osservato che le cromaticità y_r, y_g, y_b dei fosfori sono fornite dal costruttore del monitor o misurabili: comunque sia possiamo assumerle note, e quindi dalle equazioni precedenti ricaviamo E_r, E_g, E_b.

(2) Si misurano le coordinate X_w, Y_w, Z_w del punto di bianco (cioè della luce bianca più intensa) del monitor. Esse corrispondono alle coordinate $(R, G, B) = (1, 1, 1)$. Perciò da (17.16.1) e da (17.17.4) si ottiene:

$$\begin{pmatrix} X_w \\ Y_w \\ Z_w \end{pmatrix} = \begin{pmatrix} x_r & x_g & x_b \\ y_r & y_g & y_b \\ 1-x_r-y_r & 1-x_g-y_g & 1-x_b-y_b \end{pmatrix} \begin{pmatrix} E_r \\ E_g \\ E_b \end{pmatrix} \tag{17.17.5}$$

In questo sistema le incognite sono E_r, E_g, E_b mentre X_w, Y_w, Z_w sono costanti note. Risolvendo il sistema si trovano E_r, E_g, E_b.
Il secondo metodo è più preciso, perché misurare le coordinate X_w, Y_w, Z_w del punto di bianco è più preciso che misurare le coordinate di cromaticità dei colori puri rosso, verde e blu (come già osservato, ci vuole un fotometro di alta qualità per misurare con precisione le coordinate del blu, dove la sensibilità è minore).

CAPITOLO 18

Appendice: esempio di Path Tracer semplificato di Illuminazione Globale nel linguaggio C++

Nei capitoli 11, 12, 13 e 14 abbiamo presentato un codice completo di Illuminazione Globale, scritto per semplicità usando un sottoinsieme del linguaggio C++ sostanzialmente identico al linguaggio C. Come annunciato all'inizio del Capitolo 11, ora presentiamo un codice simile in C++, più semplice dal punto di vista algoritmico perché usa un minor numero di procedure di Illuminazione Globale, però scritto usando molte più strutture tipiche del linguaggio C++ (e quindi più adatto ad essere ulteriormente sviluppato al fine di dotarlo di una interfaccia utente agile).

Il codice è stato elaborato e presentato da Marco Petreri **[38]**.

---

world.cpp:

```
/*
  File di definizione della classe World, la classe principalmente coinvolta nel
      rendering della scena.
  Ha il compito di gestire e lanciare i processi basilari del calcolo
      dell'immagine, come la creazione della scena,
  la ricerca delle intersezioni e il calcolo della radianza nel loop sui campioni
      e sui pixel.
*/

#include "world.hpp"
#include "geometry/ray.hpp"
#include "accelerators/accelerator.hpp"
#include "samplers/sampler.hpp"
#include "integrators/integrator.hpp"
#include "cameras/camera.hpp"
#include "geometricObjects/objectGroup.hpp"
#include "lights/light.hpp"
#include "materials/material.hpp"
#include "utilities/constants.hpp"
#include "utilities/math.hpp"
#include "utilities/stats.hpp"
#include "utilities/parser.hpp"
#include "utilities/film.hpp"
#include "utilities/environment.hpp"

using namespace std;

int px, py;
```

```
World::World(): stats(new Stats(this)), parser(new Parser(this)){}

World::~World(){
  delete env;
  delete cam;
  delete film;
  delete turbo;
  delete stats;
  delete parser;
  delete sampler;
  delete integrator;
  for(auto & x : lights)
  delete x;
  for(auto & x : models)
  delete x;
  for(auto it = materials.begin(); it != materials.end(); ++it)
  delete it->second;
}

void World::addLight(Light * l){ // aggiunge una luce all'array delle luci
  lights.push_back(l);
}

void World::addModel(ObjectGroup * s){ // aggiunge un ObjectGroup all'array degli
    ObjectGroup
  models.push_back(s);
}

void World::addMaterial(const std::string & name, Material * m){ // aggiunge un
    materiale alla mappa dei materiali
  materials[name] = m;
}

void World::buildScene(const std::string & scene){ // costruisce la scena da
    renderizzare
  cout << "< Building Scene >" << endl;

  parser->parseScene(scene); // lancia il parsing della scena passando alla
      funzione il percorso del file xml

  turbo->init(); // una volta caricata la geometria della scena viene lanciata
      l'inizializzazione dell'acceleratore

  cout << "</ Scene Builded >" << endl;
}
```

```
Intersection World::intersectTree(const Ray & ray) const{ // interseca l'albero
    tramite un raggio e restituisce le informazioni trovate tramite l'oggetto
    Intersection
  Intersection hit; // creo un oggetto Intersection vuoto
  hit.t = cam->far; // Impongo una distanza massima per l'intersezione, in questo
      caso Ăš quella del parametro far della Camera

  turbo->startTraversing(ray, hit); // lancia il traversing dell'albero

  return hit;
}

Intersection World::intersectTree(const Ray & ray, double tFar) const{ //
    interseca l'albero tramite un raggio con distanza massima tFar e restituisce le
    informazioni trovate tramite l'oggetto Intersection
  Intersection hit;
  hit.t = tFar; // in questo caso la distanza massima dipende da tFar - viene
      utilizzato per trovare le intersezioni dei raggi d'ombra tra un punto e una
      luce

  turbo->startTraversing(ray, hit);

  return hit;
}


void World::renderScene(){ // funzione principale del rendering della scena
  cout << "< Rendering Scene >" << endl;
  Ray ray; // creo un raggio vuoto
  Vec3d Lo; // creo una tripla per contenere il valore di radianza sul pixel

  stats->initialize();
  stats->timer->start();

  for(int s = 1; s <= sampler->samples; ++s){ // questo for cicla sul numero di
      campioni del Sampler
    cout << "\tsample : " << s << "\n";
    stats->openProgressBar();
    double n = 1./s; // calcolo il reciproco dell'indice del campione per usarlo
        nella media delle radianze dei campioni
    for(int i = 0; i < film->area(); ++i){ // questo for cicla sul numero di pixel
        dell'immagine
      px = i%film->width; py = i/film->width; // trovo gli indici bidimensionali
          del pixel partendo da quello del ciclo
      sampler->setSeed(i); // cambio seed del generatore di numeri pseudocasuali
          per ogni pixel con l'indice del ciclo (evita dei fenomeni di correlazione
          dell'immagine)
```

```
      cam->generateRay(ray, sampler->sampleScreen(i, film->width)); // genero un
         raggio dalla camera, partendo da un campione generato dal Sampler sulla
         griglia dei pixel
      Lo = integrator->solve(ray); // calcolo la radianza tramite il raggio
         passante nel pixel
      film->impressAdd(i, Lo); // salvo il valore di radianza nel buffer
         dell'immagine
      stats->next();
      stats->checkProgress();
    }
    sampler->nextSample(); // passo al campione successivo
    stats->closeProgressBar();
    film->finalize(0., 1., n); // faccio la media dei campioni attuali e poi clampo
       il valore tra 0 e 1
    film->save(); // salvo l'immagine sul disco
  }
  stats->timer->stop();
  stats->printStats();
  cout << "</ Scene Rendered >" << endl;
}
```

world.hpp:

```
#ifndef _WORLD_
#define _WORLD_

#include <vector>
#include <unordered_map>
#include "utilities/intersection.hpp"

class Accelerator;
class Camera;
class Film;
class Light;
class Material;
class ObjectGroup;
class Ray;
class Stats;
class Parser;
class Sampler;
class Environment;
class Integrator;

class World{
public:
  World();
  ~World();

  void addLight(Light *);
```

```
  void addModel(ObjectGroup *);
  void addMaterial(const std::string &, Material *);
  void buildScene(const std::string &);
  Intersection intersectTree(const Ray &) const;
  Intersection intersectTree(const Ray &, double) const;
  Vec3d traceRay(Ray &, int) const;
  void renderScene();
  void displayImage() const;
  void Test();

  std::vector<Light *> lights; // vettore delle luci
  std::vector<ObjectGroup *> models; // vettore dei modelli (aggregati di
     primitive)
  std::vector<Material *> Materials; // vettore dei materiali
  std::unordered_map<std::string, Material *> materials; // mappa che associa un
     materiale al suo nome
  Environment * env; // ambiente
  Camera * cam; // videocamera
  Film * film; // immagine
  Accelerator * turbo; // struttura di accelerazione
  Stats * stats; // struttura per le statistiche
  Parser * parser; // parser
  Sampler * sampler; // campionatore
  Integrator * integrator; // integratore
};


#endif
```

ACCELERATORS:

---

aabb.hpp:

```
/*
  File di definizione della classe AABB, descrive un Axis Aligned Bounding Box.
*/

#ifndef _AABB_
#define _AABB_

#include <limits>
#include <iostream>
#include "../geometry/vec3d.hpp"
#include "../geometry/ray.hpp"
#include "../utilities/math.hpp"


class AABB{
public:
```

```
AABB(): min(-std::numeric_limits<double>::max()),
    max(std::numeric_limits<double>::max()){}
AABB(const Vec3d & v): min(-v), max(v){
  if(min > max){
    min = v;
    max = -v;
  }
}
AABB(const Vec3d & _min, const Vec3d _max): min(_min), max(_max){
  if(min > max){
    min = _max;
    max = _min;
  }
}
AABB(const AABB & b): min(b.min), max(b.max){}

AABB operator+(const AABB & b) const{ // calcola l'unione insiemistica tra due
    AABB
  Vec3d tmin = b.min, tmax = b.max;
  for(int i = 0; i < 3; ++i){
    tmin[i] = tmin[i] < min[i] ? min[i] : min[i];
    tmax[i] = tmax[i] > max[i] ? max[i] : max[i];
  }
  return AABB(tmin, tmax);
}
AABB operator+(const std::initializer_list<Vec3d> & b) const{
  return *this + AABB(*b.begin(), *b.end());
}
AABB & operator+=(const AABB & b){
  for(int i = 0; i < 3; ++i){
    min[i] = min[i] < b.min[i] ? min[i] : b.min[i];
    max[i] = max[i] > b.max[i] ? max[i] : b.max[i];
  }
  return *this;
}
AABB & operator+=(const std::initializer_list<Vec3d> & b){
  return *this += AABB(*b.begin(), *b.end());
}
AABB & operator=(const AABB & b){min = b.min; max = b.max; return *this;}

Vec3d diag() const{ // calcola la diagonale del AABB
  return max - min;
}

bool hit(const Ray & ray, double & tN, double & tF) const{ // trova
    l'intersezione tra un raggio e l'AABB ritornando la distanza minima e massima
  double tmin = D_EPS, tmax = std::numeric_limits<double>::max();
  for(int i = 0; i < 3; ++i){
    double t0 = (min[i] - ray.o[i])*ray.dInv[i];
```

```
      double t1 = (max[i] - ray.o[i])*ray.dInv[i];
      if(ray.s[i]){
        double temp = t0;
        t0 = t1; t1 = temp;
      }
      tmin = t0 > tmin ? t0 : tmin;
      tmax = t1 < tmax ? t1 : tmax;
      if(tmax <= tmin)
        return false;
    }
    tN = tmin; tF = tmax;
    return true;
  }

  std::string toString() const{
    return "[ " + min.toString() + ", " + max.toString() + ", Extention: " +
        diag().toString() + "]";
  }


  Vec3d min, max; // la descrizione del AABB avviene tramite il suo punto di
      minimo e di massimo
};


inline std::ostream & operator<<(std::ostream & os, const AABB & b){
  return os << "AABB : { " << b.min << ", " << b.max << " }";
}

#endif
```

---

accelerator.cpp:

```
/*
  File di definizione della classe Accelerator, il suo scopo Ăš quello di fornire
      una descrizione spaziale della scena
  sotto forma di un albero binario di AABB, le quali foglie sono le uniche a
      contenere le primitive geometriche.
  Viene interrogato dalla classe World per controllare le intersezioni di un
      raggio con le primitive della scena,
  l'operazione di intersezione risulterÃ  piÃč veloce (in media) dell'intersecare
      il raggio con tutte le primitive,
  poichÃš il raggio verrÃ  intersecato ricorsivamente con l'albero fino ad
      arrivare alla foglia contenete un numero minore di primitive.
*/

#include "accelerator.hpp"
#include "../geometricObjects/objectGroup.hpp"
#include "../world.hpp"
```

```
#include <algorithm>

Accelerator::Accelerator(World * _w): genMax(6), nPrimsMax(4), w(_w){}

Accelerator::Accelerator(int g, int n, World * _w): genMax(g), nPrimsMax(n),
    w(_w){}

Accelerator::Accelerator(const Accelerator & a): genMax(a.genMax),
    nPrimsMax(a.nPrimsMax), w(a.w), root(a.root){}

Accelerator::~Accelerator(){
  delete root;
}

Accelerator & Accelerator::operator=(const Accelerator & a){
  genMax = a.genMax; nPrimsMax = a.nPrimsMax; w = a.w; root = a.root;
  return *this;
}

void Accelerator::setGenMax(int g){genMax = g;} // setta la generazione massima

void Accelerator::setNPrimsMax(int n){nPrimsMax = n;} // setta il numero massimo
    di primitive per foglia

void Accelerator::setWorld(World * _w){w = _w;}// setta il numero di primitive per
    foglia massimo

int Accelerator::getGenMax() const{return genMax;} // ritorna la generazione
    massima

int Accelerator::getNPrimsMax() const{return nPrimsMax;} // ritorna il numero
    massimo di primitive per foglia

World * Accelerator::getWorld() const{return w;} // ritorna il riferimento al World

Node * Accelerator::getRoot() const{return root;} // ritorna il riferimento al
    nodo root

void Accelerator::init(){ // inizializza l'acceleratore
  if(w->models.size() == 0){
    root = new Node(0, 0, new AABB());
    return;
  }
  buildRoot(); // costruisce il nodo root
  buildSceneTree(root); // costruisce l'albero partendo dalla root
}

void Accelerator::buildRoot(){ // costruisce il nodo root a partire dagli
    ObjectGroup nel vettore models del World
```

```
  AABB * bbox = new AABB(Vec3d(0)); // creo un AABB di volume nullo
  for(auto & x : w->models){ // ciclo su tutti gli ObjectGroup
    bbox->operator+=(*x->bbox); // faccio l'unione insiemistica dei loro AABB
  }

  Vec3d boxDiag = bbox->diag()*.5; // calcolo la semidiagonale del AABB
  int axis = 0;
  for(int i = 1; i < 3; ++i) // trovo l'asse maggiore del AABB
    if(boxDiag[i] > boxDiag[i-1])
      axis = i;

  root = new Node(0, axis, bbox); // creo la root tramite il AABB trovato e l'asse
      di taglio

  for(auto & model : w->models) // carico una copia delle primitive del World nel
      nodo root
    importPrims(model);
}

bool Accelerator::buildSceneTree(Node * node){ // costruisce un generico nodo a
    partire dal padre e le informazioni che contiene
  if(node->gen == genMax || node->prims.size() <= nPrimsMax ){ // controllo se
      sono alla generazione massima o sotto al numero massimo di primitive per
      nodo, se si esco
    return false;
  }

  Vec3d min = node->bv->min, max = node->bv->max; // creo una copia del minimo e
      massimo
  int axis = node->axis; // creo una copia dell'asse su cui devo sezionare

  double cutAxis = ((max - min)*.5)[axis]; // trovo il valore della componente
      dell'asse di taglio della semidiagonale

  Vec3d leftMax = max, rightMin = min; // trovo i quattro vertici che descrivono i
      due nuovi AABB figli, prima prendo il max sinistro dal max del padre e il min
      destro dal min del padre
  leftMax[axis] -= cutAxis + D_EPS; // poi tolgo cutAxis sulla componente di
      taglio e un epsilon per evitare sovrapposizioni
  rightMin[axis] += cutAxis - D_EPS; // la stessa cosa la faccio simmetricamente
      sul minimo destro

  int nextAxis = (axis+1)%3; // passo in modo ciclico all'asse successivo
  node->addChild(new Node(node->gen + 1, nextAxis, new AABB(min,leftMax))); //
      aggiungo ai figli del nodo questi due nuovi figli creati dai AABB appena
      trovati e con generazione aumentata di 1
  node->addChild(new Node(node->gen + 1, nextAxis, new AABB(rightMin,max)));
```

```
  for(auto & x : node->prims){ // per ogni primitiva contenuta nel padre
    if(x->getMin()[axis] <= node->children[0]->bv->max[axis]){ // se il min del
        AABB della primitiva si trova a sinistra del valore di taglio
      node->children[0]->addPrim(x); // la aggiungo al figlio di sinistra rispetto
          al valore di taglio
      if(x->getMax()[axis] >= node->children[1]->bv->min[axis]) // se si trova
          anche a destra
        node->children[1]->addPrim(x); // la aggiungo anche al figlio di destra
    }
    else
      node->children[1]->addPrim(x); // altrimenti la aggiungo solo a quello di
          destra
  }

  node->prims.clear(); // svuoto le primitive dal padre

  // std::cout << "----------------------\n";
  // std::cout << "gen: " << node->gen << "\nprim: \n" <<
      node->children[0]->prims.size() << " - " << node->children[1]->prims.size()
      << "\n";

  buildSceneTree(node->children[0]); // chiamo ricorsivamente la funzione di
      costruzione sui due nuovi figli
  buildSceneTree(node->children[1]);

  return true;
}

void Accelerator::startTraversing(const Ray & ray, Intersection & hit){ // metodo
    che inizializza il traversing dell'albero
  double tN, tF;
  if(root->bv->hit(ray, tN, tF)){ // se colpisco la root della scena
    ++hit.nHits;
    pN = ray(tN);
    pF = ray(tF);
    traverse(ray, hit, root); // lancio il traversing sulla root
  }
}

bool Accelerator::traverse(const Ray & ray, Intersection & hit, Node * node)
    const{ // metodo che traversa un generico nodo dato il raggio e le informazioni
    Intersection

  if(node->isLeaf()) // se mi trovo in una foglia
    return node->intersectPrims(ray, hit); // controllo tutte le intersezioni con
        le primitive
  else if(node->children.size() == 0) // se invece non ha figli, esco
    return false;
```

```
  double x,y;

  if(node->children[0]->bv->hit(ray, x, y)) // se ho un intersezione con il box di
      sinistra
    traverse(ray, hit, node->children[0]); // traverso quel box

  if(node->children[1]->bv->hit(ray, x, y)) // se ho un intersezione con il box di
      sinistra
    traverse(ray, hit, node->children[1]); // traverso quel box

}

void Accelerator::importPrims(ObjectGroup * model){ // metodo che importa tutte le
    primitive della scena nella root in modo ricorsivo
    if(model->nPrims > 0)
      for(auto & obj : model->objs)
        root->prims.push_back(obj);
    else if(model->nChilds > 0){
      for(auto & child : model->childs)
        importPrims(child);
    }
}

std::ostream & operator<<(std::ostream & os, const Accelerator & acc){
  return os << "Accelerator : { GenMax: " << acc.genMax << ", nPrimsMax: " <<
      acc.nPrimsMax
           << ", Root: " << *acc.root << " }";
}
```

---

accelerator.hpp:

```
#ifndef _ACCELERATOR_
#define _ACCELERATOR_

#include "node.hpp"

class World;
class Intersection;
class ObjectGroup;

class Accelerator{
public:
  Accelerator(World *);
  Accelerator(int, int, World *);
  Accelerator(const Accelerator &);
  ~Accelerator();

  Accelerator & operator=(const Accelerator &);
```

```
  void setGenMax(int);
  void setNPrimsMax(int);
  void setWorld(World *);
  int getGenMax() const;
  int getNPrimsMax() const;
  World * getWorld() const;
  Node * getRoot() const;

  void init();
  void buildRoot();
  bool buildSceneTree(Node *);
  void startTraversing(const Ray &, Intersection &);
  bool traverse(const Ray &, Intersection &, Node *) const;
  void importPrims(ObjectGroup *);

  friend std::ostream & operator<<(std::ostream &, const Accelerator &);

private:
  int genMax; // generazione massima dell'albero
  uint nPrimsMax; // numero massimo di primitive per foglia
  World * w; // riferimento al World
  Node * root; // nodo radice principale
  Vec3d pN, pF;
};

std::ostream & operator<<(std::ostream &, const Accelerator &);

#endif
```

node.cpp:

```
/*
  File di definizione della classe Node, descrive un nodo dell'albero binario
      usato nell'acceleratore
*/

#ifndef _NODE_
#define _NODE_

#include <vector>
#include "aabb.hpp"
#include "../utilities/intersection.hpp"
#include "../geometricObjects/primitive.hpp"

class Node{
public:
  Node(): gen(0), axis(0), bv(new AABB()){}
  Node(int g, int a, AABB * _bv): gen(g), axis(a), bv(_bv){}
```

```
  Node(const Node & n): gen(n.gen), axis(n.axis), children(n.children),
      prims(n.prims){
    *bv = *n.bv;
  }
  ~Node(){
    delete bv;
    for(auto & x : children)
      delete x;
  }

  Node & operator=(const Node & n){
    gen = n.gen; axis = n.axis;
    *bv = *n.bv; children = n.children; prims = n.prims;
    return *this;
  }

  bool isLeaf() const{return prims.size() != 0;} // indica se il nodo Ăš una
      foglia dell'albero
  void addChild(Node * n){children.push_back(n);} // aggiunge un nodo figlio
      all'array dei figli
  void addPrim(Primitive * p){prims.push_back(p);} // aggiunge una primitiva
      all'array delle primitive
  bool intersectPrims(const Ray & ray, Intersection & hit){ // controlla
      iterativamente le intersezioni con tutte le primitive contenute nel nodo
    size_t size = prims.size();
    for(size_t i = 0; i < size; ++i){
      prims[i]->hit(ray, hit);
      hit.nHits++;
    }
    return hit.hit;
  }

  int gen, axis; // descrivono la generazione del nodo e l'asse su cui il nodo
      deve essere tagliato
  AABB * bv; // AABB per descrivere il volume del nodo
  std::vector<Node *> children; // vettore dei nodi figli
  std::vector<Primitive *> prims; // vettore delle primitive contenute
};

inline std::ostream & operator<<(std::ostream & os, const Node & n){
  return os << "Node : { " << *n.bv << ", Nodes: " << n.children.size() << ",
      Prims: "
            << n.prims.size() << " }";
}


#endif
```

BIDIRECTIONAL REFLECTANCE DISTRIBUTION FUNCTIONS:

addBrdf.hpp:

```
/*
  File di definizione della classe AddBrdf, specializza la brdf multipla in una di
     tipo additivo, ovvero nell'evaluate somma i diversi contributi dell brdf
*/

#ifndef _ADD_BRDF_
#define _ADD_BRDF_

#include "multipleBrdf.hpp"
#include "../geometry/vec3d.hpp"

class AddBrdf : public MultipleBrdf{
public:
  AddBrdf(){}
  AddBrdf(Brdf * b): MultipleBrdf(b){}
  AddBrdf(const std::vector<Brdf *> & bv): MultipleBrdf(bv){}
  ~AddBrdf(){}

  Vec3d evaluate(const Vec3d & wo, const Vec3d & wi, Intersection & hit) const{ //
     metodo che valuta la brdf multipla
    Vec3d out;
    for(auto & x : brdfs){ // per tutte le brdf
      out += x->evaluate(wo, wi, hit); // sommo le valutazioni
    }
    return out;
  }

  std::string toString() const{
    return "AddBrdf: " + MultipleBrdf::toString();
  }

};

#endif
```

blinn.hpp:

```
/*
  File di definizione della classe Blinn, modella la brdf di Blinn
*/

#ifndef _BLINN_BRDF_
#define _BLINN_BRDF_

#include "brdf.hpp"
```

```
#include "../utilities/texture.hpp"
#include "../utilities/intersection.hpp"

class Blinn: public Brdf{
public:
  Blinn(): Brdf(), specular(1.), ks(1.), alpha(2048){}
  Blinn(const Vec3d & s, int a): Brdf(), specular(s), ks(1.), alpha(a){}
  Blinn(const Vec3d & s, double _ks, int a): Brdf(), specular(s), ks(_ks),
      alpha(a){}
  Blinn(Texture * _txt, int a): Brdf(_txt), ks(1.), alpha(a){}
  Blinn(Texture * _txt, double _ks, int a): Brdf(_txt), ks(_ks), alpha(a){}
  ~Blinn(){}

  Vec3d evaluate(const Vec3d & wo, const Vec3d & wi, Intersection & hit){ //
      metodo che valuta la brdf di blinn
    Vec3d H = (wo + wi).hat(); // calcolo il vettore Halfway dalla direzione di
        entrata e di uscita
    if(txt) // se sto usando una texture
      return ks*txt->getValue(hit.texture)*pow(hit.normal*H,alpha); // calcolo il
          valore campionando la texture in base alle informazioni di intersezione
    else
      return ks*specular*pow(hit.normal*H,alpha); // altrimenti uso il valore
          specular
  }

  Vec3d sample(const Vec3d & wo, Vec3d & wi, const Vec3f & xi, Intersection &
      hit){ // metodo che campiona la brdf (usato solo nel pathtracer per modellare
      riflessi speculari perfetti)
    wi = (-wo).reflect(hit.normal); // calcolo la direzione riflessa
    return evaluate(wo, wi, hit); // valuto la brdf
    // wi = (hit.TBN->operator*(Vec4d(toHemisphere(xi), 0.))).xyz().hat();
    // return evaluate(wo, wi, hit)*TWO_PI;
  }

  std::string toString() const{
    return "Blinn: [ Specular: " + specular.toString() + ", Ks: " +
        std::to_string(ks) + ", Alpha: " + std::to_string(alpha) + ", " + (txt ?
        txt->toString() : "null") + "]";
  }

  Vec3d specular; // colore speculare
  double ks; // coefficente speculare
  int alpha; // esponente di blinn
};


#endif
```

---

brdf.hpp:

```
/*
  File di definizione della classe Brdf, modella il comportamento di una brdf
      generica
*/

#ifndef _BRDF_
#define _BRDF_

#include <string>
#include "../utilities/texture.hpp"

class Vec3d;
class Intersection;

class Brdf{
public:
  Brdf(): txt(nullptr){}
  Brdf(Texture * _txt): txt(_txt){}
  virtual ~Brdf(){
    delete txt;
  }

  virtual Vec3d evaluate(const Vec3d &, const Vec3d &, Intersection &) = 0; //
      metodo astratto per la valutazione della brdf

  virtual Vec3d sample(const Vec3d &, Vec3d &, const Vec3f &, Intersection &) = 0;
      // metodo astratto per il campionamento della brdf

  virtual std::string toString() const{return "";}

  Texture * txt; // texture usata per il valore diffusivo/speculare, in base alla
      classe di specializzazione

};

#endif
```

cooktorrance.hpp:

```
#ifndef _COOK_TORRANCE_BRDF_
#define _COOK_TORRANCE_BRDF_

#include "brdf.hpp"
#include "../utilities/texture.hpp"
#include "../utilities/intersection.hpp"

extern int px, py;

class CookTorrance: public Brdf{
```

```
public:
  CookTorrance(): Brdf(), F0(1.), r(1.), kf(1.), rmap(nullptr){}
  CookTorrance(const Vec3d & s, double _r): Brdf(), F0(s), r(_r), kf(1.),
      rmap(nullptr){}
  CookTorrance(Texture * _smap, double _r, double _kf): Brdf(_smap), r(_r),
      kf(_kf), rmap(nullptr){}
  CookTorrance(const Vec3d & s, Texture * _rmap, double _kr): Brdf(), F0(s),
      kr(_kr), rmap(_rmap){}
  CookTorrance(Texture * _smap, Texture * _rmap, double _kf, double _kr):
      Brdf(_smap), kf(_kf), kr(_kr), rmap(_rmap){}
  ~CookTorrance(){
    delete rmap;
  }

  Vec3d evaluate(const Vec3d & wo, const Vec3d & wi, Intersection & hit){ //
      metodo che valuta la brdf di cook-torrance
    Vec3d H = (wo + wi).hat();
    double NoWi = hit.normal*wi, NoWo = hit.normal*wo;
    return D(hit.normal*H, hit.texture)*F(wi*H, hit.texture)*G(NoWi, NoWo,
        hit.texture)/(4*NoWi*NoWo);
  }

  Vec3d sample(const Vec3d & wo, Vec3d & wi, const Vec3f & xi, Intersection &
      hit){ // metodo che campiona la brdf
    double a = getR(hit.texture);
    a *= a;
    double phi = TWO_PI*xi.x;
    double costheta = sqrt((1. - xi.y)/((a*a - 1.)*xi.y + 1.));
    double sintheta = sqrt(1. - costheta*costheta);
    Vec3d H(sintheta*cos(phi), sintheta*sin(phi), costheta);
    H = (hit.TBN->operator*(Vec4d(H, 0.))).xyz().hat();
    double HoWo = H*wo;
    if(HoWo < D_EPS)
      return Vec3d();
    wi = (-wo).reflect(H);
    double NoH = hit.normal*H;
    return evaluate(wo, wi, hit)/(D(NoH, hit.texture)*NoH)*(4.*HoWo);
  }

  double D(double NoH, const Vec3d & uv) const{ // metodo che calcola la
      distribuzione di microfaccette
    double a = getR(uv);
    a *= a;
    double x = a/(NoH*NoH*(a*a - 1.) + 1.);
    return x*x*invPI;
  }

  Vec3d F(double WioH, const Vec3d & uv) const{ // metodo che calcola la
      componente di Fresnel
```

```
    Vec3d f0 = getS(uv);
    return f0 + (Vec3d(1.) - f0)*pow(1. - WioH, 5);
  }

  double G(double NoWi, double NoWo, const Vec3d & uv) const{ // metodo che
      calocla la componente di Masking
    double k = getR(uv);
    k *= k*.5;
     return G1(NoWi, k)*G1(NoWo, k);
  }

  double G1(double NoW, double k) const{
    return NoW/(NoW*(1. - k) + k);
  }

  Vec3d getS(const Vec3d & uv) const{
    if(txt)
      return txt->getValue(uv)*kf;
    else
      return F0;
  }

  double getR(const Vec3d & uv) const{
    if(rmap)
      return rmap->getValue(uv)[0]*kr;
    else
      return r;
  }

  std::string toString() const{
    std::string s;
    if(txt)
      s += "sMap: " + txt->toString() + ", kf: " + std::to_string(kf);
    else
      s += "Specular: " + F0.toString();
    if(rmap)
      s += ", rMap: " + rmap->toString() + ", kr: " + std::to_string(kr);
    else
      s += ", Roughness: " + std::to_string(r);

    return "CookTorrance: [ " + s + "]";
  }

  Vec3d F0;
  double r, kf, kr;
  Texture * rmap;
};
```

```
#endif
```

lambert.hpp:

```
#ifndef _LABERT_BRDF_
#define _LABERT_BRDF_

#include "brdf.hpp"
#include "../utilities/intersection.hpp"
#include "../utilities/constants.hpp"

class Lambert: public Brdf{
public:
  Lambert(): Brdf(), albedo(1.), kd(1.){}
  Lambert(const Vec3d & a): Brdf(), albedo(a), kd(1.){}
  Lambert(const Vec3d & a, double _kd): Brdf(), albedo(a), kd(_kd){}
  Lambert(Texture *_txt): Brdf(_txt), kd(1.){}
  Lambert(Texture *_txt, double _kd): Brdf(_txt), kd(_kd){}
  ~Lambert(){}

  Vec3d evaluate(const Vec3d & wo, const Vec3d & wi, Intersection & hit){ //
      metodo che valuta la brdf di lambert
    if(txt) // se sto usando una texture
      return kd*txt->getValue(hit.texture)*invPI; // calcolo il valore campionando
          la texture in base alle informazioni di intersezione
    else
      return kd*albedo*invPI; // altrimenti uso il valore dell'albedo
  }

  Vec3d sample(const Vec3d & wo, Vec3d & wi, const Vec3f & xi, Intersection &
      hit){ // metodo che campiona la brdf tramite xi campione uniforme in
      [0,1]x[0,1]
    wi = (hit.TBN->operator*(Vec4d(toCosineHemisphere(xi), 0.))).xyz().hat(); //
        dato xi, lo mappo sull'emisfero pesato con il coseno e passo da tangent
        space a world space
    return evaluate(wo, wi, hit)*PI; // valuto la brdf con wi appena calcolato
  }

  std::string toString() const{
    return "Lambert: [ Albedo: " + albedo.toString() + ", Kd: " +
        std::to_string(kd) + ", " + (txt ? txt->toString() : "null") + "]";
  }

  Vec3d albedo;
  double kd;
};

#endif
```

multipleBrdf.hpp:

```
/*
  File di definizione della classe MultipleBrdf, gestisce le diverse brdf di un
      materiale
*/

#ifndef _MULTIPLE_BRFD_
#define _MULTIPLE_BRFD_

#include "brdf.hpp"
#include <vector>
#include <string>

class Vec3d;
class Intersection;

class MultipleBrdf{
public:
  MultipleBrdf(){}
  MultipleBrdf(Brdf * b){addBrdf(b);}
  MultipleBrdf(const std::vector<Brdf *> & bv): brdfs(bv){}
  virtual ~MultipleBrdf(){
    for(auto & x : brdfs)
      delete x;
  }

  void addBrdf(Brdf * b){ // aggiunge una brdf al vettore di brdf
    brdfs.push_back(b);
  }

  virtual Vec3d evaluate(const Vec3d & wo, const Vec3d & wi, Intersection & hit)
      const = 0; // metodo astratto per la valutazione della brdf multipla

  double pdf() const{ // ritorna l'inverso della pdf uniforme sul vettore di brdf
    return static_cast<double>(brdfs.size());
  }

  virtual std::string toString() const{
    std::string s;
    s += brdfs[0]->toString();
    for(u_int i = 1; i < brdfs.size(); ++i){
      s += " - " + brdfs[i]->toString();
    }
    return s;
  }

  std::vector<Brdf *> brdfs; // vettore di brdf
};
```

```
#endif
```

CAMERAS:

camera.cpp:

```
/*
  File di definizione della classe Camera, modella una generica videocamera che ha
     il compito di descrivere come l'osservatore guarda la scena
*/

#include "camera.hpp"
#include "../geometry/mat4f.hpp"

using namespace Mat4F;

Camera::Camera(): width(800), heigth(600), near(1e-5), far(1e5), fov(90)
               , eye(100.f, 0.f, 0.f), lookAt(0.f), up(0.f, 0.f, 1.f){
  buildCamToWorld();
}

Camera::Camera(int w, int h, double n, double f, float FOV, const Vec3f & e,
             const Vec3f & l, const Vec3f & u): width(w), heigth(h), near(n)
             , far(f), fov(FOV), eye(e), lookAt(l), up(u.hat()){
  buildCamToWorld();
}

Camera::Camera(const Camera & c): width(c.width), heigth(c.heigth), near(c.near),
    far(c.far)
             , fov(c.fov), eye(c.eye), lookAt(c.lookAt), up(c.up){
  CamToWorld = c.CamToWorld;
}

Camera::~Camera(){}

Camera & Camera::operator=(const Camera & c){
  width = c.width; heigth = c.heigth; near = c.near; far = c.far; fov = c.fov;
  eye = c.eye; lookAt = c.lookAt; up = c.up; CamToWorld = c.CamToWorld;
  return *this;
}

void Camera::buildCamToWorld(){ // metodo che costruisce la matrice di cambiamento
    di base tra camera e mondo
  Vec3f w = (eye - lookAt).hat(); // si parte costruendo il vettore di
     osservazione ribaltato
  Vec3f t = up^w; // si opera un prodotto vettoriale con il vettore del cielo per
     trovare un suo vettore ortonormale dipendente da questi due
```

```
  Vec3f u = (t == Vec3f()) ? (up - Vec3f(1e-6).hat())^w : t; // se il risultato Ăš
      un vettore nullo ( dovuto a una scelta incorretta del vettore up ) si
      perturba l'up con un valore piccolo e si ripete il calcolo
  Vec3f v = w^u; // si opera l'ultimo prodotto vettoriale tra i due vettori
      calcolati precedentemente
  CamToWorld = Mat4f(u,v,w,eye)*scaling(1.f,1.f,-1.f); // si costruisce la matrice
      grazie a questi tre vettori che ne definiranno la base dello spazio della
      camera e il punto eye che ne definisce l'origine
}

std::string Camera::toString() const{
  return "{ Res: " + std::to_string(width) + "x" + std::to_string(heigth)
         + "\nNear: " + std::to_string(near) + ", Far: " + std::to_string(far) +
             ", FOV: " + std::to_string(fov)
         + "\nEye: " + eye.toString() + ", LookAt: " + lookAt.toString() + ", Up:
             " + up.toString() + " }";
}
```

---

camera.hpp:

```
#ifndef _CAMERA_
#define _CAMERA_

#include <iostream>
#include "../geometry/vec3f.hpp"
#include "../geometry/mat4f.hpp"

class Mat4f;
class Ray;

class Camera{
public:
  Camera();
  Camera(int, int, double, double, float, const Vec3f &, const Vec3f &, const
      Vec3f &);
  Camera(const Camera &);
  virtual ~Camera();

  Camera & operator=(const Camera &);

  void buildCamToWorld();
  virtual void buildRasterToCam() = 0;
  virtual void generateRay(Ray &, const Vec3f &) const = 0;

  virtual std::string toString() const;

  int width, heigth; // larghezza e altezza dell'immagine
  double near, far; // distanza minima e massima di visuale
```

```
  float fov; // field of view
  Vec3f eye, lookAt, up; // posizione della camera, punto osservato, vettore
      indicante il cielo
  Mat4f CamToWorld; // matrice di cambiamento di base dalle coordinate della
      camera a quelle del mondo
};

#endif
```

perspective.cpp:

```
/*
  File di definizione della classe Perspective, specializzazione della classe
      Camera che modella una visuale prospettica
*/

#include "perspective.hpp"
#include "../geometry/mat4f.hpp"
#include "../geometry/ray.hpp"
#include "../utilities/math.hpp"
#include "../samplers/sampler.hpp"
#include <math.h>

using namespace Mat4F;

PerspCamera::PerspCamera(): Camera(){
  buildRasterToCam();
}

PerspCamera::PerspCamera(int w, int h, double n, double f, float FOV
                      , const Vec3f & e, const Vec3f & l, const Vec3f & u):
                      Camera(w, h, n, f, FOV, e, l, u){
                        buildRasterToCam();
                      }

PerspCamera::PerspCamera(const PerspCamera & p): Camera(p){
  RasterToCam = p.RasterToCam;
}

PerspCamera::~PerspCamera(){}

PerspCamera & PerspCamera::operator=(const PerspCamera & p){
  Camera::operator=(p);
  RasterToCam = p.RasterToCam;
  return *this;
}

void PerspCamera::buildRasterToCam(){ // metodo che costruisce la matrice di
    cambiamento di coordinate tra lo spazio Raster a quello della Camera
```

```
  float ratio = static_cast<float>(width)/heigth, alpha = toRad(fov*.5);
  float sw[4];
  if(ratio >= 1.f){
    sw[0] = -ratio; sw[1] = ratio; sw[2] = -1.f; sw[3] = 1.f;
  }
  else{
    sw[0] = -1.f; sw[1] = 1.f; sw[2] = -1.f/ratio; sw[3] = 1.f/ratio;
  }
  Mat4f R2S = inverse(scaling(width, heigth, 1.f)*
                      scaling(1.f/(sw[1]-sw[0]), 1.f/(sw[2]-sw[3]), 1.f)*
                      translation(-sw[0],-sw[3],0.f));
  Mat4f Persp(1.f, 0.f,         0.f,                0.f,
              0.f, 1.f,         0.f,                0.f,
              0.f, 0.f, far/(far-near), -far*near/(far-near),
              0.f, 0.f,         1.f,                0.f);
  Mat4f S2C = inverse(scaling(1.f/tanf(alpha), 1.f/tanf(alpha), 1.f)*
                      Persp);
  RasterToCam = S2C*R2S;
}

void PerspCamera::generateRay(Ray & ray, const Vec3f & sp) const{ // metodo che
    genera un raggio partendo da un punto sullo spazio Raster
  Vec4f p(sp, 1.f); // aumento il punto di una dimensione
  Vec4f pc = RasterToCam*p; // passo dallo spazio raster a quello della camera
  pc /= pc.w; // applico la divisione prospettica
  Vec4f pw = CamToWorld*Vec4f(pc.x, pc.y, pc.z, 0.f); // passo dallo spazio della
      camera al world
  ray.o = Vec3d(eye.x, eye.y, eye.z); // setto come origine del raggio la
      posizione della camera
  ray.d = Vec3d(pw.x, pw.y, pw.z).hat(); // setto come direzione del raggio il
      vettore appena calcolato normalizzato
  ray.refresh(); // aggiorno dei dati contenuti nel raggio utili all'intersezione
      con un Box
}

std::string PerspCamera::toString() const{
  return "Perspective Camera: " + Camera::toString();
}

std::ostream & operator<<(std::ostream & os, const PerspCamera & p){
  return os << "Perspective Camera: { Res: " << p.width << "x" << p.heigth
            << "\nNear: " << p.near << ", Far: " << p.far << ", FOV: " << p.fov
            << "\nEye: " << p.eye << ", LookAt: " << p.lookAt << ", Up: " << p.up
                << " }";
}
```

---

perspective.hpp:

```
#ifndef _PERSPECTIVE_CAMERA_
```

```
#define _PERSPECTIVE_CAMERA_

#include "camera.hpp"

class PerspCamera: public Camera{
public:
  PerspCamera();
  PerspCamera(int, int, double, double, float, const Vec3f &, const Vec3f &, const
      Vec3f &);
  PerspCamera(const PerspCamera &);
  ~PerspCamera();

  PerspCamera & operator=(const PerspCamera &);

  void buildRasterToCam();
  void generateRay(Ray &, const Vec3f &) const;

  std::string toString() const;

  Mat4f RasterToCam; // matrice che opera il passaggio dallo spazio Raster a
      quello della Camera
};

std::ostream & operator<<(std::ostream &, const PerspCamera &);


#endif
```

GEOMETRIC OBJECTS:

box.cpp:

```
/*
  File di definizione della classe Box, specializzazione della classe Primitive
      che modella un box allineato con gli assi cartesiani
*/

#include "box.hpp"
#include "../geometry/ray.hpp"
#include "../geometry/transform.hpp"
#include "../utilities/intersection.hpp"
#include "../utilities/constants.hpp"

Box::Box(): Primitive(), c(){
             min = -std::numeric_limits<double>::max();
             max = std::numeric_limits<double>::max();
           }
```

```
Box::Box(const Vec3d & v, Material * m): Primitive(m), c(){
  min = -v; max = v;
  if(min > max){
    min = v;
    max = -v;
  }
}

Box::Box(const Vec3d & _min, const Vec3d & _max, Material * m): Primitive(m){
  min = _min; max = _max;
  if(min > max){
    min = _max;
    max = _min;
  }
  c = calcCenter();
}

Box::Box(const Box & b): Primitive(b), c(b.c){}

Box & Box::operator=(const Box & b){
  Primitive::operator=(b);
  c = b.c;
  return *this;
}

bool Box::hit(const Ray & ray, Intersection & hit){ // metodo che testa
    l'intersezione tra un raggio ray e la primitive stessa, riportando le
    informazioni ottenute nell'oggetto intersection
  double tmin = D_EPS, tmax = std::numeric_limits<double>::max();
  for(int i = 0; i < 3; ++i){
    double t0 = (min[i] - ray.o[i])*ray.dInv[i];
    double t1 = (max[i] - ray.o[i])*ray.dInv[i];
    if(ray.s[i]){
      double temp = t0;
      t0 = t1; t1 = temp;
    }
    tmin = t0 > tmin ? t0 : tmin;
    tmax = t1 < tmax ? t1 : tmax;
    if(tmax <= tmin)
      return false;
  }
  if(tmin < hit.t){
    hit.t = tmin;
    hit.hit = true;
    hit.obj_ptr = this;
    return true;
  }
  return false;
}
```

```
Vec3d Box::calcNormal(Intersection & hit) const{ // metodo che calcola la normale
    a un punto
  Vec3d d = hit.hitPoint - c;
  double max = abs(d.x);
  int j = 0, sgn = (D_EPS < d.x) - (d.x < -D_EPS);
  for(int i = 1; i < 3; ++i){
    double val = abs(d[i]);
    if(val > max){
      max = val; j = i; sgn = (D_EPS < d[i]) - (d[i] < -D_EPS);
    }
  }
  Vec3d n;
  n[j] = sgn;
  return n;
}

void Box::applyTransform(Transform * tr){ // metodo che applica una trasformazione
    al box
  min = (tr->m->operator*(Vec4d(min,1.))).xyz();
  max = (tr->m->operator*(Vec4d(max,1.))).xyz();
  c = (tr->m->operator*(Vec4d(c,1.))).xyz();
}


std::ostream & operator<<(std::ostream & os, const Box & b){
  return os << "Box : { " << b.min << ", " << b.max << ", " << b.c << " }";
}
```

---

box.hpp:

```
#ifndef _BOX_
#define _BOX_

#include "primitive.hpp"
#include <limits>

class Box: public Primitive{
public:
  Box();
  Box(const Vec3d &, Material *);
  Box(const Vec3d &, const Vec3d &, Material *);
  Box(const Box &);

  Box & operator=(const Box &);

  bool hit(const Ray &, Intersection & );
  Vec3d calcNormal(Intersection & hit) const;
  Vec3d calcTexture(Intersection &) const{}
```

```
  void setBVertex(){}
  void applyTransform(Transform *);
  Vec3d calcCenter() const{return min + (max - min)*.5;}

  friend std::ostream & operator<<(std::ostream &, const Box & b);

private:
  Vec3d c; // centro del box
};

std::ostream & operator<<(std::ostream &, const Box & b);

#endif
```

disk.cpp:

```
/*
  File di definizione della classe Disk, specializzazione della classe Plane che
      modella un disco
*/

#include "disk.hpp"
#include "../geometry/ray.hpp"
#include "../geometry/transform.hpp"
#include "../utilities/constants.hpp"
#include "../utilities/intersection.hpp"

Disk::Disk(): Plane(), r(0.){}

Disk::Disk(const Vec3d & _p, const Vec3d & _n, double _r, Material * m): Plane(_p,
    _n, m), r(_r){
  setBVertex();
}

Disk::Disk(const Disk & d): Plane(d), r(d.r){}

Disk & Disk::operator=(const Disk & d){Plane::operator=(d); r = d.r; return *this;}

bool Disk::hit(const Ray & ray, Intersection & hit){ // metodo che testa
    l'intersezione tra un raggio ray e la primitive stessa, riportando le
    informazioni ottenute nell'oggetto intersection
  double t = ((p - ray.o)*n)/(ray.d*n);
  double d = (ray(t) - p).lengthSq();
  if(t > D_EPS && d <= r*r && t < hit.t){
    hit.t = t;
    hit.hit = true;
    hit.obj_ptr = this;
    return true;
  }
```

```
  else
    return false;
}

void Disk::applyTransform(Transform * tr){ // metodo che calcola la normale a un
    punto
  Plane::applyTransform(tr);
  double s = std::max(tr->m->operator()(0,0), tr->m->operator()(1,1));
  s = std::max(s, tr->m->operator()(2,2));
  r *= s;
  setBVertex();
}

std::ostream & operator<<(std::ostream & os, const Disk & d){
  return os << "Disk : { " << d.p << ", " << d.n << ", r : " << d.r << " }";
}
```

disk.hpp:

```
#ifndef _DISK_
#define _DISK_

#include "plane.hpp"

class Disk: public Plane{
public:
  Disk();
  Disk(const Vec3d & _p, const Vec3d & _n, double _r, Material * m);
  Disk(const Disk & d);

  Disk & operator=(const Disk & d);

  void setRadius(double _r){r = _r;}
  double getRadius() const{return r;}

  bool hit(const Ray &, Intersection &);
  void setBVertex(){min = p - r; max = p + r;}
  virtual void applyTransform(Transform *);

  friend std::ostream & operator<<(std::ostream &, const Disk &);

protected:
  double r; // raggio del disco
};

std::ostream & operator<<(std::ostream &, const Disk &);


#endif
```

objectGroup.hpp:

```
/*
  File di definizione della classe ObjectGroup, modella un oggetto complesso
      costituito da un albero di primitive
*/

#ifndef _OBJECT_GROUP_
#define _OBJECT_GROUP_

#include <vector>
#include "primitive.hpp"
#include "../geometry/transform.hpp"
#include "../accelerators/aabb.hpp"

class ObjectGroup{
public:
  ObjectGroup(): nPrims(0), nTotPrims(0), nChilds(0), o2w(new Transform()),
      bbox(new AABB(Vec3d(0))){}
  ObjectGroup(Transform * t): nPrims(0), nTotPrims(0), nChilds(0), o2w(t),
      bbox(new AABB(Vec3d(0))){}
  ObjectGroup(const ObjectGroup & og): nPrims(og.nPrims), nTotPrims(og.nTotPrims),
      nChilds(og.nChilds){
    objs.reserve(og.nPrims);
    for(u_int i = 0; i < og.nPrims; ++i){
      *objs[i] = *og.objs[i];
    }
    childs.reserve(og.nChilds);
    for(u_int i = 0; i < og.nChilds; ++i){
      *childs[i] = *og.childs[i];
    }
    *o2w = *og.o2w;
    *bbox = *og.bbox;
  }
  ~ObjectGroup(){
    for(auto & x : objs)
      delete x;
    for(auto & x : childs)
      delete x;
    delete o2w;
    delete bbox;
  }

  ObjectGroup & operator=(const ObjectGroup & og){
    for(auto & x : objs)
      delete x;
    for(auto & x : childs)
      delete x;
    objs.clear();
```

```
  childs.clear();
  if(og.nPrims > 0)
    objs.reserve(og.nPrims);
  if(og.nChilds > 0)
  childs.reserve(og.nChilds);
  for(u_int i = 0; i < og.nPrims; ++i){
    *objs[i] = *og.objs[i];
  }
  for(u_int i = 0; i < og.nChilds; ++i){
    *childs[i] = *og.childs[i];
  }
  nPrims = og.nPrims;
  nTotPrims = og.nTotPrims;
  nChilds = og.nChilds;
  *o2w = *og.o2w;
  *bbox = *og.bbox;
  return *this;
}

void addObject(Primitive * p){ // aggiunge una primitiva all'array di primitive
  objs.push_back(p);
  ++nPrims;
  ++nTotPrims;
}

void addChild(ObjectGroup * og){ // aggiunge un ObjectGroup all'array di figli
  childs.push_back(og);
  nTotPrims += og->nPrims;
  ++nChilds;
}

AABB setBVertex(){ // metodo che calcola ricorsivamente il bounding box
    dell'ObjectGroup
  if(nPrims > 0){ // se ci sono primitive
    for(u_int i = 0; i < nPrims; ++i){ // per tutte le primitive
      objs[i]->setBVertex(); // calcola il bounding box della primitiva
      AABB primBox(objs[i]->getMin(), objs[i]->getMax()); // costruisco il
          bounding box
      bbox->operator+=(primBox); // aggiungo al bounding box dell'ObjectGroup
          quello della primitiva
    }
  }
  else if(nChilds > 0){ // se ha dei figli
    for(u_int i = 0; i < nChilds; ++i){ // per tutti i figli
      bbox->operator+=(childs[i]->setBVertex()); // calcolo il bounding box del
          figlio e lo aggiungo al bounding box dell'objectGroup
    }
  }
  return *bbox; // ritorno il bounding box
```

```
  }

  void toWorld(Transform * tr){ // metodo che applica ricorsivamente una
      trasformazione a tutto l'albero
    Transform comp = tr ? tr->operator*(*o2w) : *o2w; // se tr non esiste allora
        uso la trasformazione dell'objectGroup
    if(nPrims > 0){ // se ci sono primitive
      for(auto & x : objs){ // per tutte le primitive
        x->applyTransform(&comp); // applico la trasformazione alla primitiva
      }
    }
    else if(nChilds > 0){ // se ci sono figli
      for(auto & x : childs){ // per tutti i figli
        x->toWorld(&comp); // richiamo la funzione sui figli
      }
    }
  }

  std::string toString() const{
    if(nPrims > 0){
      return "Object: [ nTotPrims: " + std::to_string(nTotPrims) + ", nPrims: " +
          std::to_string(nPrims) + ", BBox: " + bbox->toString() + "]";
    }
    else if(nChilds > 0){
      std::string s = "Group: [ ";
      for(auto & i : childs)
        s += i->toString() + ",\n\t\t";
      s += ", BBox: " + bbox->toString() + "]";
      return s;
    }
  }

  u_int nPrims, nTotPrims, nChilds; // numero di primitive contenute in questo
      livello, numero di primitive contenute in tutti i livelli sottostanti, numero
      di figli
  std::vector<Primitive *> objs; // vettore di primitive
  std::vector<ObjectGroup *> childs; // vettore di figli
  Transform * o2w; // trasformazione dell'oggetto
  AABB * bbox; // bounding box dell'oggetto
};

#endif
```

plane.cpp:

```
/*
  File di definizione della classe Plane, specializzazione della classe Primitive
      che modella un piano infinito
*/
```

```
#include "plane.hpp"
#include "../geometry/ray.hpp"
#include "../geometry/transform.hpp"
#include "../utilities/constants.hpp"
#include "../utilities/intersection.hpp"

Plane::Plane(): Primitive(), p(), n(){}

Plane::Plane(const Vec3d _p, const Vec3d & _n, Material * m): Primitive(m), p(_p),
    n(_n.hat()){}

Plane::Plane(const Plane & pl): Primitive(pl), p(pl.p), n(pl.n){}

Plane & Plane::operator=(const Plane & pl){
  Primitive::operator=(pl);
  p = pl.p; n = pl.n;
  return *this;
}

bool Plane::hit(const Ray & ray, Intersection & hit){ // metodo che testa
    l'intersezione tra un raggio ray e la primitive stessa, riportando le
    informazioni ottenute nell'oggetto intersection
  double t = ((p - ray.o)*n)/(ray.d*n);
  if(t > D_EPS && t < hit.t){
    hit.t = t;
    hit.hit = true;
    hit.obj_ptr = this;
    return true;
  }
  else
    return false;
}

void Plane::applyTransform(Transform * tr){ // metodo che applica una
    trasformazione al piano
  p = (tr->m->operator*(Vec4d(p,1.))).xyz();
  n = (tr->mTrInv->operator*(Vec4d(n,0.))).xyz();
}

void Plane::setBVertex(){}

std::ostream & operator<<(std::ostream & os, const Plane & pl){
  return os << "Plane : { " << pl.p << ", " << pl.n << " }";
}
```

---

plane.hpp:

```
#ifndef _PLANE_
```

```
#define _PLANE_

#include "primitive.hpp"
#include "../geometry/vec3d.hpp"

class Intersection;

class Plane: public Primitive{
public:
  Plane();
  Plane(const Vec3d _p, const Vec3d & _n, Material * m);
  Plane(const Plane & pl);

  Plane & operator=(const Plane & pl);

  void setPoint(const Vec3d & _p){p = _p;}
  void setNormal(const Vec3d & _n){n = _n;}
  Vec3d getPoint() const{return p;}
  Vec3d getNormal() const{return n;}

  virtual bool hit(const Ray &, Intersection &);
  Vec3d calcNormal(Intersection & hit) const{return n;}
  virtual Vec3d calcTexture(Intersection &) const{}
  virtual void applyTransform(Transform *);
  virtual void setBVertex();

  friend std::ostream & operator<<(std::ostream &, const Plane &);

protected:
  Vec3d p; // punto di applicazione della normale
  Vec3d n; // normale del piano
};

std::ostream & operator<<(std::ostream &, const Plane &);

#endif
```

primitive.hpp:

```
/*
  File di definizione della classe Primitive, modella una generica primitiva
     geometrica.
*/

#ifndef _PRIMITIVE_
#define _PRIMITIVE_

#include "../geometry/vec3d.hpp"
#include "../materials/material.hpp"
```

```
#include <memory>

class Ray;
class Intersection;
class Transform;

class Primitive{
public:
  Primitive(): mat_ptr(new Material()), min(), max(){}
  Primitive(Material * m): mat_ptr(m){}
  Primitive(const Primitive & p): mat_ptr(p.mat_ptr), min(p.min), max(p.max){}
  virtual ~Primitive(){} // NOTE: serve o no?

  Primitive & operator=(const Primitive & p){
    mat_ptr = p.mat_ptr; min = p.min; max = p.max;
    return *this;
  }


  void setMaterial(Material * m){mat_ptr = m;} // setta il materiale
  void setMin(const Vec3d & m){min = m;} // setta il vertice minimo del bounding
      box
  void setMax(const Vec3d & m){max = m;} // setta il vertice massimo del bounding
      box
  Material * getMaterial() const{return mat_ptr;} // ritorna il materiale
  Vec3d getMin() const{return min;} // ritorna il vertice minimo del bounding box
  Vec3d getMax() const{return max;} // ritorna il vertice massimo del bounding box

  virtual bool hit(const Ray &, Intersection & ) = 0; // metodo astratto per
      l'intersezione tra raggio e primitiva
  virtual Vec3d calcNormal(Intersection &) const = 0; // metodo astratto che
      calcola le normali
  virtual Vec3d calcTangent(Intersection &) const = 0; // metodo astratto che
      calcola le tangenti
  virtual Vec3d calcTexture(Intersection &) const = 0; // metodo astratto che
      calcola le texture
  virtual void setBVertex() = 0; // metodo astratto che calcola il bounding box
  virtual void applyTransform(Transform *) = 0; // metodo astratto che applica una
      matrice di trasformazione alla primitiva

protected:
  Material * mat_ptr; // puntatore al materiale
  Vec3d min, max; // vertici del bounding box
};

#endif
```

---

ring.cpp:

```
/*
  File di definizione della classe Ring, specializzazione della classe Disk che
      modella un anello
*/

#include "ring.hpp"
#include "../geometry/ray.hpp"
#include "../geometry/transform.hpp"
#include "../utilities/constants.hpp"
#include "../utilities/intersection.hpp"

Ring::Ring(): Disk(), r1(0.){}

Ring::Ring(const Vec3d & _p, const Vec3d & _n, double _r, double _r1, Material *
    m): Disk(_p, _n, _r, m), r1(_r1){
  if(r1 > r)
    std::swap(r1, r);
}

Ring::Ring(const Ring & r): Disk(r), r1(r.r1){}

Ring & Ring::operator=(const Ring & r){Disk::operator=(r); r1 = r.r1; return
    *this;}

bool Ring::hit(const Ray & ray, Intersection & hit){ // metodo che testa
    l'intersezione tra un raggio ray e la primitive stessa, riportando le
    informazioni ottenute nell'oggetto intersection
  double t = ((p - ray.o)*n)/(ray.d*n);
  double d = (ray(t) - p).lengthSq();
  if(t > EPS && d >= r1*r1 && d <= r*r && t < hit.t){
    hit.t = t;
    hit.hit = true;
    hit.obj_ptr = this;
    return true;
  }
  else
    return false;
}

void Ring::applyTransform(Transform * tr){ // metodo che calcola la normale a un
    punto
  Disk::applyTransform(tr);
  double s = std::max(tr->m->operator()(0,0), tr->m->operator()(1,1));
  s = std::max(s, tr->m->operator()(2,2));
  r1 *= s;
}

std::ostream & operator<<(std::ostream & os, const Ring & r){
  return os << "Ring : { " << r.p << ", " << r.n
```

```
            << "\nr1 : " << r.r << ", r2 : " << r.r1
            << "\nmin: " << r.min << ", max: " << r.max << " }";
}
```

ring.hpp:

```
#ifndef _RING_
#define _RING_

#include "disk.hpp"

class Ring: public Disk{
public:
  Ring();
  Ring(const Vec3d & _p, const Vec3d & _n, double _r, double _r1, Material * m);
  Ring(const Ring & r);

  Ring & operator=(const Ring & r);

  void setRadius1(double _r1){r1 = _r1;}
  double getRadius1() const{return r1;}

  bool hit(const Ray &, Intersection &);
  void applyTransform(Transform *);

  friend std::ostream & operator<<(std::ostream &, const Ring &);

private:
  double r1; // raggio interno dell'anello
};

std::ostream & operator<<(std::ostream &, const Ring &);

#endif
```

sphere.cpp:

```
/*
  File di definizione della classe Sphere, specializzazione della classe Primitive
      che modella una sfera
*/

#include "sphere.hpp"
#include "../geometry/ray.hpp"
#include "../geometry/transform.hpp"
#include "../utilities/constants.hpp"

Sphere::Sphere(): Primitive(), c(), r(0.){}
```

```
Sphere::Sphere(const Vec3d & _c, double _r, Material * m): Primitive(m), c(_c),
    r(_r){
  setBVertex();
}

Sphere::Sphere(const Sphere & s): Primitive(s), c(s.c), r(s.r){}

Sphere & Sphere::operator=(const Sphere & s){
  Primitive::operator=(s); c = s.c; r = s.r; min = s.min; max = s.max;
  return *this;
}

bool Sphere::hit(const Ray & ray, Intersection & hit){ // metodo che testa
    l'intersezione tra un raggio ray e la primitive stessa, riportando le
    informazioni ottenute nell'oggetto intersection
  double r2 = r*r;
  Vec3d l = c - ray.o;
  double s = l*ray.d;
  double l2 = l.lengthSq();
  if(s < 0. && l2 > r2)
    return false;
  double m2 = l2 - s*s;
  if(m2 > r2)
    return false;
  double q = sqrt(r2 - m2);
  double t = (l2 > r2) ? s - q - D_EPS : s + q + D_EPS;
  if(t > D_EPS && t < hit.t){
    hit.t = t;
    hit.hit = true;
    hit.obj_ptr = this;
    return true;
  }
  return false;
}

// bool Sphere::hit(const Ray & ray, Intersection & hit){ // metodo che testa
    l'intersezione tra un raggio ray e la primitive stessa, riportando le
    informazioni ottenute nell'oggetto intersection
//   Vec3d w = ray.o - c;
//   double beta = 2*ray.d*w;
//   double gamma = w.lengthSq() - r*r;
//   double t1 = 0, t2 = 0;
//
//   if(solveQuadratic(1.f, beta, gamma, t1, t2) != 0){
//     if(t2 > D_EPS && t2 < hit.t){
//       hit.t = t2;
//       hit.hit = true;
//       hit.obj_ptr = this;
//       return true;
```

```
//     }
//     else if(t1 > D_EPS && t1 < hit.t){
//       hit.t = t1;
//       hit.hit = true;
//       hit.obj_ptr = this;
//       return true;
//     }
//   }
//   hit.hit = false;
//   return false;
// }

Vec3d Sphere::calcTangent(Intersection & hit) const{ // metodo che calcola la
    tangente in un punto
  Vec3d up = fabs(hit.normal.z) < 0.999 ? Vec3d(0.0,0.0,1.0) : Vec3d(1,0,0);
  return (up^hit.normal).hat();
}


Vec3d Sphere::calcTexture(Intersection & hit) const{ // metodo che calcola le
    texture in un punto
  return Vec3d(.5 + atan2(hit.normal.x,hit.normal.y)*invTWO_PI
              ,.5 - asin(hit.normal.z)*invPI
              ,1.);
}

void Sphere::applyTransform(Transform * tr){ // metodo che applica una
    trasformazione alla sfera
  double s = std::max(tr->m->operator()(0,0), tr->m->operator()(1,1));
  s = std::max(s, tr->m->operator()(2,2));
  c = (tr->m->operator*(Vec4d(c,1.))).xyz();
  r *= s;
  setBVertex();
}

std::ostream & operator<<(std::ostream & os, const Sphere & sp){
  return os << "Sphere : { " << sp.c << ", " << sp.r << ", " << sp.min << ", " <<
      sp.max << " }";
}
```

sphere.hpp:

```
#ifndef _SPHERE_
#define _SPHERE_

#include "primitive.hpp"
#include "../geometry/vec3d.hpp"
#include "../utilities/intersection.hpp"
```

```
class Sphere: public Primitive{
public:
  Sphere();
  Sphere(const Vec3d &, double, Material *);
  Sphere(const Sphere &);

  Sphere & operator=(const Sphere & s);

  void setCenter(const Vec3d & _c){c = _c;}
  void setRadius(double _r){r = _r;}
  Vec3d getCenter() const{return c;}
  double getRadius() const{return r;}

  bool hit(const Ray &, Intersection &);
  Vec3d calcNormal(Intersection & hit) const{return (hit.hitPoint - c)/r;}
  Vec3d calcTangent(Intersection & hit) const;
  Vec3d calcTexture(Intersection &) const;
  void applyTransform(Transform *);
  void setBVertex(){min = c - r; max = c + r;}

  friend std::ostream & operator<<(std::ostream & os, const Sphere &);

private:
  Vec3d c; // centro della sfera
  double r; // raggio della sfera
};

std::ostream & operator<<(std::ostream & os, const Sphere &);


#endif
```

triangle.cpp:

```
/*
  File di definizione della classe Triangle, specializzazione della classe
      Primitive che modella triangolo
*/

#include "triangle.hpp"
#include "../geometry/ray.hpp"
#include "../geometry/transform.hpp"
#include "../utilities/intersection.hpp"
#include "../utilities/constants.hpp"

Triangle::Triangle(): Primitive(), p0(), p1(), p2(), n0(), n1(), n2(), t0(), t1(),
    t2(), e1(), e2(), n(){}
Triangle::Triangle(const Vec3d & _p0, const Vec3d & _p1, const Vec3d & _p2
        ,const Vec3d & _n0, const Vec3d & _n1, const Vec3d & _n2
```

```
        ,const Vec3d & _t0, const Vec3d & _t1, const Vec3d & _t2
        ,const Vec3d & _tn0, const Vec3d & _tn1, const Vec3d & _tn2
        ,Material * m)
        : Primitive(m), p0(_p0), p1(_p1), p2(_p2), n0(_n0), n1(_n1), n2(_n2),
        t0(_t0), t1(_t1), t2(_t2), tn0(_tn0), tn1(_tn1), tn2(_tn2), e1(_p1-_p0),
            e2(_p2-_p0), n(e1^e2){
          setBVertex();
        }
Triangle::Triangle(const Triangle & t): Primitive(t), p0(t.p0), p1(t.p1),
    p2(t.p2), n0(t.n0),
                                        n1(t.n1), n2(t.n2), t0(t.t0), t1(t.t1),
                                            t2(t.t2),
                                        tn0(t.tn0), tn1(t.tn1), tn2(t.tn2), e1(t.e1),
                                            e2(t.e2), n(t.n){}

Vec3d Triangle::operator[](int i){
  return (&p0)[i];
}

Triangle & Triangle::operator=(const Triangle & t){
  Primitive::operator=(t);
  p0 = t.p0; p1 = t.p1; p2 = t.p2;
  e1 = t.e1; e2 = t.e2;
  n = t.n; n0 = t.n0; n1 = t.n1; n2 = t.n2;
  t0 = t.t0; t1 = t.t1; t2 = t.t2;
  tn0 = t.t0; tn1 = t.t1; tn2 = t.t2;
  return *this;
}

bool Triangle::hit(const Ray & ray, Intersection & hit){ // metodo che testa
    l'intersezione tra un raggio ray e la primitive stessa, riportando le
    informazioni ottenute nell'oggetto intersection
  double det = -n*ray.d;
  if(det < D_EPS)
    return false;
  Vec3d s = ray.o - p0;
  Vec3d m = s^ray.d;
  double u = m*e2;
  if(u < 0. || u > det)
    return false;
  double v = -m*e1;
  if(v < 0. || u+v > det)
    return false;
  double t = n*s;
  if(t > D_EPS){
    double invDet = 1./det;
    double tf = t*invDet;
    if(tf < hit.t){
      hit.t = tf;
```

```
      hit.u = u*invDet;
      hit.v = v*invDet;
      hit.hit = true;
      hit.obj_ptr = this;
      return true;
    }
  }
  else
    return false;
}

Vec3d Triangle::calcNormal(Intersection & hit) const{ // metodo che calcola la
    normale in un punto interpolando le normali ai vertici
  return (n0*(1.-hit.u-hit.v) + n1*hit.u + n2*hit.v).hat();
}

Vec3d Triangle::calcTangent(Intersection & hit) const{ // metodo che calcola la
    tangente in un punto
  Vec3d t = (tn0*(1.-hit.u-hit.v) + tn1*hit.u + tn2*hit.v).hat();
  t = (t - hit.normal*(hit.normal*t)).hat();
  // std::cout << tn0 << " " << tn1 << " " << tn2 << "\n";
  // std::cout << t << "\n";
  return t;
}

Vec3d Triangle::calcTexture(Intersection & hit) const{ // metodo che calcola le
    texture in un punto interpolando le texture ai vertici
  return (t0*(1.-hit.u-hit.v) + t1*hit.u + t2*hit.v);
}

void Triangle::applyTransform(Transform * tr){ // metodo che applica una
    trasformazione al triangolo
  p0 = (tr->m->operator*(Vec4d(p0,1.))).xyz();
  p1 = (tr->m->operator*(Vec4d(p1,1.))).xyz();
  p2 = (tr->m->operator*(Vec4d(p2,1.))).xyz();
  n0 = (tr->mTrInv->operator*(Vec4d(n0,0.))).xyz();
  n1 = (tr->mTrInv->operator*(Vec4d(n1,0.))).xyz();
  n2 = (tr->mTrInv->operator*(Vec4d(n2,0.))).xyz();
  tn0 = (tr->mTrInv->operator*(Vec4d(tn0,0.))).xyz();
  tn1 = (tr->mTrInv->operator*(Vec4d(tn1,0.))).xyz();
  tn2 = (tr->mTrInv->operator*(Vec4d(tn2,0.))).xyz();
  e1 = p1-p0;
  e2 = p2-p0;
  n = e1^e2;
}

void Triangle::setBVertex(){ // metodo che calcola il bounding box del triangolo
  min = max = p0;
  for(int i = 0; i < 3; ++i)
```

```
    for(int j = 1; j < 3; ++j){
      double val = (*this)[j][i];
      min[i] = val < min[i] ? val : min[i];
      max[i] = val > max[i] ? val : max[i];
    }
  min -= D_EPS; //HACK: evitare che esca un bbox 2d
  max += D_EPS;
}

std::ostream & operator<<(std::ostream & os, const Triangle & t){
  return os << "Triangle : { " << t.p0 << ", " << t.p1 << ", " << t.p2
            << "\n\t\t" << t.n0 << ", " << t.n1 << ", " << t.n2 << "\n\t\t"
            << t.min << ", " << t.max << " }";
}
```

---

triangle.hpp:

```
#ifndef _TRIANGLE_
#define _TRIANGLE_

#include "primitive.hpp"

class Triangle: public Primitive{
public:
  Triangle();
  Triangle(const Vec3d & _p0, const Vec3d & _p1, const Vec3d & _p2
          ,const Vec3d & _n0, const Vec3d & _n1, const Vec3d & _n2
          ,const Vec3d & _t0, const Vec3d & _t1, const Vec3d & _t2
          ,const Vec3d & _tn0, const Vec3d & _tn1, const Vec3d & _tn2
          ,Material * m);
  Triangle(const Triangle & t);

  Vec3d operator[](int);
  Triangle & operator=(const Triangle & t);

  bool hit(const Ray &, Intersection &);
  Vec3d calcNormal(Intersection &) const;
  Vec3d calcTangent(Intersection & hit) const;
  Vec3d calcTexture(Intersection &) const;
  void applyTransform(Transform *);
  void setBVertex();

  friend std::ostream & operator<<(std::ostream &, const Triangle &);

  Vec3d p0, p1, p2; // vertici
  Vec3d n0, n1, n2; // normali ai vertici
  Vec3d t0, t1, t2; // texture ai vertici
  Vec3d tn0, tn1, tn2; // tangenti ai vertici
  Vec3d e1, e2, n; // spigoli e normale della faccia
```

```
};

std::ostream & operator<<(std::ostream &, const Triangle &);

#endif
```

GEOMETRY:

math4d.cpp:

```
/*
  File di definizione della classe Mat4d, modella una matrice quadridimensionale
      di double
*/

#include "mat4d.hpp"
#include "../utilities/math.hpp"
#include <utility>

Mat4d & Mat4d::inverse(){ // inverte la matrice con il metodo di inversione di
    gauss
  int i, j, k;
  double a[32] = {m[0], m[4], m[8], m[12], 1., 0., 0., 0.
                 ,m[1], m[5], m[9], m[13], 0., 1., 0., 0.
                 ,m[2], m[6], m[10], m[14], 0., 0., 1., 0.
                 ,m[3], m[7], m[11], m[15], 0., 0., 0., 1.};
  double d;

  /************** partial pivoting **************/
  for (i = 24; i >= 8; i-= 8){
      if (abs(a[i-8]) < abs(a[i]))
          for (j = i; j < i + 8; j++){
              std::swap(a[j],a[j-8]);
          }
  }

  /********** reducing to diagonal***********/
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      for (j = 0; j <= 24; j += 8)
          if (j != i){
              d = a[j+k] / a[i+k];
              for (int l = i,k = j; k < j + 8; k++, l++)
                  a[k] -= a[l] * d;
          }
  }

  /************** reducing to unit *************/
```

```
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      d = 1. / a[i + k];
      for (j = i; j < i + 8; j++)
          a[j] *= d;
  }

  return *this = Mat4d( a[4], a[5], a[6], a[7]
                      ,a[12], a[13], a[14], a[15]
                      ,a[20], a[21], a[22], a[23]
                      ,a[28], a[29], a[30], a[31]);
}

std::ostream & operator<<(std::ostream & os, const Mat4d & m){
  if(m.m){
    os << "M :\n";
    for(int i = 0; i < 4; ++i)
      os << "|" << m.m[i] << "\t" << m.m[i+4] << "\t" << m.m[i+8] << "\t" <<
          m.m[i+12] << "|" << std::endl;
  }
  else
  os << "empty matrix" << std::endl;
  return os;
}

Mat4d Matrix4D::transpose(const Mat4d & m){ // transpone la matrice
  Mat4d t;
  for(int i = 0; i < 16; ++i){
    div_t d = div(i,4);
    t.m[i] = m.m[d.quot + (d.rem)*4];
  }
  return t;
}

Mat4d Matrix4D::inverse(const Mat4d & m){ // inverte la matrice con il metodo di
    inversione di gauss
  int i, j, k;
  double a[32] = {m.m[0], m.m[4], m.m[8], m.m[12], 1.f, 0.f, 0.f, 0.f
                ,m.m[1], m.m[5], m.m[9], m.m[13], 0.f, 1.f, 0.f, 0.f
                ,m.m[2], m.m[6], m.m[10], m.m[14], 0.f, 0.f, 1.f, 0.f
                ,m.m[3], m.m[7], m.m[11], m.m[15], 0.f, 0.f, 0.f, 1.f};
  double d;

  /************** partial pivoting **************/
  for (i = 24; i >= 8; i-= 8){
      if (a[i-8] < a[i])
          for (j = i; j < i + 8; j++){
              std::swap(a[j],a[j-8]);
          }
  }
```

```
  /********** reducing to diagonal**********/
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      for (j = 0; j <= 24; j += 8)
          if (j != i){
              d = a[j+k] / a[i+k];
              for (int l = i,k = j; k < j + 8; k++, l++)
                  a[k] -= a[l] * d;
          }
  }

  /************** reducing to unit *************/
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      d = 1.f / a[i + k];
      for (j = i; j < i + 8; j++)
          a[j] *= d;
  }

  return Mat4d( a[4], a[5], a[6], a[7]
              ,a[12], a[13], a[14], a[15]
              ,a[20], a[21], a[22], a[23]
              ,a[28], a[29], a[30], a[31]);
}
```

---

math4d.hpp:

```
/*
  File di dichiarazione della classe Mat4d, modella una matrice quadridimensionale
      di double
*/

#ifndef _MAT_4D_
#define _MAT_4D_

#include "vec3d.hpp"
#include "vec4d.hpp"
#include <string.h>
#include <stdlib.h>

class Mat4d{
public:
  Mat4d(){memset(m, 0., sizeof(double)*16); m[0] = m[5] = m[10] = m[15] = 1.;}
  Mat4d(double s){memset(m, 0., sizeof(double)*16); m[0] = m[5] = m[10] = m[15] =
      s;}
  Mat4d(double v[16]): m(v){}
  Mat4d(double m0, double m1, double m2, double m3,
        double m4, double m5, double m6, double m7,
        double m8, double m9, double m10, double m11,
        double m12, double m13, double m14, double m15){
```

```
       m[0]  = m0; m[1] = m4; m[2] = m8; m[3] = m12;
       m[4]  = m1; m[5] = m5; m[6] = m9; m[7] = m13;
       m[8]  = m2; m[9] = m6; m[10] = m10; m[11] = m14;
       m[12] = m3; m[13] = m7; m[14] = m11; m[15] = m15;
     }
Mat4d(const Vec3d & u, const Vec3d & v, const Vec3d & z, const Vec3d & w){
  m[0]  = u.x; m[1] = u.y; m[2] = u.z; m[3] = 0.;
  m[4]  = v.x; m[5] = v.y; m[6] = v.z; m[7] = 0.;
  m[8]  = z.x; m[9] = z.y; m[10] = z.z; m[11] = 0.;
  m[12] = w.x; m[13] = w.y; m[14] = w.z; m[15] = 1.;
}
Mat4d(const Vec4d & u, const Vec4d & v, const Vec4d & z, const Vec4d & w){
  m[0]  = u.x; m[1] = u.y; m[2] = u.z; m[3] = u.w;
  m[4]  = v.x; m[5] = v.y; m[6] = v.z; m[7] = v.w;
  m[8]  = z.x; m[9] = z.y; m[10] = z.z; m[11] = z.w;
  m[12] = w.x; m[13] = w.y; m[14] = w.z; m[15] = w.w;
}
Mat4d(const Mat4d & n){memcpy(m, n.m, sizeof(double)*16);}
Mat4d(Mat4d && n){delete [] m; m = n.m; n.m = nullptr;}
~Mat4d(){delete [] m;}

Mat4d operator*(const Mat4d & n) const{ // moltiplicazione tra matrici
  Mat4d t;
  for(int i = 0; i < 4; ++i)
    for(int j = 0; j < 4; ++j)
      t(j,i) = n.m[4*i]*m[j] + n.m[4*i+1]*m[j+4] + n.m[4*i+2]*m[j+8] +
          n.m[4*i+3]*m[j+12];
  return t;
}
// Mat4d operator*(const Mat4d & n) const{
//   Mat4d t;
//   for(int i = 0, j = 0, k = 0; i < 16; ++i, j = i%4, k = (i/4)*4)
//     t.m[i] = n.m[0+k]*m[0+j] + n.m[1+k]*m[4+j] + n.m[2+k]*m[8+j] +
    n.m[3+k]*m[12+j];
//   return t;
// }
Mat4d & operator*=(const Mat4d & n){
  Mat4d t;
  for(int i = 0; i < 4; ++i)
    for(int j = 0; j < 4; ++j)
      t(j,i) = n.m[4*i]*m[j] + n.m[4*i+1]*m[j+4] + n.m[4*i+2]*m[j+8] +
          n.m[4*i+3]*m[j+12];
  return *this = t;
}
Vec4d operator*(const Vec4d & v) const{ // moltiplicazione matrice vettore
  Vec4d t(m[0]*v.x + m[4]*v.y + m[8]*v.z + m[12]*v.w,
          m[1]*v.x + m[5]*v.y + m[9]*v.z + m[13]*v.w,
          m[2]*v.x + m[6]*v.y + m[10]*v.z + m[14]*v.w,
          m[3]*v.x + m[7]*v.y + m[11]*v.z + m[15]*v.w);
```

```
    return t;
  }
  Mat4d & operator=(const Mat4d & n){memcpy(m, n.m, sizeof(double)*16); return
      *this;}
  Mat4d & operator=(Mat4d && n){delete [] m; m = n.m; n.m = nullptr; return *this;}
  double operator()(int r, int c) const{return m[c*4 + r];} // ritorna l'elemento
      di riga r e colonna c
  double & operator()(int r, int c){return m[c*4 + r];}

  Vec4d row(int i) const{return Vec4d(m[i], m[i+4], m[i+8], m[i+12]);} // ritorna
      la riga i
  Vec4d col(int i) const{return Vec4d(m[i*4], m[i*4+1], m[i*4+2], m[i*4+3]);} //
      ritorna la colonna i
  Mat4d & transpose(){ // transpone la matrice
    Mat4d t;
    for(int i = 0; i < 16; ++i){
      div_t d = div(i,4);
      t.m[i] = m[d.quot + (d.rem)*4];
    }
    return *this = t;
  }
  Mat4d & inverse();

  double * m = new double[16];
};

std::ostream & operator<<(std::ostream &, const Mat4d &);

namespace Matrix4D{
  Mat4d transpose(const Mat4d &);
  Mat4d inverse(const Mat4d &);
}

#endif
```

---

math4f.cpp:

```
/*
  File definizione della classe Mat4f, modella una matrice quadridimensionale di
      float
*/

#include "mat4f.hpp"
#include "../utilities/math.hpp"
#include <utility>

Mat4f & Mat4f::inverse(){ // inverte la matrice con il metodo di inversione di
    gauss
  int i, j, k;
```

```
  float a[32] = {m[0], m[4], m[8], m[12], 1.f, 0.f, 0.f, 0.f
                ,m[1], m[5], m[9], m[13], 0.f, 1.f, 0.f, 0.f
                ,m[2], m[6], m[10], m[14], 0.f, 0.f, 1.f, 0.f
                ,m[3], m[7], m[11], m[15], 0.f, 0.f, 0.f, 1.f};
  float d;

  /************** partial pivoting **************/
  for (i = 24; i >= 8; i-= 8){
      if (a[i-8] < a[i])
          for (j = i; j < i + 8; j++){
              std::swap(a[j],a[j-8]);
          }
  }

  /********** reducing to diagonal***********/
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      for (j = 0; j <= 24; j += 8)
          if (j != i){
              d = a[j+k] / a[i+k];
              for (int l = i,k = j; k < j + 8; k++, l++)
                  a[k] -= a[l] * d;
          }
  }

  /************** reducing to unit *************/
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      d = 1.f / a[i + k];
      for (j = i; j < i + 8; j++)
          a[j] *= d;
  }

  return *this = Mat4f( a[4], a[5], a[6], a[7]
                      ,a[12], a[13], a[14], a[15]
                      ,a[20], a[21], a[22], a[23]
                      ,a[28], a[29], a[30], a[31]);
}

std::ostream & operator<<(std::ostream & os, const Mat4f & m){
  if(m.m){
    os << "M :\n";
    for(int i = 0; i < 4; ++i)
      os << "|" << m.m[i] << "\t" << m.m[i+4] << "\t" << m.m[i+8] << "\t" <<
          m.m[i+12] << "|" << std::endl;
  }
  else
  os << "empty matrix" << std::endl;
  return os;
}
```

```
Mat4f Mat4F::transpose(const Mat4f & m){ // transpone la matrice
  Mat4f t;
  for(int i = 0; i < 16; ++i){
    div_t d = div(i,4);
    t.m[i] = m.m[d.quot + (d.rem)*4];
  }
  return t;
}

Mat4f Mat4F::inverse(const Mat4f & m){ // inverte la matrice con il metodo di
    inversione di gauss
  int i, j, k;
  float a[32] = {m.m[0], m.m[4], m.m[8], m.m[12], 1.f, 0.f, 0.f, 0.f
              ,m.m[1], m.m[5], m.m[9], m.m[13], 0.f, 1.f, 0.f, 0.f
              ,m.m[2], m.m[6], m.m[10], m.m[14], 0.f, 0.f, 1.f, 0.f
              ,m.m[3], m.m[7], m.m[11], m.m[15], 0.f, 0.f, 0.f, 1.f};
  float d;

  /************** partial pivoting **************/
  for (i = 24; i >= 8; i-= 8){
      if (a[i-8] < a[i])
          for (j = i; j < i + 8; j++){
              std::swap(a[j],a[j-8]);
          }
  }

  /********** reducing to diagonal***********/
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      for (j = 0; j <= 24; j += 8)
          if (j != i){
              d = a[j+k] / a[i+k];
              for (int l = i,k = j; k < j + 8; k++, l++)
                  a[k] -= a[l] * d;
          }
  }

  /************** reducing to unit *************/
  for (k = 0, i = 0; i <= 24; i += 8, k++){
      d = 1.f / a[i + k];
      for (j = i; j < i + 8; j++)
          a[j] *= d;
  }

  return Mat4f( a[4], a[5], a[6], a[7]
              ,a[12], a[13], a[14], a[15]
              ,a[20], a[21], a[22], a[23]
              ,a[28], a[29], a[30], a[31]);
}
```

```
Mat4f Mat4F::translation(float a, float b, float c){ // crea una matrice di
    traslazione
  return Mat4f(1.f, 0.f, 0.f, a
              ,0.f, 1.f, 0.f, b
              ,0.f, 0.f, 1.f, c
              ,0.f, 0.f, 0.f, 1.f);
}

Mat4f Mat4F::scaling(float s){ // crea una matrice di scaling
  return Mat4f( s, 0.f, 0.f, 0.f
              ,0.f,  s, 0.f, 0.f
              ,0.f, 0.f,  s, 0.f
              ,0.f, 0.f, 0.f, 1.f);
}

Mat4f Mat4F::scaling(float a, float b, float c){ // crea una matrice di scaling
  return Mat4f( a, 0.f, 0.f, 0.f
              ,0.f,  b, 0.f, 0.f
              ,0.f, 0.f,  c, 0.f
              ,0.f, 0.f, 0.f, 1.f);
}

Mat4f Mat4F::rotationX(float deg){ // crea una matrice di rotazione sull'asse X
  float sinT = sin(toRad(deg));
  float cosT = cos(toRad(deg));
  return Mat4f(1.f, 0.f,  0.f,  0.f
              ,0.f, cosT, -sinT, 0.f
              ,0.f, sinT, cosT, 0.f
              ,0.f, 0.f,  0.f, 1.f);

}

Mat4f Mat4F::rotationY(float deg){ // crea una matrice di rotazione sull'asse Y
  float sinT = sin(toRad(deg));
  float cosT = cos(toRad(deg));
  return Mat4f( cosT, 0.f, sinT, 0.f
              ,  0.f, 1.f, 0.f, 0.f
              ,-sinT, 0.f, cosT, 0.f
              ,  0.f, 0.f, 0.f, 1.f);

}

Mat4f Mat4F::rotationZ(float deg){ // crea una matrice di rotazione sull'asse Z
  float sinT = sin(toRad(deg));
  float cosT = cos(toRad(deg));
  return Mat4f(cosT, -sinT, 0.f, 0.f
              ,cosT, sinT, 0.f, 0.f
              , 0.f,  0.f, 1.f, 0.f
              , 0.f,  0.f, 0.f, 1.f);
```

```
}
```

math4f.hpp:

```
/*
  File di dichiarazione della classe Mat4f, modella una matrice quadridimensionale
     di float
*/

#ifndef _MAT_4F_
#define _MAT_4F_

#include "vec4f.hpp"
#include <string.h>
#include <stdlib.h>

class Mat4f{
public:
  Mat4f(){memset(m, 0.f, sizeof(float)*16); m[0] = m[5] = m[10] = m[15] = 1.f;}
  Mat4f(float s){memset(m, 0.f, sizeof(float)*16); m[0] = m[5] = m[10] = m[15] =
     s;}
  Mat4f(float v[16]): m(v){}
  Mat4f(float m0, float m1, float m2, float m3,
        float m4, float m5, float m6, float m7,
        float m8, float m9, float m10, float m11,
        float m12, float m13, float m14, float m15){
          m[0]  = m0; m[1] = m4; m[2] = m8; m[3] = m12;
          m[4]  = m1; m[5] = m5; m[6] = m9; m[7] = m13;
          m[8]  = m2; m[9] = m6; m[10] = m10; m[11] = m14;
          m[12] = m3; m[13] = m7; m[14] = m11; m[15] = m15;
        }
  Mat4f(const Vec3f & u, const Vec3f & v, const Vec3f & z, const Vec3f & w){
    m[0]  = u.x; m[1] = u.y; m[2] = u.z; m[3] = 0.f;
    m[4]  = v.x; m[5] = v.y; m[6] = v.z; m[7] = 0.f;
    m[8]  = z.x; m[9] = z.y; m[10] = z.z; m[11] = 0.f;
    m[12] = w.x; m[13] = w.y; m[14] = w.z; m[15] = 1.f;
  }
  Mat4f(const Vec4f & u, const Vec4f & v, const Vec4f & z, const Vec4f & w){
    m[0]  = u.x; m[1] = u.y; m[2] = u.z; m[3] = u.w;
    m[4]  = v.x; m[5] = v.y; m[6] = v.z; m[7] = v.w;
    m[8]  = z.x; m[9] = z.y; m[10] = z.z; m[11] = z.w;
    m[12] = w.x; m[13] = w.y; m[14] = w.z; m[15] = w.w;
  }
  Mat4f(const Mat4f & n){memcpy(m, n.m, sizeof(float)*16);}
  Mat4f(Mat4f && n){delete [] m; m = n.m; n.m = nullptr;}
  ~Mat4f(){delete [] m;}

  Mat4f operator*(const Mat4f & n) const{ // moltiplicazione tra matrici
```

```
    Mat4f t;
    for(int i = 0; i < 4; ++i)
      for(int j = 0; j < 4; ++j)
        t(j,i) = n.m[4*i]*m[j] + n.m[4*i+1]*m[j+4] + n.m[4*i+2]*m[j+8] +
            n.m[4*i+3]*m[j+12];
    return t;
  }
  Mat4f & operator*=(const Mat4f & n){
    Mat4f t;
    for(int i = 0; i < 4; ++i)
      for(int j = 0; j < 4; ++j)
        t(j,i) = n.m[4*i]*m[j] + n.m[4*i+1]*m[j+4] + n.m[4*i+2]*m[j+8] +
            n.m[4*i+3]*m[j+12];
    return *this = t;
  }
  Vec4f operator*(const Vec4f & v) const{ // moltiplicazione matrice vettore
    Vec4f t(m[0]*v.x + m[4]*v.y + m[8]*v.z + m[12]*v.w,
            m[1]*v.x + m[5]*v.y + m[9]*v.z + m[13]*v.w,
            m[2]*v.x + m[6]*v.y + m[10]*v.z + m[14]*v.w,
            m[3]*v.x + m[7]*v.y + m[11]*v.z + m[15]*v.w);
    return t;
  }
  Mat4f & operator=(const Mat4f & n){memcpy(m, n.m, sizeof(float)*16); return
      *this;}
  Mat4f & operator=(Mat4f && n){delete [] m; m = n.m; n.m = nullptr; return *this;}
  float operator()(int r, int c) const{return m[c*4 + r];} // ritorna l'elemento
      di riga r e colonna c
  float & operator()(int r, int c){return m[c*4 + r];}

  Vec4f row(int i) const{return Vec4f(m[i], m[i+4], m[i+8], m[i+12]);} // ritorna
      la riga i
  Vec4f col(int i) const{return Vec4f(m[i*4], m[i*4+1], m[i*4+2], m[i*4+3]);} //
      ritorna la colonna i
  Mat4f & transpose(){ // transpone la matrice
    Mat4f t;
    for(int i = 0; i < 16; ++i){
      div_t d = div(i,4);
      t.m[i] = m[d.quot + (d.rem)*4];
    }
    return *this = t;
  }
  Mat4f & inverse();

  float * m = new float[16];
};

std::ostream & operator<<(std::ostream &, const Mat4f &);

namespace Mat4F{
```

```
  Mat4f transpose(const Mat4f &);
  Mat4f inverse(const Mat4f &);
  Mat4f translation(float, float, float);
  Mat4f scaling(float);
  Mat4f scaling(float, float, float);
  Mat4f rotationX(float);
  Mat4f rotationY(float);
  Mat4f rotationZ(float);
}

#endif
```

---

ray.hpp:

```
/*
  File di dichiarazione della classe Ray, modella una raggio
*/

#ifndef _RAY_
#define _RAY_

#include "vec3d.hpp"
#include "../utilities/constants.hpp"

class Ray{
public:
  Ray(): o(), d(), dInv(), s(){}
  Ray(const Vec3d & _o, const Vec3d & _d): o(_o), d(_d.hat()), dInv(1./d.x,
      1./d.y, 1./d.z){
    s[0] = dInv.x < D_EPS; s[1] = dInv.y < D_EPS; s[2] = dInv.z < D_EPS;
  }
  Ray(const Ray & r): o(r.o), d(r.d), dInv(r.dInv), s{r.s[0],r.s[1],r.s[2]}{}

  Ray & operator=(const Ray & r){
    o = r.o; d = r.d; dInv = r.dInv;
    s[0] = r.s[0]; s[1] = r.s[1]; s[2] = r.s[2];
    return *this;
  }
  Vec3d operator()(double t) const{return o + d*t;} // calcola il punto dato t

  void refresh(){ // aggiorna i valori di dInv e s
    dInv = Vec3d(1./d.x, 1./d.y, 1./d.z);
    s[0] = dInv.x < D_EPS; s[1] = dInv.y < D_EPS; s[2] = dInv.z < D_EPS;
  }

  Vec3d o, d, dInv; // origine, direzione, e reciproco della direzione
  bool s[3]; // indica il segno delle componenti della direzione
};
```

```
inline std::ostream & operator<<(std::ostream & os, const Ray & r){
  return os << "R : { " << r.o << ", " << r.d << ", " << r.dInv
            << ", { " << r.s[0] << ", " << r.s[1] << ", " << r.s[2] << "} }";
}

#endif
```

---

transform.cpp:

```
/*
  File di dichiarazione della classe Transform, modella una trasformazione
     geometrica nello spazio
*/

#include "transform.hpp"
#include "../utilities/math.hpp"

Transform::Transform(): m(new Mat4d()), mInv(new Mat4d()), mTrInv(new Mat4d()){}

Transform::Transform(const Vec3d & x, const Vec3d & y, const Vec3d & z, const
    Vec3d & w): m(new Mat4d(x, y, z, w)){
  mInv = new Mat4d(x, y, z, w); mInv->inverse();
  mTrInv = new Mat4d(*mInv); mTrInv->transpose();
}

Transform::Transform(const Vec4d & x, const Vec4d & y, const Vec4d & z, const
    Vec4d & w): m(new Mat4d(x, y, z, w)){
  mInv = new Mat4d(x, y, z, w); mInv->inverse();
  mTrInv = new Mat4d(*mInv); mTrInv->transpose();
}

Transform::Transform(Mat4d * _m): m(_m), mInv(new Mat4d(*_m)){
  mInv->inverse();
  mTrInv = new Mat4d(*mInv); mTrInv->transpose();
}

Transform::Transform(Mat4d * _m, Mat4d * _mInv, Mat4d * _mTrInv): m(_m),
    mInv(_mInv), mTrInv(_mTrInv){}

Transform::Transform(const Transform & tr){
  m = new Mat4d(*tr.m); mInv = new Mat4d(*tr.mInv); mTrInv = new Mat4d(*tr.mTrInv);
}

Transform::~Transform(){
  delete m; delete mInv; delete mTrInv;
}

Transform Transform::operator*(const Transform & tr) const{ // composizione di
    trasformazioni
```

```
  return Transform(new Mat4d(m->operator*(*tr.m))
                  ,new Mat4d(mInv->operator*(*tr.mInv))
                  ,new Mat4d(mTrInv->operator*(*tr.mTrInv)));
}

Transform & Transform::operator*=(const Transform & tr){
  m->operator*=(*tr.m); mInv->operator*=(*tr.mInv); mTrInv->operator*=(*tr.mTrInv);
  return *this;
}

Transform & Transform::operator=(const Transform & tr){
  *m = *tr.m; *mInv = *tr.mInv; *mTrInv = *tr.mTrInv;
  return *this;
}

void Transform::inverse(){ // inverte la trasformazione
  std::swap(m,mInv);
  *mTrInv = *m; mTrInv->transpose();
}

Transform translation(double a, double b, double c){ // crea una traslazione
  return Transform(new Mat4d(1., 0., 0., a
                            ,0., 1., 0., b
                            ,0., 0., 1., c
                            ,0., 0., 0., 1.)
                  ,new Mat4d(1., 0., 0., -a
                            ,0., 1., 0., -b
                            ,0., 0., 1., -c
                            ,0., 0., 0., 1.)
                  ,new Mat4d(1., 0., 0., 0.
                            ,0., 1., 0., 0.
                            ,0., 0., 1., 0.
                            ,0., 0., 0., 1.)
                  );
}

Transform scaling(double s){ // crea uno scaling
  double d = 1./s;
  return Transform(new Mat4d( s, 0., 0., 0.
                            ,0.,  s, 0., 0.
                            ,0., 0., s, 0.
                            ,0., 0., 0., 1.)
                  ,new Mat4d( d, 0., 0., 0.
                            ,0.,  d, 0., 0.
                            ,0., 0., d, 0.
                            ,0., 0., 0., 1.)
                  ,new Mat4d(1., 0., 0., 0.
                            ,0., 1., 0., 0.
                            ,0., 0., 1., 0.
```

```
                        ,0., 0., 0., 1.)
                );
}

Transform scaling(double a, double b, double c){ // crea uno scaling
  return Transform(new Mat4d( a, 0., 0., 0.
                        ,0., b, 0., 0.
                        ,0., 0., c, 0.
                        ,0., 0., 0., 1.)
                ,new Mat4d(1./a, 0.,  0., 0.
                        , 0., 1./b,  0., 0.
                        , 0.,  0., 1./c, 0.
                        , 0.,  0.,  0., 1.)
                ,new Mat4d(1./a, 0.,  0., 0.
                        , 0., 1./b,  0., 0.
                        , 0.,  0., 1./c, 0.
                        , 0.,  0.,  0., 1.)
                );
}

Transform rotationX(double deg){ // crea una rotazione intorno all'asse X
  double sinT = sin(toRad(deg));
  double cosT = cos(toRad(deg));
  return Transform(new Mat4d(1., 0.,  0., 0.
                        ,0., cosT, -sinT, 0.
                        ,0., sinT, cosT, 0.
                        ,0.,  0.,   0., 1.)
                ,new Mat4d(1., 0.,   0., 0.
                        ,0., cosT, sinT, 0.
                        ,0., -sinT, cosT, 0.
                        ,0.,   0.,  0., 1.)
                ,new Mat4d(1., 0.,   0., 0.
                        ,0., cosT, -sinT, 0.
                        ,0., sinT, cosT, 0.
                        ,0.,  0.,   0., 1.)
                );
}

Transform rotationY(double deg){ // crea una rotazione intorno all'asse Y
  double sinT = sin(toRad(deg));
  double cosT = cos(toRad(deg));
  return Transform(new Mat4d( cosT, 0., sinT, 0.
                        ,  0., 1.,  0., 0.
                        ,-sinT, 0., cosT, 0.
                        ,  0., 0.,  0., 1.)
                ,new Mat4d(cosT, 0., -sinT, 0.
                        , 0., 1.,   0., 0.
                        ,sinT, 0., cosT, 0.
                        , 0., 0.,   0., 1.)
```

```
                ,new Mat4d( cosT, 0., sinT, 0.
                          ,   0., 1.,  0., 0.
                          ,-sinT, 0., cosT, 0.
                          ,   0., 0.,  0., 1.)
                );
}

Transform rotationZ(double deg){ // crea una rotazione intorno all'asse Z
  double sinT = sin(toRad(deg));
  double cosT = cos(toRad(deg));
  return Transform(new Mat4d(cosT, -sinT, 0., 0.
                          ,sinT, cosT, 0., 0.
                          , 0.,   0., 1., 0.
                          , 0.,   0., 0., 1.)
                ,new Mat4d( cosT, sinT, 0., 0.
                          ,-sinT, cosT, 0., 0.
                          ,   0.,  0., 1., 0.
                          ,   0.,  0., 0., 1.)
                ,new Mat4d(cosT, -sinT, 0., 0.
                          ,sinT, cosT, 0., 0.
                          , 0.,   0., 1., 0.
                          , 0.,   0., 0., 1.)
                );
}
```

---

transform.hpp:

```
#ifndef _TRANSFORM_
#define _TRANSFORM_

#include "mat4d.hpp"

class Vec3d;

class Transform{
public:
  Transform();
  Transform(const Vec3d &, const Vec3d &, const Vec3d &, const Vec3d &);
  Transform(const Vec4d &, const Vec4d &, const Vec4d &, const Vec4d &);
  Transform(Mat4d *);
  Transform(Mat4d *, Mat4d *, Mat4d *);
  Transform(const Transform &);
  ~Transform();

  Transform operator*(const Transform &) const;
  Transform & operator*=(const Transform &);
  Transform & operator=(const Transform &);

  void inverse();
```

```
  Mat4d * m, * mInv, * mTrInv; // matrice, inversa e inversa trasposta
};

Transform translation(double, double, double);
Transform scaling(double);
Transform scaling(double, double, double);
Transform rotationX(double);
Transform rotationY(double);
Transform rotationZ(double);



#endif
```

vect3d.hpp:

```
/*
  File di dichiarazione della classe Vec3d, modella un vettore tridimensionale di
     double
*/

#ifndef _VEC_3D_
#define _VEC_3D_

#include <iostream>
#include <math.h>
#include <assimp/vector3.h>

class Vec3d{
public:
  Vec3d(): x(), y(), z(){}
  Vec3d(double s): x(s), y(s), z(s){}
  Vec3d(double _x, double _y, double _z): x(_x), y(_y), z(_z){}
  Vec3d(const Vec3d & v): x(v.x), y(v.y), z(v.z){}
  Vec3d(const aiVector3D & v): x(v.x), y(v.y), z(v.z){}

  Vec3d operator+(const Vec3d & v) const{return Vec3d(x+v.x, y+v.y, z+v.z);} //
     somma tra vettori
  Vec3d & operator+=(const Vec3d & v){x += v.x; y += v.y; z += v.z; return *this;}
  Vec3d operator-(const Vec3d & v) const{return Vec3d(x-v.x, y-v.y, z-v.z);} //
     differenza tra vettori
  Vec3d & operator-=(const Vec3d & v){x -= v.x; y -= v.y; z -= v.z; return *this;}
  Vec3d operator*(double s) const{return Vec3d(x*s, y*s, z*s);} // prodotto per
     scalare
  Vec3d & operator*=(double s){x *= s; y *= s; z *= s; return *this;}
  double operator*(const Vec3d & v) const{return x*v.x + y*v.y + z*v.z;} //
     prodotto scalare
```

```
  Vec3d operator^(const Vec3d & v) const{return Vec3d(y*v.z - z*v.y, z*v.x -
      x*v.z, x*v.y - y*v.x);} // prodotto vettoriale
  Vec3d operator%(const Vec3d & v) const{return Vec3d(x*v.x, y*v.y, z*v.z);} //
      prodotto componente per componente
  Vec3d operator/(double s) const{double d = 1./s; return Vec3d(x*d, y*d, z*d);}
      // divisione per scalare
  Vec3d & operator/=(double s){double d = 1./s; x *= d; y *= d; z *= d; return
      *this;}
  Vec3d operator-() const{return Vec3d(-x, -y, -z);} // moltiplicazione per -1
  bool operator==(const Vec3d & v) const{return x == v.x && y == v.y && z == v.z;}
  bool operator!=(const Vec3d & v) const{return x != v.x || y != v.y || z != v.z;}
  bool operator>(const Vec3d & v) const{return x > v.x && y > v.y && z > v.z;}
  bool operator>=(const Vec3d & v) const{return x >= v.x && y >= v.y && z >= v.z;}
  bool operator<(const Vec3d & v) const{return x < v.x && y < v.y && z < v.z;}
  bool operator<=(const Vec3d & v) const{return x <= v.x && y <= v.y && z <= v.z;}
  Vec3d & operator=(const Vec3d & v){x = v.x; y = v.y; z = v.z; return *this;}
  Vec3d & operator=(const aiVector3D & v){x = v.x; y = v.y; z = v.z; return *this;}
  double operator[](int i) const{return (&x)[i];}
  double & operator[](int i) {return (&x)[i];}

  double lengthSq() const{return x*x + y*y + z*z;} // norma quadra
  double length() const{return sqrt(lengthSq());} // norma
  Vec3d hat() const{return (*this)/length();} // ritorna il vettore normalizzato
  Vec3d & normalize(){return (*this)/=length();} // normalizza il vettore
  Vec3d reflect(const Vec3d & n) const{return *this - n*((*this)*n)*2.;} //
      riflette il vettore rispetto alla normale n
  Vec3d & reflect(const Vec3d & n){return *this -= n*((*this)*n)*2.;}

  std::string toString() const{
    return "{ "+std::to_string(x)+", "+std::to_string(y)+", "+std::to_string(z)+"}";
  }

  double x, y, z;
};

inline Vec3d operator*(double s, const Vec3d & v){return v*s;}
inline std::ostream & operator<<(std::ostream & os, const Vec3d & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << "}";
}

#endif
```

---

vect3f.hpp:

```
/*
  File di dichiarazione della classe Vec3f, modella un vettore tridimensionale di
      float
*/
```

```
#ifndef _VEC_3F_
#define _VEC_3F_

#include <iostream>
#include <math.h>

class Vec3f{
public:
  Vec3f(): x(), y(), z(){}
  Vec3f(float s): x(s), y(s), z(s){}
  Vec3f(float _x, float _y, float _z): x(_x), y(_y), z(_z){}
  Vec3f(const Vec3f & v): x(v.x), y(v.y), z(v.z){}

  Vec3f operator+(const Vec3f & v) const{return Vec3f(x+v.x, y+v.y, z+v.z);} //
     somma tra vettori
  Vec3f & operator+=(const Vec3f & v){x += v.x; y += v.y; z += v.z; return *this;}
  Vec3f operator-(const Vec3f & v) const{return Vec3f(x-v.x, y-v.y, z-v.z);} //
     differenza tra vettori
  Vec3f & operator-=(const Vec3f & v){x -= v.x; y -= v.y; z -= v.z; return *this;}
  Vec3f operator*(float s) const{return Vec3f(x*s, y*s, z*s);} // prodotto per
     scalare
  Vec3f & operator*=(float s){x *= s; y *= s; z *= s; return *this;}
  float operator*(const Vec3f & v) const{return x*v.x + y*v.y + z*v.z;} //
     prodotto scalare
  Vec3f operator^(const Vec3f & v) const{return Vec3f(y*v.z - z*v.y, z*v.x -
     x*v.z, x*v.y - y*v.x);} // prodotto vettoriale
  Vec3f operator%(const Vec3f & v) const{return Vec3f(x*v.x, y*v.y, z*v.z);} //
     prodotto componente per componente
  Vec3f operator/(float s) const{float d = 1.f/s; return Vec3f(x*d, y*d, z*d);} //
     divisione per scalare
  Vec3f & operator/=(float s){float d = 1.f/s; x *= d; y *= d; z *= d; return
     *this;}
  Vec3f operator-() const{return Vec3f(-x, -y, -z);} // moltiplicazione per -1
  bool operator==(const Vec3f & v) const{return x == v.x && y == v.y && z == v.z;}
  bool operator!=(const Vec3f & v) const{return x != v.x || y != v.y || z != v.z;}
  bool operator>(const Vec3f & v){return x > v.x && y > v.y && z > v.z;}
  bool operator<(const Vec3f & v){return x < v.x && y < v.y && z < v.z;}
  Vec3f & operator=(const Vec3f & v){x = v.x; y = v.y; z = v.z; return *this;}
  float operator[](int i) const{return (&x)[i];}
  float & operator[](int i) {return (&x)[i];}

  float lengthSq() const{return x*x + y*y + z*z;} // norma quadra
  float length() const{return sqrtf(lengthSq());} // norma
  Vec3f hat() const{return (*this)/length();} // ritorna il vettore normalizzato
  Vec3f & normalize(){return (*this)/=length();} // normalizza il vettore
  Vec3f reflect(const Vec3f & n) const{return *this - n*((*this)*n)*2.f;} //
     riflette il vettore rispetto alla normale
  Vec3f & reflect(const Vec3f & n){return *this -= n*((*this)*n)*2.f;}
```

```
  std::string toString() const{
    return "{ "+std::to_string(x)+", "+std::to_string(y)+", "+std::to_string(z)+"}";
  }

  float x, y, z;
};

inline Vec3f operator*(float s, const Vec3f & v){return v*s;}
inline std::ostream & operator<<(std::ostream & os, const Vec3f & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << "}";
}

#endif
```

vect4d.hpp:

```
/*
  File di dichiarazione della classe Vec4d, modella un vettore quadridimensionale
      di double
*/

#ifndef _VEC_4D_
#define _VEC_4D_

#include "vec3d.hpp"
#include <assimp/vector3.h>

class Vec4d{
public:
  Vec4d(): x(), y(), z(), w(){}
  Vec4d(double s, double _w): x(s), y(s), z(s), w(_w){}
  Vec4d(double _x, double _y, double _z, double _w): x(_x), y(_y), z(_z), w(_w){}
  Vec4d(const Vec3d & v, double _w): x(v.x), y(v.y), z(v.z), w(_w){}
  Vec4d(const aiVector3D & v, double _w): x(v.x), y(v.y), z(v.z), w(_w){}
  Vec4d(const Vec4d & v): x(v.x), y(v.y), z(v.z), w(v.w){}

  Vec4d operator/(double s){double d = 1./s; return Vec4d(x*d, y*d, z*d, w*d);} //
      divisione per scalare
  Vec4d & operator/=(double s){double d = 1./s; x *= d; y *= d; z *= d; w *= d;
      return *this;}

  Vec3d xyz() const{return Vec3d(x,y,z);} // ritorna il vettore tridimensionale

  double x, y, z, w;
};

inline std::ostream & operator<<(std::ostream & os, const Vec4d & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << ", " << v.w << "}";
}
```

```
#endif
```

vect4f.hpp:

```
/*
  File di dichiarazione della classe Vec4f, modella un vettore quadridimensionale
     di float
*/

#ifndef _VEC_4F_
#define _VEC_4F_

#include "vec3f.hpp"

class Vec4f{
public:
  Vec4f(): x(), y(), z(), w(){}
  Vec4f(float s, float _w): x(s), y(s), z(s), w(_w){}
  Vec4f(float _x, float _y, float _z, float _w): x(_x), y(_y), z(_z), w(_w){}
  Vec4f(const Vec3f & v, float _w): x(v.x), y(v.y), z(v.z), w(_w){}
  Vec4f(const Vec4f & v): x(v.x), y(v.y), z(v.z), w(v.w){}

  Vec4f operator/(float s){float d = 1.f/s; return Vec4f(x*d, y*d, z*d, w*d);} //
     divisione per scalare
  Vec4f & operator/=(float s){float d = 1.f/s; x *= d; y *= d; z *= d; w *= d;
     return *this;}

  Vec3f xyz() const{return Vec3f(x,y,z);} // ritorna il vettore tridimensionale

  float x, y, z, w;
};

inline std::ostream & operator<<(std::ostream & os, const Vec4f & v){
  return os << "{" << v.x << ", " << v.y << ", " << v.z << ", " << v.w << "}";
}

#endif
```

INTEGRATORS:

integrator.hpp:

```
/*
  File di definizione della classe Integrator, modella un integratore generico per
     la radianza entrante in un pixel
*/
```

```
#ifndef _INTEGRATOR_
#define _INTEGRATOR_

#include "../geometry/vec3d.hpp"
#include "../geometry/ray.hpp"
#include "../lights/light.hpp"
#include "../utilities/environment.hpp"
#include "../utilities/stats.hpp"
#include "../world.hpp"

class Integrator{
public:
  Integrator(World * _w): w(_w){}
  virtual ~Integrator(){}

  virtual Vec3d solve(const Ray &) const = 0; // metodo astratto che risolve
      l'integrale della radianza

  virtual std::string toString() const = 0;

  World * w;
};


#endif
```

pathtracing.hpp:

```
/*
  File di definizione della classe PathTracingIntegrator, specializzazione della
      classe Integrator che modella l'integratore di pathtracing,
  ovvero calcola l'integrale della radianza completo, tramite l'estimatore di
      montecarlo. Tratta solo luci non ad area
*/

#ifndef _PATHTRACING_INTEGRATOR_
#define _PATHTRACING_INTEGRATOR_

#include "integrator.hpp"
#include "../samplers/sampler.hpp"

extern int px, py;

class PathTracingIntegrator : public Integrator{
public:
  PathTracingIntegrator(World * w): Integrator(w){}
  PathTracingIntegrator(World * w, int d): Integrator(w), depth(d){}
```

```
Vec3d solve(const Ray & ray) const{ // metodo che risolve l'integrale della
    radianza
  return traceRay(ray, depth);
}

Vec3d traceRay(const Ray & ray, int _depth) const{ // metodo che calcola la
    radianza ricorsiva dato un raggio e il numero di rimbalzi
  Vec3d bias;
  Intersection hit;
  Material * mat;
  Vec3d t;
  hit = w->intersectTree(ray); // trovo l'intersezione traversando l'albero
  w->stats->hits += hit.nHits;
  if(hit.hit){ // se ho colpito
    hit.hitPoint = ray(hit.t); // calcolo il punto colpito
    hit.normal = hit.obj_ptr->calcNormal(hit); // calcolo la normale al punto
    t = hit.obj_ptr->calcTangent(hit); // calcolo la tangente al punto
    hit.texture = hit.obj_ptr->calcTexture(hit); // calcolo le texture del punto
    mat = hit.obj_ptr->getMaterial(); // mi salvo un riferimento al materiale
        della primitiva colpita
    hit.TBN = new Mat4d(t, hit.normal^t, hit.normal, Vec3d(0)); // creo la
        matrice TangentToWorld che passa dal TangentSpace al WorldSpace
    if(mat->normalMap){ // se il materiale ha un normalmap
      Vec3d n = mat->normalMap->getValue(Vec3d(hit.texture.x, 1. - hit.texture.y,
          0.)); // recupero la normale dalla normalmap
      n.x = n.x*2 - 1; // rimappo i valori della normale tra [-1,1]
      n.y = n.y*2 - 1; // rimappo i valori della normale tra [-1,1]
      hit.normal = (hit.TBN->operator*(Vec4d(n, 0.))).xyz().hat(); // passo la
          normale al worldspace
      // *hit.TBN = Mat4d(t, hit.normal^t, hit.normal, Vec3d(0));
    }
    if(mat->isEmissive()) // se il materiale Ăš emissivo
      hit.radiance += mat->getEmission(hit.texture); // aggiungo alla radianza
          l'emissione del materiale
    for(u_int i = 0; i < w->lights.size(); ++i){ // per tutte le luci della scena
      Vec3d L = -w->lights[i]->getDirection(hit.hitPoint); // calcolo la
          direzione tra il punto e la luce
      double NoL = L*hit.normal; // calcolo il prodotto scalare tra la normale e
          la direzione tra il punto e la luce
      bias = hit.hitPoint + hit.normal*D_EPS; // calcolo il punto di partenza del
          raggio d'ombra, shiftando il punto di intersezione di una quantitĂ
          infinitiesima in direzione della normale (serve ad evitare
          autointersezioni)
      if(NoL < D_EPS) // se il punto non guarda la luce
        continue; // salto i calcoli restanti che darebbero contributo nullo a
            causa del selfshadowing
      Ray shadowRay = Ray(bias, L); // costruisco il raggio d'ombra
```

```
      Intersection shadowHit = w->intersectTree(shadowRay,
          w->lights[i]->getDistance(hit.hitPoint)); // traverso l'albero con il
          raggio d'ombra specificando la distanza sotto la quale deve avvenire
          l'intersezione
      if(shadowHit.hit) // se ho colpito
        continue; // salto i calcoli restanti che darebbero contributo nullo a
            causa di un altro oggetto che si interpone tra il punto e la luce
      hit.radiance += NoL*mat->shade(-ray.d, L,
          hit)%w->lights[i]->getColor()*w->lights[i]->getIntensity()*w->lights[i]->getAttenuatio
          // aggiungo alla radianza il contributo della luce pesato con la brdf e
          il termine di selfshadow
      // hit.radiance = hit.normal;
      // hit.radiance = t;
      // hit.radiance = hit.normal^t;
    }

    --_depth; // scalo la profonditÃ  di rimbalzo

    if(_depth > 0){ // se ho ulteriori rimbalzi
      ++w->stats->rays;
      Vec3d smpDir;
      Vec3d f = mat->sampleMaterial(-ray.d, smpDir, w->sampler->sample(), hit);
          // campiono il materiale e salvo il campione in smpDir e ritorno la
          brdf/pdf
      if(f == Vec3d()) // se f Ãš il vettore nullo
        return hit.radiance; // non considero il contributo
      // smpDir.normalize();
      Ray rayInd(bias, smpDir); // construisco il raggio riflesso con la
          direzione campionata
      double NoD = hit.normal*smpDir; // calcolo il prodotto scalare tra la
          direzione e la normale
      if(NoD < D_EPS) // se ho campionato una direzione che punta sotto la
          superfice
        return hit.radiance; // non considero il contributo
      hit.radiance += NoD*traceRay(rayInd, _depth)%f; // aggiungo alla radianza
          la radianza in entrata dalla direzione campionata pesata con il fattore
          f e NoD
    }

  }
  else{
    hit.radiance += w->env->getColor(ray.d); // se non ho colpito nulla campiono
        il colore dall'ambiente
  }

  return hit.radiance; // ritorno la radianza
}

std::string toString() const{
```

```
    return "PathTracingIntegrator: [ depth: " + std::to_string(depth) + "]";
  }


  int depth; // profonditÄ  massima di rimbalzi
};

#endif
```

whitted.hpp:

```
/*
  File di definizione della classe WhittedIntegrator, specializzazione della
      classe Integrator che modella l'integratore di whitted,
  ovvero calcola solo la luce diretta proveniente da uno o piu riflessi speculari
*/

#ifndef _WHITTED_INTEGRATOR_
#define _WHITTED_INTEGRATOR_

#include "integrator.hpp"

class WhittedIntegrator : public Integrator{
public:
  WhittedIntegrator(World * w): Integrator(w), depth(1){}
  WhittedIntegrator(World * w, int d): Integrator(w), depth(d){}
  ~WhittedIntegrator(){}


  Vec3d solve(const Ray & ray) const{ // metodo che risolve l'integrale della
      radianza
    return traceRay(ray, depth);
  }

  Vec3d traceRay(const Ray & ray, int _depth) const{ // metodo che calcola la
      radianza ricorsiva dato un raggio e il numero di rimbalzi
    Vec3d bias;
    Intersection hit;
    Material * mat;
    hit = w->intersectTree(ray); // trovo l'intersezione traversando l'albero
    w->stats->hits += hit.nHits;
    if(hit.hit){ // se ho colpito
     hit.hitPoint = ray(hit.t); // calcolo il punto colpito
     hit.normal = hit.obj_ptr->calcNormal(hit); // calcolo la normale al punto
     hit.texture = hit.obj_ptr->calcTexture(hit); // calcolo le texture del punto
     mat = hit.obj_ptr->getMaterial(); // mi salvo un riferimento al materiale
         della primitiva colpita
     if(mat->isEmissive()) // se il materiale Ăš emissivo
```

```
 hit.radiance += mat->getEmission(hit.texture); // aggiungo alla radianza
     l'emissione del materiale
for(u_int i = 0; i < w->lights.size(); ++i){ // per tutte le luci della scena
  Vec3d L = -w->lights[i]->getDirection(hit.hitPoint); // calcolo la direzione
      tra il punto e la luce
  double NoL = L*hit.normal; // calcolo il prodotto scalare tra la normale e
      la direzione tra il punto e la luce
  bias = hit.hitPoint + hit.normal*D_EPS; // calcolo il punto di partenza del
      raggio d'ombra, shiftando il punto di intersezione di una quantitĂ 
      infinitiesima in direzione della normale (serve ad evitare
      autointersezioni)
  if(NoL < D_EPS) // se il punto non guarda la luce
    continue; // salto i calcoli restanti che darebbero contributo nullo a
        causa del selfshadowing
  Ray shadowRay = Ray(bias, L); // costruisco il raggio d'ombra
  Intersection shadowHit = w->intersectTree(shadowRay,
      w->lights[i]->getDistance(hit.hitPoint)); // traverso l'albero con il
      raggio d'ombra specificando la distanza sotto la quale deve avvenire
      l'intersezione
  if(shadowHit.hit) // se ho colpito
    continue; // salto i calcoli restanti che darebbero contributo nullo a
        causa di un altro oggetto che si interpone tra il punto e la luce
  hit.radiance += NoL*mat->shade(-ray.d, L, hit)
  %w->lights[i]->getColor()*w->lights[i]->getIntensity()*w
  ->lights[i]->getAttenuation(hit.hitPoint);
  // aggiungo alla radianza il contributo della luce pesato con la brdf e il
      termine di selfshadow
}

--_depth; // scalo la profonditĂ  di rimbalzo

if(_depth > 0 && mat->isReflective()){ // se ho ulteriori rimbalzi da fare e
    il materiale Ăš riflettente
  ++w->stats->rays;
 //  bias += hit.normal*D_EPS;
  Vec3d R = ray.d.reflect(hit.normal); // calcolo la direzione di riflessione
  Ray reflection(bias, R); // costruisco il raggio riflesso
  hit.radiance += traceRay(reflection,
      _depth)%mat->getReflection(hit.texture); // aggiungo alla radianza, la
      radianza in entrata dal raggio riflesso pesata con in cofficiente di
      riflessione del materiale
}

}
else{
 hit.radiance += w->env->getColor(ray.d); // se non ho colpito nulla allora
     aggiungo alla radianza quella proveniente dal background
}
```

```
    return hit.radiance; // ritorno la radianza
  }

  std::string toString() const{
    return "WhittedIntegrator: [ depth: " + std::to_string(depth) + "]";
  }

  int depth; // profonditÃ  massima di rimbalzi
};

#endif
```

LIGHTS

---

directional.hpp:

```
/*
  File di definizione della classe PointLight, specializzazione della classe Light
      che modella una luce puntiforme
*/

#ifndef _DIRECTIONAL_LIGHT_
#define _DIRECTIONAL_LIGHT_

#include "light.hpp"
#include <limits>


class DirectionalLight: public Light{
public:
  DirectionalLight(): Light(), dir(-1., 0., 0.){}
  DirectionalLight(double i, const Vec3d & _c, const Vec3d & _dir): Light(i, _c),
      dir(_dir.hat()){}
  DirectionalLight(const DirectionalLight & dl): Light(dl.intensity, dl.c),
      dir(dl.dir){}

  DirectionalLight & operator=(const DirectionalLight & dl){
    Light::operator=(dl);
    dir = dl.dir;
    return *this;
  }

  void setDirection(const Vec3d & d){dir = d.hat();} // setta la direzione
  Vec3d getDirection(const Vec3d & p) const{return dir;} // ritorna la direzione
  double getAttenuation(const Vec3d & d) const{return 1.f;} // metodo astratto che
      calcola il fattore di attenuazione della luce
```

```
  double getDistance(const Vec3d & p) const{return
      std::numeric_limits<double>::max();} // metodo che calcola la distanza tra la
      posizione della luce e un altro punto della scena


  std::string toString() const{
    return "DirectionalLight: " + Light::toString() + ", Direction: " +
        dir.toString();
  }

private:
  Vec3d dir; // direzione della luce
};


#endif
```

light.hpp:

```
/*
  File di definizione della classe Light, modella una luce generica
*/

#ifndef _LIGHT_
#define _LIGHT_

#include "../geometry/vec3d.hpp"

class Light{
public:
  Light(): intensity(1.), c(1.){}
  Light(double i, const Vec3d & _c): intensity(i), c(_c){}
  Light(const Light & l): intensity(l.intensity), c(l.c){}
  virtual ~Light(){}

  Light & operator=(const Light & l){
    intensity = l.intensity; c = l.c;
    return *this;
  }

  void setIntensity(double i){intensity = i;} // setta l'intensitĂ  o potenza
  void setColor(const Vec3d & _c){c = _c;} // setta il colore
  double getIntensity() const{return intensity;} // ritorna l'intensitĂ  o potenza
  Vec3d getColor() const{return c;} // ritorna il colore
  virtual Vec3d getDirection(const Vec3d &) const = 0; // metodo astratto che
      ritorna la direzione tra la posizione della luce e un altro punto della scena
  virtual double getAttenuation(const Vec3d &) const = 0; // metodo astratto che
      calcola il fattore di attenuazione della luce
```

```
  virtual double getDistance(const Vec3d &) const = 0; // metodo che calcola la
      distanza tra la posizione della luce e un altro punto della scena

  virtual std::string toString() const{
    return "Power: " + std::to_string(intensity) + ", Color: " + c.toString();
  }

protected:
  double intensity; // intensitÃ  o potenza della luce
  Vec3d c; // colore della luce
};

#endif
```

point.hpp:

```
/*
  File di definizione della classe PointLight, specializzazione della classe Light
      che modella una luce puntiforme
*/

#ifndef _POINT_LIGHT_
#define _POINT_LIGHT_

#include "light.hpp"

class PointLight: public Light{
public:
  PointLight(): Light(), p(){}
  PointLight(double i, const Vec3d & _c, const Vec3d & _p): Light(i, _c), p(_p){}
  PointLight(const PointLight & pl): Light(pl), p(pl.p){}

  PointLight & operator=(const PointLight & pl){
    Light::operator=(pl);
    p = pl.p;
    return *this;
  }

  void setPosition(const Vec3d & _p){p = _p;} // setta la posizione
  Vec3d getPosition() const{return p;} // ritorna la posizione
  Vec3d getDirection(const Vec3d & p1) const{return (p1 - p).hat();} // ritorna la
      direzione tra la posizione della luce e un altro punto della scena
  double getAttenuation(const Vec3d & p1) const{return
      1./std::max(1.,(p1-p).lengthSq()*.01);} // calcola il fattore di attenuazione
      della luce
  double getDistance(const Vec3d & p1) const{return (p1 - p).length();} // calcola
      la distanza tra la posizione della luce e un altro punto della scena

  std::string toString() const{
```

```
    return "PointLight: " + Light::toString() + ", Position: " + p.toString();
  }

private:
  Vec3d p; // posizione della luce
};

#endif
```

MATERIALS

---

material.hpp:

```
/*
  File di definizione della classe Material, modella un materiale generico
*/

#ifndef _MATERIAL_
#define _MATERIAL_

#include "../brdfs/brdf.hpp"
#include "../brdfs/multipleBrdf.hpp"
#include "../brdfs/addBrdf.hpp"
#include "../geometry/vec3d.hpp"
#include "../utilities/intersection.hpp"
#include <vector>

class Material{
public:
  Material(): mBrdf(new AddBrdf()), reflective(false), emissive(false),
      reflection(0.)
  ,emission(0.), reflMap(nullptr), emisMap(nullptr), normalMap(nullptr){}
  Material(MultipleBrdf * mb): mBrdf(mb), reflective(false), emissive(false),
      reflection(0.)
  ,emission(0.), reflMap(nullptr), emisMap(nullptr), normalMap(nullptr){}
  virtual ~Material(){
    delete mBrdf;
    delete reflMap;
    delete emisMap;
    delete normalMap;
  }

  Vec3d shade(const Vec3d & wo, const Vec3d & wi, Intersection & hit) const{ //
      metodo che valuta la brdf multipla
    return mBrdf->evaluate(wo, wi, hit);
  }
```

```
Vec3d sampleMaterial(const Vec3d & wo, Vec3d & wi, const Vec3f & xi,
    Intersection & hit) const{ // metodo che campiona una delle brdf con
    probabilitÃ  uniforme
  int n = rand()%mBrdf->brdfs.size(); // numero casuale tra 0 e brdfs.size()-1
  return mBrdf->brdfs[n]->sample(wo, wi, xi, hit)*mBrdf->pdf(); // ritorna la
      n-esima brdf campionata
}

void setReflection(const Vec3d & r){ // setta la riflessione tramite vettore
  reflective = true;
  reflection = r;
}

void setEmission(const Vec3d & r){ // setta l'emissione tramite vettore
  emissive = true;
  emission = r;
}

void setReflMap(const std::string & path){ // setta la riflessione tramite
    texture
  reflective = true;
  delete reflMap;
  reflMap = new Texture(path);
}

void setEmisMap(const std::string & path){ // setta l'emissione tramite texture
  emissive = true;
  delete emisMap;
  emisMap = new Texture(path);
}

void setNormalMap(const std::string & path){ // setta la normalmap tramite
    texture
  delete normalMap;
  normalMap = new Texture(path);
}

Vec3d getReflection(const Vec3d & uv) const{ // ritorna la riflessione
  if(reflMap) // se ho una texture
    return reflMap->getValue(uv); // campiona la texture tramite il vettore uv
  else
    return reflection; // altrimenti ritorna il valore vettoriale
}

Vec3d getEmission(const Vec3d & uv) const{ // ritorna l'emissione
  if(emisMap) // se ho una texture
    return emisMap->getValue(uv); // campiona la texture tramite il vettore uv
  else
```

```
    return emission; // altrimenti ritorna il valore vettoriale
  }

  bool isReflective() const{ // controlla se Ăš riflettente
    return reflective;
  }

  bool isEmissive() const{ // controlla se Ăš emissivo
    return emissive;
  }

  virtual std::string toString() const{
    std::string s, t;
    if(reflective){
      if(reflMap)
        s = "[ RefMap: " + reflMap->toString();
      else
        s = "[ Amount: " + reflection.toString();
    }
    else
      s = "none";
    if(emissive){
      if(emisMap)
        t = "[ EmisMap: " + emisMap->toString();
      else
        t = "[ Amount: " + emission.toString();
    }
    else
      t = "none";
    return "Material: [Refl: " + s + ", Emis:" + t + ", " + mBrdf->toString() + "]";
  }


  MultipleBrdf * mBrdf; // brdf multipla
  bool reflective, emissive; // booleani per la riflessione o emissione
  Vec3d reflection, emission; // vettori per il colore della riflessione e
      dell'emissione
  Texture * reflMap, * emisMap,* normalMap; // texture per la riflessione,
      emissione e normalmap
};

#endif
```

SAMPLERS:

hammerslay.hpp:

```
/*
```

```
  File di definizione della classe HammerslaySampler, specializzazione della
      classe StratifiedSampler
   che modella un campionatore che campiona il quadrato tramite la sequenza di
       numeri di Halton
*/

#ifndef _HAMMERSLAY_SAMPLER_
#define _HAMMERSLAY_SAMPLER_

#include "stratified.hpp"

class HammerslaySampler : public StratifiedSampler{
public:
  HammerslaySampler(): StratifiedSampler(){}
  HammerslaySampler(int s): StratifiedSampler(s){}
  ~HammerslaySampler(){}

  Vec3f sample(){ // calcola il campione sul quadrato
    return Vec3f(current*invSamples, halton(current, 2), 0.f);
  }

  std::string toString() const{
    return "HammerslaySampler: " + Sampler::toString();
  }

};

#endif
```

---

multijitter.hpp:

```
/*
  File di definizione della classe MultijitterSampler, specializzazione della
      classe StratifiedSampler
   che modella un campionatore che campiona il quadrato tramite due sotto-griglie
   con numero di celle per lato pari alla radice quadrata del numero di campioni.
   Il campione Ăš situato uniformemente in ogni cella della seconda sotto-griglia
   in modo che la sua proiezione sugli assi x e y non coincida con quella degli
       altri campioni.
   In seguito vengono operate due permutazioni di celle sui due assi.
  (implementato grazie al paper della Pixar
      "http://graphics.pixar.com/library/MultiJitteredSampling/paper.pdf")
*/

#ifndef _MULTIJITTER_SAMPLER_
#define _MULTIJITTER_SAMPLER_

#include "stratified.hpp"
```

```
class MultijitterSampler : public StratifiedSampler{
public:
  MultijitterSampler(): StratifiedSampler(), seed(0){}
  MultijitterSampler(int s): StratifiedSampler(s), seed(0){}
  ~MultijitterSampler(){}

  Vec3f sample(){ // calcola il campione sul quadrato
    div_t xy = div(current, n); // calcolo gli indici del campione corrente che
        giace nella cella della griglia
    u_int px = permute(xy.rem, n, seed*0xa511e9b3); // calcolo l'indice colonna
        permutato
    u_int py = permute(xy.quot, n, seed*0x63d83595); // calcolo l'indice riga
        permutato
    float jx = randfloat(current, seed*0xa399d265); // calcolo il jitter uniforme
        tra [0,1] per le x
    float jy = randfloat(current, seed*0x711ad6a5); // calcolo il jitter uniforme
        tra [0,1] per le y
    return Vec3f((xy.rem + (py+jx)*invN), (xy.quot + (px+jy)*invN), 0)*invN; //
        ritorno il campione
  }

  void setSeed(int s){ // setto il seme
    seed = s;
  }

  std::string toString() const{
    return "MultijitterSampler: " + Sampler::toString();
  }

  int seed; // seme del campionatore
};


#endif
```

---

random.hpp:

```
/*
  File di definizione della classe RandomSampler, specializzazione della classe
      Sampler
   che modella un campionatore che campiona il quadrato [0,1]x[0,1] in modo uniforme
*/

#ifndef _RANDOM_SAMPLER_
#define _RANDOM_SAMPLER_

#include "sampler.hpp"

class RandomSampler : public Sampler{
```

```
public:
  RandomSampler(): Sampler(){}
  RandomSampler(int s): Sampler(s){}
  ~RandomSampler(){}

  Vec3f sample(){ // calcola il campione uniforme sul quadrato
    return Vec3f(dist(gen), dist(gen), 0.f);
  }

  void nextSample(){} // passa al sample successivo

  std::string toString() const{
    return "RandomSampler: " + Sampler::toString();
  }

};

#endif
```

---

regular.hpp:

```
/*
  File di definizione della classe RegularSampler, specializzazione della classe
      Sampler
   che modella un campionatore che campiona l'origine del quadrato [0,1]x[0,1]
  (serve soltanto se non si vuole avere l'antialiasig sullo schermo)
*/

#ifndef _REGULAR_SAMPLER_
#define _REGULAR_SAMPLER_

#include "sampler.hpp"

class RegularSampler : public Sampler{
public:
  RegularSampler(): Sampler(1){}
  ~RegularSampler(){}

  Vec3f sample(){ // calcola il campione (0,0,0)
    return Vec3f(0.f);
  }

  void nextSample(){} // passa al sample successivo

  std::string toString() const{
    return "RegularSampler: " + Sampler::toString();
  }

};
```

```
#endif
```

sampler.hpp:

```
/*
  File di definizione della classe Sampler, modella un generico campionatore
*/

#ifndef _SAMPLER_
#define _SAMPLER_

#include "../geometry/vec3f.hpp"
#include "../utilities/constants.hpp"
#include <random>

class Sampler{
public:
  Sampler(): samples(4), gen(0), dist(0.f, 1.f){}
  Sampler(int s): samples(s), gen(0), dist(0.f, 1.f){}
  virtual ~Sampler(){}

  virtual Vec3f sample() = 0; // metodo astratto che calcola un campione in
      [0,1]x[0,1]

  virtual void nextSample() = 0; // metodo astratto che passa al campione
      successivo della lista

  virtual void setSeed(int s){ // setta il seme per la generazione dei numeri
      casuali
    gen.seed(s);
  }

  Vec3f sampleScreen(int i, int width){ // metodo che campiona la griglia di pixel
      dato un indice lineare e la larghezza dello schermo
    div_t px = div(i, width); // calcolo la divisione intera con resto
    Vec3f sp = sample(); // calcolo il campione sul pixel in base all campionatore
        attuale
    return Vec3f(px.rem + sp.x, px.quot + sp.y, 0.f); // calcolo il campione sullo
        schermo shiftando gli indici della matrice tramite il campione sul pixel
  }

  virtual std::string toString() const{
    return "{Samples: " + std::to_string(samples) + "}";
  }

  int samples; // numero complessivo di campioni
  std::mt19937_64 gen; // generatore di numeri casuali
```

```
  std::uniform_real_distribution<float> dist; // distribuzione uniforme di numeri
      reali
};

#endif
```

stratified.hpp:

```
/*
  File di definizione della classe StratifiedSampler, specializzazione della
      classe Sampler
   che modella un generico campionatore sequenziale o stratificato
*/

#ifndef _STRATIFIED_SAMPLER_
#define _STRATIFIED_SAMPLER_

#include "sampler.hpp"
#include "../utilities/math.hpp"

class StratifiedSampler : public Sampler{
public:
  StratifiedSampler(): Sampler(), current(0), n(sqrt(samples)), invN(1.f/n),
      invSamples(1.f/samples){}
  StratifiedSampler(int s): Sampler(s), current(0), n(sqrt(s)), invN(1.f/n),
      invSamples(1.f/samples){}
  virtual ~StratifiedSampler(){}

  void nextSample(){ // passa al campione successivo
    ++current %= samples; // incremento il sample corrente e lo clampo nel range
        del numero di campioni
  }

  u_int current; // campione corrente
  u_int n; // radice quadrata dei campioni, ovvero campioni per lato del quadrato
  float invN, invSamples; // reciproco del numero di campioni per lato e del
      numero di campioni totale
};


#endif
```

superSampler.hpp:

```
/*
  File di definizione della classe SuperSampler, specializzazione della classe
      StratifiedSampler
   che modella un campionatore che campiona un quadrato tramite una sotto-griglia
   con numero di celle per lato pari alla radice quadrata del numero di campioni.
   Il campione Ăš situato nell'origine della cella.
```

```
*/

#ifndef _SUPER_SAMPLER_
#define _SUPER_SAMPLER_

#include "stratified.hpp"

class SuperSampler : public StratifiedSampler{
public:
  SuperSampler(): StratifiedSampler(){}
  SuperSampler(int s): StratifiedSampler(s){}
  ~SuperSampler(){}

  Vec3f sample(){ // calcola il campione sul quadrato
    div_t xy = div(current, n); // calcolo gli indici del campione corrente che
        giace nella cella della griglia
    return Vec3f(xy.rem , xy.quot, 0.f)*invN; // ritorno il campione normalizzato
        sulla lunghezza del lato della griglia
  }

  std::string toString() const{
    return "SuperSampler: " + Sampler::toString();
  }

};


#endif
```

UTILITIES:

constants.hpp:

```
/*
  File di definizione delle costanti usate dal programma
*/

#ifndef _CONSTANTS
#define _CONSTANTS

#include "../geometry/vec3d.hpp"
#include "../geometry/vec3f.hpp"

const double PI = 3.1415926535897932384;
const double TWO_PI = 6.2831853071795864769;
const double PI_ON_180 = 0.0174532925199432957;
const double invPI = 0.3183098861837906715;
const double invTWO_PI = 0.1591549430918953358;
```

```
const float EPS = 1e-6;
const double D_EPS = 1e-12;

const Vec3d X_AXIS = Vec3d(1.,0.,0.);
const Vec3d Y_AXIS = Vec3d(0.,1.,0.);
const Vec3d Z_AXIS = Vec3d(0.,0.,1.);
const Vec3d XY_VEC = Vec3d(1.,1.,0.);
const Vec3d YZ_VEC = Vec3d(0.,1.,1.);
const Vec3d ZX_VEC = Vec3d(1.,0.,1.);
const Vec3d XYZ_VEC = Vec3d(1.,1.,1.);

const Vec3d BLACK = Vec3d(0.);
const Vec3d WHITE = Vec3d(1.);
const Vec3d GREY = Vec3d(.5, .5, .5);
const Vec3d RED = Vec3d(1., 0., 0.);
const Vec3d MATTE_RED = Vec3d(1., .25, .25);
const Vec3d GREEN = Vec3d(0., 1., 0.);
const Vec3d MATTE_GREEN = Vec3d(0., 1., .25);
const Vec3d BLUE = Vec3d(0., 0., 1.);
const Vec3d MATTE_BLUE = Vec3d(.25, .25, 1.);
const Vec3d YELLOW = Vec3d(1., 1., 0.);
const Vec3d MATTE_YELLOW = Vec3d(1., 1., .25);
const Vec3d MAGENTA = Vec3d(1., 0., 1.);
const Vec3d MATTE_MAGENTA = Vec3d(1., .25, 1.);
const Vec3d CYAN = Vec3d(0., 1., 1.);
const Vec3d MATTE_CYAN = Vec3d(.25, 1., 1.);
const Vec3d ORANGE = Vec3d(1., .5, 0.);
const Vec3d MATTE_ORANGE = Vec3d(1., .5, .25);

#endif
```

---

environment.hpp:

```
/*
  File di definizione della classe Environment, modella l'ambiente di sfondo della
     scena.
  PuÃČ avere un colore uniforme o una texture sferica
*/

#ifndef _ENVIRONMENT_
#define _ENVIRONMENT_

#include "../geometry/vec3d.hpp"
#include "../utilities/texture.hpp"

class Environment{
public:
  Environment(): color(), map(nullptr){}
```

```
  Environment(const Vec3d & c): color(c), map(nullptr){}
  Environment(const std::string & path): color(), map(new Texture(path)){}
  ~Environment(){
    delete map;
  }

  Vec3d getUV(const Vec3d & n) const{ // calcola il valore texture data una
      direzione
    return Vec3d(.5 + atan2(n.x,n.y)*invTWO_PI
                ,.5 - asin(n.z)*invPI
                ,1.);
  }

  Vec3d getColor(const Vec3d & n) const{ // ritorna il colore ambientale
    if(map) // se ho una texture
      return map->getValue(getUV(n)); // campiona le texture
    else
      return color; // altrimenti ritorna il colore
  }

  std::string toString() const{
    std::string s;
    if(map)
      s = "Map: " + map->toString();
    else
      s = "Color: " + color.toString();
    return "Environment: [ " + s + "]";
  }

  Vec3d color; // colore
  Texture * map; // texture sferica
};



#endif
```

film.hpp:

```
/*
  File di definizione della classe Film, specializzazione della classe PixelBuffer
      che modella l'immagine finale del rendering
*/

#ifndef _FILM_
#define _FILM_

#include "pixelBuffer.hpp"
#include "../utilities/math.hpp"
```

```
class Film: public PixelBuffer{
public:
  Film(): PixelBuffer(), path("rendering.bmp"), temp_map(nullptr){}
  Film(int w, int h, int c): PixelBuffer(w,h,c), path("rendering.bmp"){
    temp_map = new double[size()];
    for(int i = 0; i < size(); ++i)
      temp_map[i] = 0.;
  }
  Film(int w, int h, int c, const std::string & p): PixelBuffer(w, h, c), path(p){
    temp_map = new double[size()];
    for(int i = 0; i < size(); ++i)
      temp_map[i] = 0.;
  }
  ~Film(){
    delete [] map;
    delete [] temp_map;
  }

  void impress(int i, const Vec3d & c){ // imprime il colore c all'indice i
      dell'array
    i *= channels; // scalo l'indice per i canali
    Vec3d val = to255(c); // mappo il colore tra 0 e 255
    map[i] = static_cast<int>(val.x); // salvo il colore nel canale corrispondente
    map[i+1] = static_cast<int>(val.y); // salvo il colore nel canale corrispondente
    map[i+2] = static_cast<int>(val.z); // salvo il colore nel canale corrispondente
  }

  void impressAdd(int i, const Vec3d & c){ // versione additiva di impress
    i *= channels; // scalo l'indice per i canali
    temp_map[i] += c.x; // aggiungo il colore nel canale corrispondente
    temp_map[i+1] += c.y; // aggiungo il colore nel canale corrispondente
    temp_map[i+2] += c.z; // aggiungo il colore nel canale corrispondente
  }

  void finalize(double a, double b, double n){ // finalizza l'immagine forzando i
      valori in [0,1], applicando la gamma, e rimappando i valori in [0,255]
    for(int i = 0; i < size(); ++i){
      map[i] = static_cast<int>(pow(clamp(temp_map[i]*n, a, b), .4545)*255);
    }
  }

  int save(){ // salva l'immagine su disco usando la libreria SOIL
    if(SOIL_save_image(path.c_str(), SOIL_SAVE_TYPE_BMP, width, heigth, channels,
       map)){
      std::cout << "\tImage succesfully saved to " << path << "\n";
      return true;
    }
    else{
```

```
      std::cout << "\t[ERROR] File saving failed.\n";
    }
  }

  int save(const std::string & path){ // salva l'immagine su disco usando la
      libreria SOIL nel percorso path
    if(SOIL_save_image(path.c_str(), SOIL_SAVE_TYPE_BMP, width, heigth, channels,
        map)){
      std::cout << "\tImage succesfully saved to " << path << "\n";
      return true;
    }
    else{
      std::cout << "\t[ERROR] File saving failed.\n";
    }
  }

  std::string toString() const{
    return "Film: " + PixelBuffer::toString() + ", Path: " + path + "]";
  }

  std::string path; // percorso dell'immagine finale
  double * temp_map; // array temporaneo usato in impressAdd per aggiungere
      progressivamente colore
};

inline std::ostream & operator<<(std::ostream & os, const Film & f){
  return os << "Film: {Res: " << f.width << "x" << f.heigth << " - Path: " <<
      f.path << "}";
}

#endif
```

intersection.hpp:

```
/*
  File di definizione della classe Intersection, struttura utilizzata per salvare
      le informazioni durante la routine di intersezione e di rendering
*/

#ifndef _INTERSECTION_
#define _INTERSECTION_

#include "../geometry/vec3d.hpp"
#include "../geometry/transform.hpp"

class Primitive;

class Intersection{
public:
```

```
  Intersection(): hit(false), nHits(0), t(0.), u(0.), v(0.), hitPoint(), normal()
                , radiance(), texture(), obj_ptr(nullptr), TBN(nullptr){}
  Intersection(const Intersection & i): hit(i.hit), nHits(i.nHits), t(i.t), u(i.u)
                                      , v(i.v), hitPoint(i.hitPoint)
                                      , normal(i.normal), radiance(i.radiance)
                                      , texture(i.texture), obj_ptr(i.obj_ptr),
                                         TBN(i.TBN){}
  ~Intersection(){
    delete TBN;
  }

  Intersection & operator=(const Intersection & i){
    t = i.t; u = i.u; v = i.v; hit = i.hit; nHits = i.nHits; hitPoint = i.hitPoint;
    normal = i.normal; radiance = i.radiance; texture = i.texture; obj_ptr =
        i.obj_ptr;
    delete TBN;
    TBN = i.TBN;
    return *this;
  }

  bool hit; // booleano per indicare se ha colpito
  long int nHits;
  double t, u, v; // distanza parametrica di intersezione, prima coordinata
      baricentrica, seconda coordianta baricentrica
  Vec3d hitPoint, normal, radiance, texture; // punto colpito, normale al punto,
      radianza nel punto, texture nel punto
  Primitive * obj_ptr; // puntatore alla primitiva colpita
  Mat4d * TBN; // matrice di passaggio da TangentSpace a WorldSpace
};

inline std::ostream & operator<<(std::ostream & os, const Intersection & it){
  return os << "Inters - " << "hit: " << it.hit << " - nHits: " << it.nHits << " -
      t: "
            << it.t << " - uv: [" << it.u << ", " << it.v << "]" << " - HitPoint: "
            << it.hitPoint << " - Normal: " << it.normal << " - Radiance: " <<
               it.radiance
            << " - Ptr: " << it.obj_ptr;
}


#endif
```

---

math.hpp:

```
/*
  File di definizione delle principali funzioni matematiche usate nel programma
*/
```

```
#ifndef _MATHEMATICS
#define _MATHEMATICS

#include "constants.hpp"
#include <math.h>

inline float toRad(float deg){ // passa da gradi a radianti
  return deg*PI_ON_180;
}

inline float toDeg(float rad){ // passa da radianti a gradi
  return rad/PI_ON_180;
}

inline Vec3d to01(double r, double g, double b){ // passa da [0,255] a [0,1]
  return Vec3d(r, g, b)/255;
}

inline Vec3d to255(const Vec3d & c){ // passa tra [0,1] a [0,255]
  return c*255;
}

inline int clamp(int n, int a, int b){ // forza il numero n a stare tra a e b
  return n <= a ? a : n >= b ? b : n;
}

inline double clamp(double n, double a, double b){ // forza il numero n a stare
    tra a e b
  return n <= a ? a : n >= b ? b : n;
}

inline Vec3d clamp(const Vec3d & v, double a, double b){ // forza il vettore v a
    stare tra a e b
  Vec3d u;
  for(int i = 0; i < 3; ++i){
    u[i] = clamp(v[i], a, b);
  }
  return u;
}

inline bool isZero(double val){ // controlla se val tende a 0
  return val > -D_EPS && val < D_EPS;
}

inline bool isValue(double n, double val ){ // controlla se n tende a val
  return n > val - D_EPS && n < val + D_EPS;
}
```

```
inline int solveQuadratic(double a, double b, double c, double & x1, double & x2){
    // risolve un equazione di secondo grado
  double delta = b*b - 4*a*c;
  double aInv = 1./(2*a);
  if(delta < 0){
    return 0;
  }
  else if(isZero(delta)){
    x1 = -b*aInv;
    return 1;

  }
  else{
    x1 = (-b + sqrt(delta))*aInv;
    x2 = -2*b*aInv - x1;
    return 2;
  }
}


inline float halton(u_int n, u_int b){ // calcola l'n-esimo valore della sequenza
    di Halton in base b
  float val = 0.f;
  u_int _n = n;
  float baseInv = 1.f/b;
  float _bInv = baseInv;
  while(_n > 0.f){
    u_int d = _n%b;
    val += d*_bInv;
    _n = (_n >> 1);
    _bInv *= baseInv;
  }
  return val;
}

inline u_int permute(u_int i, u_int l, u_int p){ // calcola una permutazione
  u_int w = l - 1;
  w |= w >> 1;
  w |= w >> 2;
  w |= w >> 4;
  w |= w >> 8;
  w |= w >> 16;
  do{
    i ^= p; i *= 0xe170893d;
    i ^= p >> 16;
    i ^= (i & w) >> 4;
    i ^= p >> 8; i *= 0x0929eb3f;
    i ^= p >> 23;
    i ^= (i & w) >> 1; i *= 1 | p >> 27;
```

```
    i *= 0x6935fa69;
    i ^= (i & w) >> 11; i *= 0x74dcb303;
    i ^= (i & w) >> 2; i *= 0x9e501cc3;
    i ^= (i & w) >> 2; i *= 0xc860a3df;
    i &= w;
    i ^= i >> 5;
 }
 while (i >= l);
 return (i + p)%l;
}

inline float randfloat(u_int i, u_int p) { // calcola un numero random uniforme
    tra [0,1]
  i ^= p;
  i ^= i >> 17;
  i ^= i >> 10; i *= 0xb36534e5;
  i ^= i >> 12;
  i ^= i >> 21; i *= 0x93fc4795;
  i ^= 0xdf6e307f;
  i ^= i >> 17; i *= 1 | p >> 18;
  return i * (1.0f / 4294967808.0f);
}

inline Vec3f toDisk(const Vec3f & smp){ // mappa un campione giacente nel quadrato
    [0,1]x[0,1] a un disco unitario
  float phi, r, piq = PI*.25;
  float a = 2*smp.x-1;
  float b = 2*smp.y-1;
  if(a > -b){
    if(a > b){
      r = a;
      phi = (piq)*(b/a);
    }
    else{
      r = b;
      phi = (piq)*(2 - (a/b));
    }
  }
  else{
    if(a < b){
      r = -a;
      phi = (piq) * (4 + (b/a));
    }
    else{
      r = -b;
      if(b != 0){
        phi = (piq)*(6 - (a/b));
      }
      else{
```

```
        phi = 0;
      }
    }
  }
  return Vec3f(r*cos(phi), r*sin(phi), 0.f);
}

inline Vec3d toHemisphere(const Vec3f & s){ // mappa un campione giacente in un
    disco unitario all'emisfero in modo uniforme
  Vec3f smp = toDisk(s);
  double r = sqrt(2. - smp.lengthSq());
  return Vec3d(smp.x*r, smp.y*r, 1. - smp.lengthSq());
}

inline Vec3d toCosineHemisphere(const Vec3f & s){ // mappa un campione giacente in
    un disco unitario all'emisfero pesato con il coseno
  Vec3f smp = toDisk(s);
  return Vec3d(smp.x, smp.y, sqrt(1 - smp.x*smp.x - smp.y*smp.y));
}

inline Vec3d buildTangent(const Vec3d & dp1, const Vec3d & dp2, const Vec3d & du1,
    const Vec3d & du2){
  double r = 1./(du1.x*du2.y - du1.y*du2.x);
  return (dp1*du2.y - dp2*du1.y)*r;
}

#endif
```

---

parser.cpp:

```
/*
  File di definizione della classe Parser, struttura usata per importare i dati
  di una scena descritta in un file xml e per caricare i dati di un oggetto 3D
      descritto in un file obj
*/

#include "parser.hpp"
#include "../geometry/vec3d.hpp"
#include "../geometry/transform.hpp"
#include "../cameras/perspective.hpp"
#include "../cameras/orthographic.hpp"
#include "../samplers/multijitter.hpp"
#include "../samplers/hammerslay.hpp"
#include "../samplers/superSampler.hpp"
#include "../samplers/random.hpp"
#include "../samplers/regular.hpp"
#include "../accelerators/accelerator.hpp"
#include "../geometricObjects/objectGroup.hpp"
#include "../geometricObjects/triangle.hpp"
```

```
#include "../geometricObjects/sphere.hpp"
#include "../lights/directional.hpp"
#include "../lights/point.hpp"
#include "../materials/material.hpp"
#include "../brdfs/lambert.hpp"
#include "../brdfs/blinn.hpp"
#include "../brdfs/cooktorrance.hpp"
#include "../utilities/texture.hpp"
#include "../utilities/film.hpp"
#include "../utilities/environment.hpp"
#include "../integrators/whitted.hpp"
#include "../integrators/pathtracing.hpp"
#include "../integrators/ambientOcclusion.hpp"
#include "../world.hpp"
#include <iostream>
#include <fstream>

using namespace rapidxml;

Parser::Parser(World * _w): w(_w), imp(new Assimp::Importer()){
  parseList = {"Film", "Camera", "Sampler", "Accelerator", "Integrator",
      "Materials", "Objects", "Lights", "Environment"};
}

Parser::Parser(const Parser & p): w(p.w), imp(p.imp), parseList(p.parseList){}

Parser::~Parser(){
  delete imp;
}

Parser & Parser::operator=(const Parser & p){
  w = p.w; imp = p.imp;
  return *this;
}

bool Parser::parseScene(const std::string & path){ // metodo principale che parsa
    una scena da un file xml con percorso path
  std::ifstream sceneFile(path);
  std::vector<char> buffer((std::istreambuf_iterator<char>(sceneFile)),
      std::istreambuf_iterator<char>());
  buffer.push_back('\0');
  scene.parse<0>(&buffer[0]);
  // std::cout << scene << std::endl;

  for(auto & item : parseList){
    xml_node<> * node;
    if((node = scene.first_node(item.c_str()))){
      parseSceneNode(node);
    }
```

```
    else{
      std::cout << "[Warning] <" << item << "> tag missing! Used default value.\n";
      setDefault(item);
    }
  }

  w->turbo->init();

  std::cout << "\n" << *w->film << "\n\n" << w->cam->toString() << "\n\n" <<
      w->sampler->toString()
            << "\n\n" << *w->turbo << "\n\n" << w->env->toString() << "\n\n";

  for(auto & mat : w->materials)
    std::cout << mat.first << ": " << mat.second->toString() << "\n";

    std::cout << std::endl;

  for(auto & model : w->models)
    std::cout << "Model: " << model->toString() << "\n";

    std::cout << std::endl;

  for(auto & light : w->lights)
    std::cout << light->toString() << "\n";


  return true;
}

ObjectGroup * Parser::importModel(const std::string & path, Material * mat,
    Transform * tr){ // metodo che importa un modello 3D dato il percorso del file
    obj, il materiale e la trasformazione da associargli
  const aiScene * scene = imp->ReadFile(path,
                                        aiProcess_CalcTangentSpace    |
                                        aiProcess_Triangulate         |
                                        aiProcess_JoinIdenticalVertices |
                                        aiProcess_SortByPType         |
                                        aiProcess_FlipUVs
                                        );

  ObjectGroup * og = new ObjectGroup(tr);

  if(!scene){
    std::cout << "[ERROR] File reading failed, maybe invalid path?\n";
    return og;
  }

  traverseModelNode(scene->mRootNode, scene, og, mat);
  // og->setBVertex();
```

```
  // Vec3d diag = og->bbox->diag();
  // std::cout << "Min: " << og->bbox->min << " - Max: " << og->bbox->max << "\n";
  // std::cout << "Extention: " << diag << "\n";
  return og;
}

bool Parser::importMesh(const aiMesh * mesh, ObjectGroup * og, Material * mat){ //
    metodo che importa una mesh di triangoli
  Vec3d p0, p1, p2, n0, n1, n2, t0, t1, t2, tn0, tn1, tn2;
  u_int nFaces = mesh->mNumFaces;
  std::cout << "Faces #: " << nFaces << "\n";
  for(u_int j = 0; j < nFaces; ++j){
    aiFace & face = mesh->mFaces[j];
    p0 = mesh->mVertices[face.mIndices[0]];
    p1 = mesh->mVertices[face.mIndices[1]];
    p2 = mesh->mVertices[face.mIndices[2]];
    if(mesh->HasTextureCoords(0)){
      t0 = mesh->mTextureCoords[0][face.mIndices[0]];
      t1 = mesh->mTextureCoords[0][face.mIndices[1]];
      t2 = mesh->mTextureCoords[0][face.mIndices[2]];
    }
    if(mesh->HasNormals()){
      n0 = mesh->mNormals[face.mIndices[0]];
      n1 = mesh->mNormals[face.mIndices[1]];
      n2 = mesh->mNormals[face.mIndices[2]];
    }
    else{
      n0 = n1 = n2 = (p1-p0)^(p2-p0);
    }
    if(/*mesh->HasTangentsAndBitangents()*/false){
      tn0 = mesh->mTangents[face.mIndices[0]];
      tn1 = mesh->mTangents[face.mIndices[1]];
      tn2 = mesh->mTangents[face.mIndices[2]];
      // std::cout << tn0 << tn1 << tn2 << "\n";
      // std::cout << "yes";
    }
    // else if(mesh->HasTextureCoords(0)){
    //   tn0 = tn1 = tn2 = buildTangent(p1-p0, p2-p0, t1-t0, t2-t0);
    // }
    else{
      Vec3d up = fabs(n0.z) < 1.-D_EPS ? Vec3d(0.0,0.0,1.0) : Vec3d(1,0,0);
      tn0 = (up^n0).hat();
      up = fabs(n1.z) < 1.-D_EPS ? Vec3d(0.0,0.0,1.0) : Vec3d(1,0,0);
      tn1 = (up^n1).hat();
      up = fabs(n2.z) < 1.-D_EPS ? Vec3d(0.0,0.0,1.0) : Vec3d(1,0,0);
      tn2 = (up^n2).hat();
      // std::cout << tn0 << tn1 << tn2 << "\n";
      // std::cout << "no";
    }
```

```
    og->addObject(new Triangle(p0, p1, p2, n0, n1, n2, t0, t1, t2, tn0, tn1, tn2,
        mat));
  }
  return true;
}

bool Parser::traverseModelNode(const aiNode * node, const aiScene * scene,
    ObjectGroup * og, Material * mat){ // metodo che traversa un nodo del modello 3D

  int nMeshes = node->mNumMeshes, nChildren = node->mNumChildren;
  std::string name = node->mName.C_Str();

  std::cout << "Node name: " << name;

  Material * m = mat;
  for(auto & i : w->materials){
    if(i.first == name){
      m = i.second;
    }
  }

  if(nChildren != 0){
    std::cout << " - Children #: " << nChildren << " -->\n";
    for(int i = 0; i < nChildren; ++i){
      std::cout << "Child " << i << ":\n";
      traverseModelNode(node->mChildren[i], scene, og, m);
    }
    return true;
  }
  else if(nMeshes != 0){
    std::cout << " - Meshes #: " << nMeshes << " -->\n";
    for(int i = 0; i < nMeshes; i++){
      std::cout << "Mesh " << ":\n";
      importMesh(scene->mMeshes[node->mMeshes[i]], og, m);
    }
    return true;
  }
  else
    return false;

  std::cout << "<--\n";

}

ObjectGroup * Parser::traverseObjectsNode(xml_node<> * node){ // metodo che
    traversa un nodo xml della scena
  xml_attribute<> * attr;
  xml_node<> * child;
  std::string matName, modelPath;
```

```
  std::vector<std::string> res;
  char * value;
  Transform * tr = new Transform();
  ObjectGroup * og;

  if((attr = node->first_attribute("mat")) && strcmp(attr->value(),"") != 0){
    matName = attr->value();
  }
  if(strcmp((value = node->value()),"")){
    parseTransform(value, tr);
  }

  if((attr = node->first_attribute("model")) && strcmp(attr->value(),"") != 0){
    modelPath = attr->value();
    if(matName != ""){
      og = importModel(modelPath, w->materials[matName], tr);
    }
    else{
      std::cout << "[Warning] <Group \"" << attr->name() << "\"> attribute without
          Material! Used default value.\n";
      og = importModel(modelPath, w->materials["lambert"], tr);
    }
    return og;
  }
  else if((child = node->first_node("Object"))){
    og = new ObjectGroup(tr);
    Primitive * pr = traverseObjectNode(child, matName);
    if(pr)
      og->addObject(pr);
    while((child = child->next_sibling("Object"))){
      pr = traverseObjectNode(child, matName);
      if(pr)
        og->addObject(pr);
    }
    return og;
  }
  else if((child = node->first_node("Group"))){
    og = new ObjectGroup(tr);
    og->addChild(traverseObjectsNode(child));
    while((child = child->next_sibling("Group"))){
      og->addChild(traverseObjectsNode(child));
    }
    return og;
  }

}

Primitive * Parser::traverseObjectNode(rapidxml::xml_node<> * node, const
    std::string & mat){ // metodo che traversa un nodo xml di tipo Object
```

```
  xml_attribute<> * attr;
  std::string matName = mat;
  std::vector<std::string> res;
  char * value;

  if((attr = node->first_attribute("mat")) && strcmp(attr->value(),"") != 0){
    matName = attr->value();
  }

  if(strcmp((value = node->value()),"")){
    parseArray(value,res);
    if(res.size() < 4){
      std::cout << "[Warning] <" << node->name() << " \"center,radius\"> values
          size < 4. Object creation skipped.\n";
      return nullptr;
    }
    else{
      std::vector<double> n;
      for(auto & i : res){
        n.push_back(stod(i, nullptr));
      }
      if((attr = node->first_attribute("type")) && strcmp(attr->value(),"") != 0){
        if(strcmp(attr->value(),"sphere") == 0){
          if(matName != "")
            return new Sphere(Vec3d(n[0],n[1],n[2]), n[3], w->materials[matName]);
          else
            return new Sphere(Vec3d(n[0],n[1],n[2]), n[3], w->materials["lambert"]);
        }
      }
      else{
        std::cout << "[Warning] <" << node->name() << " \"type\"> value unknow.
            Object creation skipped.\n";
        return nullptr;
      }
    }
  }
  else{
    std::cout << "[Warning] <" << node->name() << " \"center,radius\"> values
        unknow. Object creation skipped.\n";
    return nullptr;
  }
}


bool Parser::traverseMaterialsNode(xml_node<> * node){ // metodo che traversa un
    nodo xml di tipo Material
  xml_attribute<> * attr;
  xml_node<> * nextNode, * brdfs;
  std::vector<std::string> matsNames;
```

```
std::string reflMapName, emisMapName, normMapName;
Material * mat;
Vec3d refl, emis;

if((attr = node->first_attribute("name")) && strcmp(attr->value(),"") != 0){
  parseArray(attr->value(), matsNames);
}
else{
  std::cout << "[Warning] <Material \"" << attr->name() << "\"> attribute
      missing! Material creation skipped.\n";
  return false;
}

if((attr = node->first_attribute("refl")) && strcmp(attr->value(),"") != 0){
  std::vector<std::string> val;
  parseArray(attr->value(), val);
  if(val.size() < 2)
    refl = Vec3d(stod(val[0], nullptr));
  else if(val.size() > 2)
    refl = Vec3d(stod(val[0], nullptr), stod(val[1], nullptr), stod(val[2],
        nullptr));
}
if((attr = node->first_attribute("reflMap")) && strcmp(attr->value(),"") != 0){
  reflMapName = attr->value();
}
if((attr = node->first_attribute("emis")) && strcmp(attr->value(),"") != 0){
  std::vector<std::string> val;
  parseArray(attr->value(), val);
  if(val.size() < 2)
    emis = Vec3d(stod(val[0], nullptr));
  else if(val.size() > 2)
    emis = Vec3d(stod(val[0], nullptr), stod(val[1], nullptr), stod(val[2],
        nullptr));
}
if((attr = node->first_attribute("emisMap")) && strcmp(attr->value(),"") != 0){
  emisMapName = attr->value();
}
if((attr = node->first_attribute("normMap")) && strcmp(attr->value(),"") != 0){
  normMapName = attr->value();
}


if((brdfs = node->first_node("Brdfs"))){
  if((attr = brdfs->first_attribute("type")) && strcmp(attr->value(),"") != 0){
    if(strcmp(attr->value(),"mix") == 0){

    }
    else{
      AddBrdf * mBrdf = new AddBrdf();
```

```
        traverseBrdfsNode(brdfs->first_node("Brdf"), mBrdf);
        mat = new Material(mBrdf);
      }
    }
    else{
      AddBrdf * mBrdf = new AddBrdf();
      std::cout << "[Warning] <Brdfs \"" << attr->name() << "\"> attribute missing!
          Used default value.\n";
      traverseBrdfsNode(brdfs->first_node("Brdf"), mBrdf);
      mat = new Material(mBrdf);
    }
  }
  else{
    std::cout << "[Error] <Material> without <Brdfs> node.\n";
    return false;
  }

  if(refl != Vec3d())
    mat->setReflection(refl);
  else if(reflMapName != "")
    mat->setReflMap(reflMapName);

  if(emis != Vec3d())
    mat->setEmission(emis);
  else if(emisMapName != "")
    mat->setEmisMap(emisMapName);

  if(normMapName != "")
    mat->setNormalMap(normMapName);

  for(auto & name : matsNames)
    w->materials[name] = mat;

  if((nextNode = node->next_sibling())){
    return traverseMaterialsNode(nextNode);
  }
  else
    return false;


}

bool Parser::traverseBrdfsNode(xml_node<> * node, MultipleBrdf * mBrdf){ // metodo
    che traversa un nodo xml di tipo Brdfs

  xml_node<> * nextNode;
  xml_attribute<> * attr;
  char * value;
  std::vector<std::string> res;
```

```
  Texture * txt = nullptr;

  if((attr = node->first_attribute("type")) && strcmp(attr->value(),"") != 0){
    if(strcmp(attr->value(),"lambert") == 0){
      if(strcmp((value = node->value()),"")){
        parseArray(value, res);
        if((attr = node->first_attribute("map")) && strcmp(attr->value(),"") != 0){
          if((txt = new Texture(attr->value()))){
            if(res.size() > 0){
              mBrdf->addBrdf(new Lambert(txt, stod(res[0],nullptr)));
            }
            else{
              mBrdf->addBrdf(new Lambert(txt));
            }
          }
          else{
            std::cout << "[Warning] <" << node->name() << " \"map\"> attribute
                error!.\n";
          }
        }
        else if(res.size() > 3){
          mBrdf->addBrdf(new Lambert(Vec3d(stod(res[0],nullptr),
              stod(res[1],nullptr), stod(res[2],nullptr)), stod(res[3],nullptr)));
        }
        else if(res.size() > 1 ){
          mBrdf->addBrdf(new Lambert (Vec3d(stod(res[0],nullptr)),
              stod(res[1],nullptr)));
        }
        else{
          std::cout << "[Warning] <" << node->name() << " \"albedo,kd\"> values
              size != 4! Used default value.\n";
          mBrdf->addBrdf(new Lambert(Vec3d(.5),1));
        }
      }
      else{
        std::cout << "[Warning] <" << node->name() << " \"albedo,kd\"> values
            missing! Used default.\n";
        mBrdf->addBrdf(new Lambert(Vec3d(.5),1));
      }
    }
    else if(strcmp(attr->value(),"blinn") == 0){
      if(strcmp((value = node->value()),"")){
        parseArray(value, res);
        if((attr = node->first_attribute("map")) && strcmp(attr->value(),"") != 0){
          if((txt = new Texture(attr->value()))){
            if(res.size() > 1 ){
              mBrdf->addBrdf(new Blinn(txt, stod(res[0],nullptr),
                  stoi(res[1],nullptr, 10)));
            }
```

```
        else if(res.size() > 0 ){
          mBrdf->addBrdf(new Blinn(txt, stoi(res[0],nullptr, 10)));
        }
        else{
          std::cout << "[Warning] <" << node->name() << " \"alpha\"> value
              required! Used default value.\n";
          mBrdf->addBrdf(new Blinn(txt,1000));
        }
      }
      else{
        std::cout << "[Warning] <" << node->name() << " \"map\"> attribute
            error!.\n";
      }
    }
    else if(res.size() > 4 ){
      mBrdf->addBrdf(new Blinn(Vec3d(stod(res[0],nullptr),
          stod(res[1],nullptr), stod(res[2],nullptr)), stod(res[3],nullptr),
          stod(res[4],nullptr)));
    }
    else if(res.size() > 2 ){
      mBrdf->addBrdf(new Blinn(Vec3d(stod(res[0],nullptr)),
          stod(res[1],nullptr), stod(res[2],nullptr)));
    }
    else{
      std::cout << "[Warning] <" << node->name() << " \"specular,ks,alpha\">
          values size != 4! Used default value.\n";
      mBrdf->addBrdf(new Blinn(Vec3d(.5),1000));
    }
  }
  else{
    std::cout << "[Warning] <" << node->name() << " \"specular,ks,alpha\">
        values missing! Used default.\n";
    mBrdf->addBrdf(new Blinn(Vec3d(.5),1000));
  }
}
else if(strcmp(attr->value(),"cooktorrance") == 0){
  if(strcmp((value = node->value()),""))
    parseArray(value, res);
  if((attr = node->first_attribute("sMap")) && strcmp(attr->value(),"") != 0){
    if((txt = new Texture(attr->value()))){
      if((attr = node->first_attribute("rMap")) && strcmp(attr->value(),"") !=
          0){
        Texture * txt2;
        if((txt2 = new Texture(attr->value()))){
          if(res.size() > 1)
            mBrdf->addBrdf(new CookTorrance(txt, txt2, stod(res[0],nullptr),
                stod(res[1],nullptr)));
          else
            mBrdf->addBrdf(new CookTorrance(txt, txt2, 1., 1.));
```

```
        }
      }
      else if(res.size() > 1 ){
        mBrdf->addBrdf(new CookTorrance(txt, stod(res[0],nullptr),
            stod(res[1],nullptr)));
      }
      else if(res.size() > 0 ){
        mBrdf->addBrdf(new CookTorrance(txt, stod(res[0],nullptr), 1.));
      }
      else{
        std::cout << "[Warning] <" << node->name() << " \"roughness\"> value
            required! Used default value.\n";
        mBrdf->addBrdf(new CookTorrance(txt,1, 1));
      }
    }
    else{
      std::cout << "[Warning] <" << node->name() << " \"sMap\"> attribute
          error!.\n";
    }
  }
  else if((attr = node->first_attribute("rMap")) && strcmp(attr->value(),"") !=
      0){
    if((txt = new Texture(attr->value()))){
      if((attr = node->first_attribute("sMap")) && strcmp(attr->value(),"") !=
          0){
        Texture * txt2;
        if((txt2 = new Texture(attr->value()))){
          if(res.size() > 1)
            mBrdf->addBrdf(new CookTorrance(txt, txt2, stod(res[0],nullptr),
                stod(res[1],nullptr)));
          else
            mBrdf->addBrdf(new CookTorrance(txt, txt2, 1., 1.));
        }
      }
      else if(res.size() > 1 ){
        mBrdf->addBrdf(new CookTorrance(stod(res[0],nullptr), txt,
            stod(res[1],nullptr)));
      }
      else if(res.size() > 0 ){
        mBrdf->addBrdf(new CookTorrance(stod(res[0],nullptr), txt, 1.));
      }
      else{
        std::cout << "[Warning] <" << node->name() << " \"roughness\"> value
            required! Used default value.\n";
        mBrdf->addBrdf(new CookTorrance(1,txt, 1.));
      }
    }
    else{
```

```
          std::cout << "[Warning] <" << node->name() << " \"rMap\"> attribute
              error!.\n";
        }
      }
      else if(res.size() > 3 ){
        mBrdf->addBrdf(new CookTorrance(Vec3d(stod(res[0],nullptr),
            stod(res[1],nullptr), stod(res[2],nullptr)), stod(res[3],nullptr)));
      }
      else if(res.size() > 1 ){
        mBrdf->addBrdf(new CookTorrance(Vec3d(stod(res[0],nullptr)),
            stod(res[1],nullptr)));
      }
      else{
        std::cout << "[Warning] <" << node->name() << " \"specular,roughness\">
            values size != 4! Used default value.\n";
        mBrdf->addBrdf(new CookTorrance(Vec3d(.5),1000));
      }
    }
    else{
      std::cout << "[Error] <Brdf \"type\"> unknow.\n";
      return false;
    }

    if((nextNode = node->next_sibling())){
      return traverseBrdfsNode(nextNode, mBrdf);
    }
    else
      return false;

  }
}

bool Parser::traverseLightsNode(rapidxml::xml_node<> * node){ // metodo che
    traversa un nodo xml di tipo Light
  xml_attribute<> * attr;
  char * value;
  std::vector<std::string> res;

  if(strcmp((value = node->value()),"")){
    parseArray(value, res);
  }
  else{
    std::cout << "[Error] <Light> values unknow.\n";
    return false;
  }

  if((attr = node->first_attribute("type")) && strcmp(attr->value(),"") != 0){
    if(res.size() < 7){
      std::cout << "[Error] <Light> values size < 7.\n";
```

```
      return false;
    }
    else{
      if(strcmp(attr->value(),"point") == 0){
        w->addLight(new PointLight(stod(res[0],nullptr),
            Vec3d(stod(res[1],nullptr), stod(res[2],nullptr), stod(res[3],nullptr)),
            Vec3d(stod(res[4],nullptr), stod(res[5],nullptr),
            stod(res[6],nullptr))));
        return true;
      }
      else if(strcmp(attr->value(),"directional") == 0){
        w->addLight(new DirectionalLight(stod(res[0],nullptr),
            Vec3d(stod(res[1],nullptr), stod(res[2],nullptr), stod(res[3],nullptr)),
            Vec3d(stod(res[4],nullptr), stod(res[5],nullptr),
            stod(res[6],nullptr))));
        return true;
      }
      else{
        std::cout << "[Error] <Light \"type\"> unknow.\n";
        return false;
      }
    }
  }
}


void Parser::parseSceneNode(xml_node<> * node){ // metodo che parsa un nodo
    pricipale della scena xml
  std::string name = node->name();
  xml_attribute<> * attr;
  xml_node<> * child;
  char * value;
  std::vector<std::string> res;


  if(name == "Film"){
    int width, heigth;
    std::string path, fileName;

    if((attr = node->first_attribute("width")) && strcmp(attr->value(),"") != 0){
      width = std::stoi(attr->value(), nullptr, 10);
    }
    else{
      std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
          missing! Used default value.\n";
      width = 800;
    }
    if((attr = node->first_attribute("heigth")) && strcmp(attr->value(),"") != 0){
      heigth = std::stoi(attr->value(), nullptr, 10);
```

```
  }
  else{
    std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
        missing! Used default value.\n";
    heigth = 600;
  }
  if((attr = node->first_attribute("path")) && strcmp(attr->value(),"") != 0){
    path = attr->value();
  }
  else{
    std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
        missing! Used default value.\n";
    path = "";
  }
  if(strcmp((value = node->value()),"")){
    fileName = value;
    parseString(fileName);
  }
  else{
    std::cout << "[Warning] <" << name << " \"filename\"> value missing! Used
        default value.\n";
    fileName = "rendering.bmp";
  }
  w->film = new Film(width, heigth, 3, path+fileName);
}
else if(name == "Camera"){
  double near, far;
  float fov;
  Vec3f eye, lookAt, up;
  if((attr = node->first_attribute("near")) && strcmp(attr->value(),"") != 0){
    near = std::stod(attr->value(), nullptr);
  }
  else{
    std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
        missing! Used default value.\n";
    near = 10e-5;
  }
  if((attr = node->first_attribute("far")) && strcmp(attr->value(),"") != 0){
    far = std::stod(attr->value(), nullptr);
  }
  else{
    std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
        missing! Used default value.\n";
    far = 10e5;
  }
  if((attr = node->first_attribute("fov")) && strcmp(attr->value(),"") != 0){
    fov = std::stof(attr->value(), nullptr);
  }
  else{
```

```
      std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
          missing! Used default value.\n";
      fov = 90;
    }
    if(strcmp((value = node->value()),"")){
      parseArray(value, res);
      if(res.size() < 9){
        std::cout << "[Warning] <" << name << " \"eye, lookAt, up\"> values size !=
            9! Used default value.\n";
        eye = Vec3f(50, 0, 0); lookAt = Vec3f(0); up = Vec3f(0, 0, 1);
      }
      else{
        eye = Vec3f(stof(res[0],nullptr), stof(res[1],nullptr),
            stof(res[2],nullptr));
        lookAt = Vec3f(stof(res[3],nullptr), stof(res[4],nullptr),
            stof(res[5],nullptr));
        up = Vec3f(stof(res[6],nullptr), stof(res[7],nullptr),
            stof(res[8],nullptr));
      }
    }
    else{
      std::cout << "[Warning] <" << name << " \"eye, lookAt, up\"> values missing!
          Used default value.\n";
      eye = Vec3f(50, 0, 0); lookAt = Vec3f(0); up = Vec3f(0, 0, 1);
    }
    if((attr = node->first_attribute("type"))){
      if(strcmp(attr->value(),"orthographic") == 0)
        w->cam = new OrthoCamera(w->film->width, w->film->heigth, near, far, fov,
            eye, lookAt, up);
      else
        w->cam = new PerspCamera(w->film->width, w->film->heigth, near, far, fov,
            eye, lookAt, up);
    }
    else{
      std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
          missing! Used default value.\n";
      w->cam = new PerspCamera(w->film->width, w->film->heigth, near, far, fov,
          eye, lookAt, up);
    }
  }
  else if(name == "Sampler"){
    int samples;
    if((attr = node->first_attribute("samples")) && strcmp(attr->value(),"") != 0){
      samples = std::stoi(attr->value(), nullptr, 10);
    }
    else{
      std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
          missing! Used default value.\n";
      samples = 4;
```

```
  }
  if((attr = node->first_attribute("type"))){
    if(strcmp(attr->value(),"multijitter") == 0)
      w->sampler = new MultijitterSampler(samples);
    else if(strcmp(attr->value(),"hammerslay") == 0)
      w->sampler = new HammerslaySampler(samples);
    else if(strcmp(attr->value(),"supersampler") == 0)
      w->sampler = new SuperSampler(samples);
    else if(strcmp(attr->value(),"random") == 0)
      w->sampler = new RandomSampler(samples);
    else
      w->sampler = new RegularSampler();
  }
  else{
    std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
        missing! Used default value.\n";
    w->sampler = new MultijitterSampler(samples);
  }
}
else if(name == "Accelerator"){
  int gen, prim;
  if((attr = node->first_attribute("gen")) && strcmp(attr->value(),"") != 0){
    gen = std::stoi(attr->value(), nullptr, 10);
  }
  else{
    std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
        missing! Used default value.\n";
    gen = 20;
  }
  if((attr = node->first_attribute("prim")) && strcmp(attr->value(),"") != 0){
    prim = std::stoi(attr->value(), nullptr, 10);
  }
  else{
    std::cout << "[Warning] <" << name << " \"" << attr->name() << "\"> attribute
        missing! Used default value.\n";
    prim = 50;
  }
  w->turbo = new Accelerator(gen, prim, w);
}
else if(name == "Integrator"){
  int depth = 2;
  if((attr = node->first_attribute("depth")) && strcmp(attr->value(),"") != 0){
    depth = std::stoi(attr->value(), nullptr, 10);
  }
  if((attr = node->first_attribute("type")) && strcmp(attr->value(),"") != 0){
    if(strcmp(attr->value(),"whitted") == 0)
      w->integrator = new WhittedIntegrator(w, depth);
    else if(strcmp(attr->value(),"pathtracing") == 0)
      w->integrator = new PathTracingIntegrator(w, depth);
```

```
      else if(strcmp(attr->value(),"ambientocclusion") == 0)
        w->integrator = new AmbientOcclusionIntegrator(w);
      else
        w->integrator = new WhittedIntegrator(w, 2);
    }
    else{
      w->integrator = new WhittedIntegrator(w, 2);
    }
  }
  else if(name == "Materials"){
    w->materials["lambert"] = new Material(new AddBrdf(new Lambert()));
    if((child = node->first_node("Material"))){
      traverseMaterialsNode(child);
    }
    else{
      std::cout << "[Warning] No <Material> found in <" << node->name() << ">" <<
          "\n";
    }
  }
  else if(name == "Objects"){
    if((child = node->first_node("Group"))){
      ObjectGroup * og = traverseObjectsNode(child);
      og->toWorld(nullptr);
      og->setBVertex();
      w->addModel(og);
      while((child = child->next_sibling("Group"))){
        og = traverseObjectsNode(child);
        og->toWorld(nullptr);
        og->setBVertex();
        w->addModel(og);
      }
    }
    else{
      std::cout << "[Warning] No <Group> found in <" << node->name() << ">" << "\n";
    }
  }
  else if(name == "Lights"){
    if((child = node->first_node("Light"))){
      traverseLightsNode(child);
      while((child = child->next_sibling("Light"))){
        traverseLightsNode(child);
      }
    }
  }
  else if(name == "Environment"){
    if((attr = node->first_attribute("map")) && strcmp(attr->value(),"") != 0){
      w->env = new Environment(attr->value());
    }
    else if(strcmp((value = node->value()),"")){
```

```
      parseArray(value, res);
      if(res.size() < 3){
        std::cout << "[Warning] <" << name << " \"color\"> array size != 3! Used
            default value.\n";
        w->env = new Environment();
      }
      else{
        w->env = new Environment(Vec3d(stod(res[0], nullptr), stod(res[1],
            nullptr), stod(res[2], nullptr)));
      }
    }
    else{
      std::cout << "[Warning] <" << name << " \"map\"> or color values missing!
          Used default value.\n";
      setDefault(name);
    }
  }


}

void Parser::setDefault(const std::string & name){
  if(name == "Film"){
    w->film = new Film(800, 600, 3);
  }
  else if(name == "Camera"){
    w->cam = new
        PerspCamera(w->film->width,w->film->heigth,1e-5,1e5,90,Vec3f(50,0,0),
        Vec3f(0,0,0), Vec3f(0,0,1));
  }
  else if(name == "Sampler"){
    w->sampler = new RegularSampler();
  }
  else if(name == "Accelerator"){
    w->turbo = new Accelerator(5, 35, w);
  }
  else if(name == "Integrator"){

  }
  else if(name == "Environment"){
    w->env = new Environment();
  }
}

void Parser::tokenize(const std::string & str, const std::string & delim,
    std::vector<std::string> & parts) {
  size_t start, end = 0;
  while (end < str.size()) {
    start = end;
```

```
    while (start < str.size() && (delim.find(str[start]) != std::string::npos)) {
      start++; // skip initial whitespace
    }
    end = start;
    while (end < str.size() && (delim.find(str[end]) == std::string::npos)) {
      end++; // skip to end of word
    }
    if (end-start != 0) { // just ignore zero-length strings.
      parts.push_back(std::string(str, start, end-start));
    }
  }
}

void Parser::parseArray(const std::string & s, std::vector<std::string> &
    parseRes){
  // char * val = strtok(s, ", \n");
  // while(val != nullptr){
  //   parseRes.push_back(val);
  //   val = strtok(nullptr, ", \n");
  // }
  std::string str = s;
  tokenize(str, ", \n", parseRes);
}


void Parser::parseTransform(const std::string & _s, Transform * tr){ // metodo che
    parsa una trasformazione spaziale
  std::vector<std::string> strs;
  std::string s = _s;

  parseString(s);
  if(s[0] == '*'){
    tokenize(s, " *", strs);
    for(auto & str : strs){
      u_int pos1 = str.find('(');
      u_int n = str.find(')') - pos1 - 1;
      std::string sdatas = str.substr(pos1+1, n);
      std::vector<std::string> vdatas;
      parseArray(sdatas,vdatas);
      std::string type = str.substr(0,2);
      if( type == "tr"){
        if(vdatas.size() < 3){
          std::cout << "[ERROR] Translation datas incorrect. Creation skipped.\n";
        }
        else
        tr->operator*=(translation(stod(vdatas[0], nullptr), stod(vdatas[1],
            nullptr), stod(vdatas[2])));
      }
      else if(type == "sc"){
```

```
      if(vdatas.size() < 1){
        std::cout << "[ERROR] Scaling datas incorrect. Creation skipped.\n";
      }
      else if(vdatas.size() < 3)
      tr->operator*=(scaling(stod(vdatas[0], nullptr)));
      else if(vdatas.size() > 2)
      tr->operator*=(scaling(stod(vdatas[0], nullptr), stod(vdatas[1], nullptr),
          stod(vdatas[2])));
    }
    else if(type == "rx"){
      if(vdatas.size() < 1){
        std::cout << "[ERROR] RotationX datas incorrect. Creation skipped.\n";
      }
      else
      tr->operator*=(rotationX(stod(vdatas[0], nullptr)));
    }else if(type == "ry"){
      if(vdatas.size() < 1){
        std::cout << "[ERROR] RotationY datas incorrect. Creation skipped.\n";
      }
      else
      tr->operator*=(rotationY(stod(vdatas[0], nullptr)));
    }else if(type == "rz"){
      if(vdatas.size() < 1){
        std::cout << "[ERROR] RotationZ datas incorrect. Creation skipped.\n";
      }
      else
      tr->operator*=(rotationZ(stod(vdatas[0], nullptr)));
    }
    else
    std::cout << "[ERROR] Transform type unknow. Creation skipped.\n";
  }
 }
 else{
   parseArray(s,strs);
   if(strs.size() < 12){
     std::cout << "[ERROR] Transform datas size =! 16. Used default value.\n";
   }
   else{
     std::vector<double> n;
     for(auto & i : strs){
       n.push_back(stod(i, nullptr));
     }
     tr->operator*=(Transform(Vec3d(n[0],n[4],n[8]), Vec3d(n[1],n[5],n[9]),
         Vec3d(n[2],n[6],n[10]), Vec3d(n[3],n[7],n[11])));
   }
 }


}
```

```
void Parser::parseString(std::string & str){
  std::vector<std::string> res;
  tokenize(str, " \n\t\r\v", res);
  str = "";
  for(auto & i : res)
    str += i;
}
```

parser.hpp:

```
#ifndef _PARSER_
#define _PARSER_

#include <string>
#include <unordered_map>
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
#include "../../otherLibs/rapidxml-1.13/rapidxml.hpp"
#include "../../otherLibs/rapidxml-1.13/rapidxml_print.hpp"

class Material;
class Texture;
class Transform;
class World;
class ObjectGroup;
class MultipleBrdf;
class Primitive;

class Parser{
public:
  Parser(World *);
  Parser(const Parser &);
  ~Parser();

  Parser & operator=(const Parser &);

  bool parseScene(const std::string &);
  ObjectGroup * importModel(const std::string &, Material *, Transform *);
  bool importMesh(const aiMesh *, ObjectGroup *, Material *);
  bool traverseModelNode(const aiNode *, const aiScene *, ObjectGroup *, Material
      *);
  ObjectGroup * traverseObjectsNode(rapidxml::xml_node<> *);
  Primitive * traverseObjectNode(rapidxml::xml_node<> *, const std::string &);
  bool traverseMaterialsNode(rapidxml::xml_node<> *);
  bool traverseBrdfsNode(rapidxml::xml_node<> *, MultipleBrdf *);
  bool traverseLightsNode(rapidxml::xml_node<> *);
  void parseSceneNode(rapidxml::xml_node<> *);
```

```
  void setDefault(const std::string &);
  void tokenize(const std::string &, const std::string &, std::vector<std::string>
      &);
  void parseArray(const std::string &, std::vector<std::string> &);
  void parseTransform(const std::string &, Transform *);
  void parseString(std::string &);

  World * w; // riferimento al World
  Assimp::Importer * imp; // riferimento all'importer di ASSIMP
  rapidxml::xml_document<> scene; // nodo radice della scena
  std::vector<std::string> parseList; // lista della procedura di parsing
};

#endif
```

pixelBuffer.hpp:

```
/*
  File di definizione della classe PixelBuffer, modella una generica matrice di
      pixel a valori [0,255]
*/

#ifndef _PIXEL_BUFFER_
#define _PIXEL_BUFFER_

#include "../geometry/vec3d.hpp"
#include "../utilities/math.hpp"
#include <SOIL/SOIL.h>
#include <iostream>


class PixelBuffer{
public:
  PixelBuffer(): width(0), heigth(0), channels(0), map(nullptr){}
  PixelBuffer(int w, int h, int c): width(w), heigth(h), channels(c){
    map = new unsigned char [size()];
    for(int i = 0; i < size(); ++i)
      map[i] = 0;
  }
  PixelBuffer(const std::string & path){
    load(path);
  }
  virtual ~PixelBuffer(){}

  int size(){ // ritorna la grandezza effettiva dell'array di pixel
    return heigth*width*channels;
  }

  int area(){ // ritorna la risoluzione dell'immagine
```

```
    return heigth*width;
  }

  virtual bool load(const std::string & path){ // carica un'immagine in memoria
      tramite la libreria SOIL
    map = SOIL_load_image(path.c_str(), &width, &heigth, &channels, SOIL_LOAD_AUTO);
    if(map){
      std::cout << "\t<Map>\n\t\tSuccesfully imported " << path << ", Size: " <<
          width << "x" << heigth
      << ", channels: " << channels << "\n\t</Map>\n";
      return true;
    }
    else{
      std::cout << "\t\t[ERROR] File reading failed, maybe invalid path?\n";
      return false;
    }
  }

  virtual std::string toString() const{
    return "[ Res: " + std::to_string(width) + "x" + std::to_string(heigth);
  }

  int width, heigth, channels; // larghezza, altezza e numero di canali
  unsigned char * map; // array di valori tra 0 e 255
};

#endif
```

stats.hpp:

```
#ifndef _STATS_
#define _STATS_

#include "timer.hpp"
#include "../world.hpp"
#include "../utilities/film.hpp"
#include "../geometricObjects/objectGroup.hpp"

class Stats{
public:
  Stats(World * _w): timer(new Timer), w(_w){}
  ~Stats(){
    delete timer;
  }

  void initialize(){
    for(auto & x : w->models)
      totalPrims += x->nTotPrims;
    totalHits = w->film->width*w->film->heigth*totalPrims;
```

```
    rays = totalPixel = w->film->width*w->film->heigth;
    step = 20;
    increment = totalPixel/step;
  }

  void openProgressBar() const{
    std::cout << "\t0\%";
    for(int i = 0; i < step/2 - 2; ++i)
      std::cout << " ";
    std::cout << "50\%";
    for(int i = 0; i < step/2 - 3; ++i)
      std::cout << " ";
    std::cout << "100\%\n\t[";
    std::cout.flush();
  }

  void closeProgressBar() const{
    std::cout << "]\n";
  }

  void printStats() const{
    std::cout << "\tTempo di rendering: "; timer->print(); std::cout << " ms\n";
    std::cout << "\tNumero di intersezioni teoriche: " << totalHits << "\n";
    std::cout << "\tNumero di intersezioni lanciate: " << hits << "\n";
    std::cout << "\tNumero di raggi tracciati: " << rays << "\n";
    std::cout << "\tNumero di primitive: " << totalPrims << "\n";
    std::cout << "\tRapporto: " << totalHits/hits << "\n";
  }

  void next(){
    ++current;
  }

  void checkProgress(){
    if(current > progress){
      std::cout << "=";
      std::cout.flush();
      progress += increment;
    }
  }

  double hits, totalHits;
  double rays, totalPrims = 0;
  double totalPixel, increment, progress = 0, current = 0, step;
  Timer * timer;
  World * w;
};
```

```
#endif
```

texture.hpp:

```
/*
  File di definizione della classe Texture, specializzazione della classe
     PixelBuffer che modella una texture
*/

#ifndef _TEXTURE_
#define _TEXTURE_

#include "../geometry/vec3d.hpp"
#include "../utilities/math.hpp"
#include <iostream>

#include "pixelBuffer.hpp"

class Texture: public PixelBuffer{
public:
  Texture(): PixelBuffer(){}
  Texture(const std::string & path): PixelBuffer(path){}
  ~Texture(){
    SOIL_free_image_data(map);
  }

  Vec3d getValue(const Vec3d & uv){ // campiona la texture tramite il vettore uv
    int pos = static_cast<int>(uv.y*heigth)*width + static_cast<int>(uv.x*width);
        // calcolo l'indice monodimensionale
    pos *= channels; // lo scalo per il numero di canali
    return Vec3d(to01(map[pos], map[pos+1], map[pos+2])); // ritorno il vettore di
        valori rgb
  }

  bool load(const std::string & path){ // carica un'immagine dal percorso path
    PixelBuffer::load(path);
    return true;
  }

  std::string toString() const{
    return "Texture: " + PixelBuffer::toString() + "]";
  }

};

#endif
```

timer.hpp:

```
#ifndef _TIMER
#define _TIMER

#include <chrono>
#include <thread>
#include <iostream>

class Timer{
public:
  Timer(): t1(), t2(), dur(){}

  void start(){
    t1 = std::chrono::high_resolution_clock::now();
  }

  void stop(){
    t2 = std::chrono::high_resolution_clock::now();
    dur = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1);
  }

  void sleep(u_int s){
    std::this_thread::sleep_for(std::chrono::milliseconds(s));
  }

  void print(){
    std::cout << dur.count();
  }

  std::chrono::high_resolution_clock::time_point t1, t2;
  std::chrono::milliseconds dur;
};

#endif
```

CAPITOLO 19

Appendice: un secondo esempio di Path Tracer stocastico, nel linguaggio C++

Nell'Appendice 18, abbiamo presentato un esempio di path tracer basato su generazione stocastica di raggi riflessi, che utilizza pesantemente le strutture del linguaggio C++ . Per facilitare il lettore ora ne presentiamo una variante differente, resa più sintetica grazie al rinvio alla libreria grafica glm (OpenGL) di C++. La libreria grafica non è trascritta nel listato qui sotto. Il codice è stato elaborato e presentato da Giuseppe Giordano [**15**].

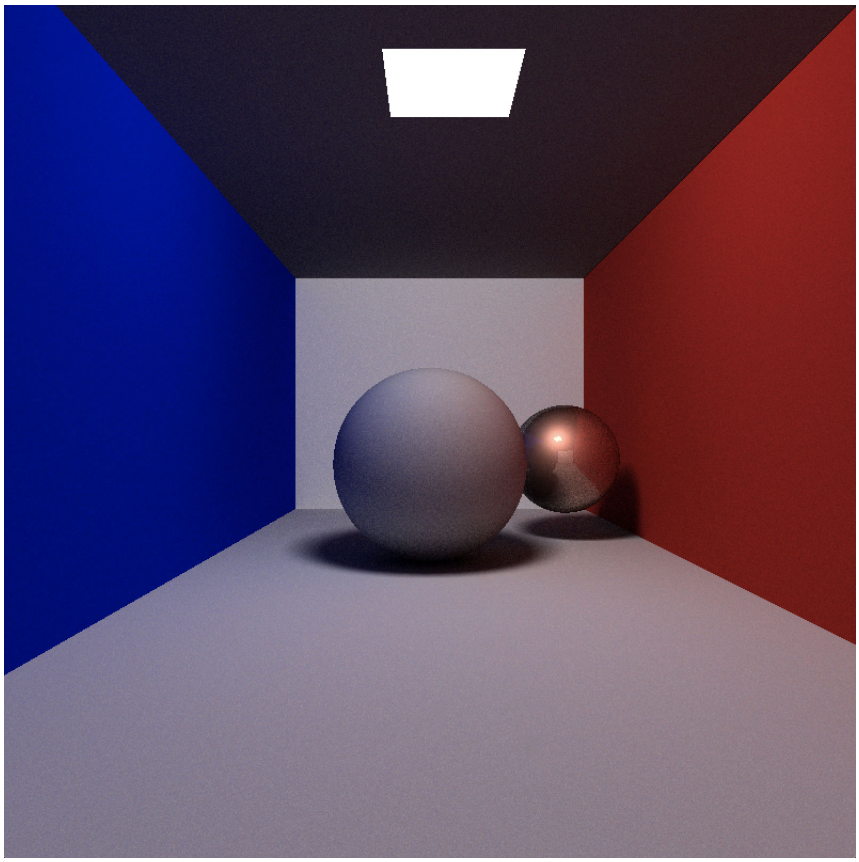


FIGURA 19.0.1. Cornell box con lampada quadrata centrale, sfera di gesso e sfera di rame, rese con ray tracer stocastico

---

main.cpp:

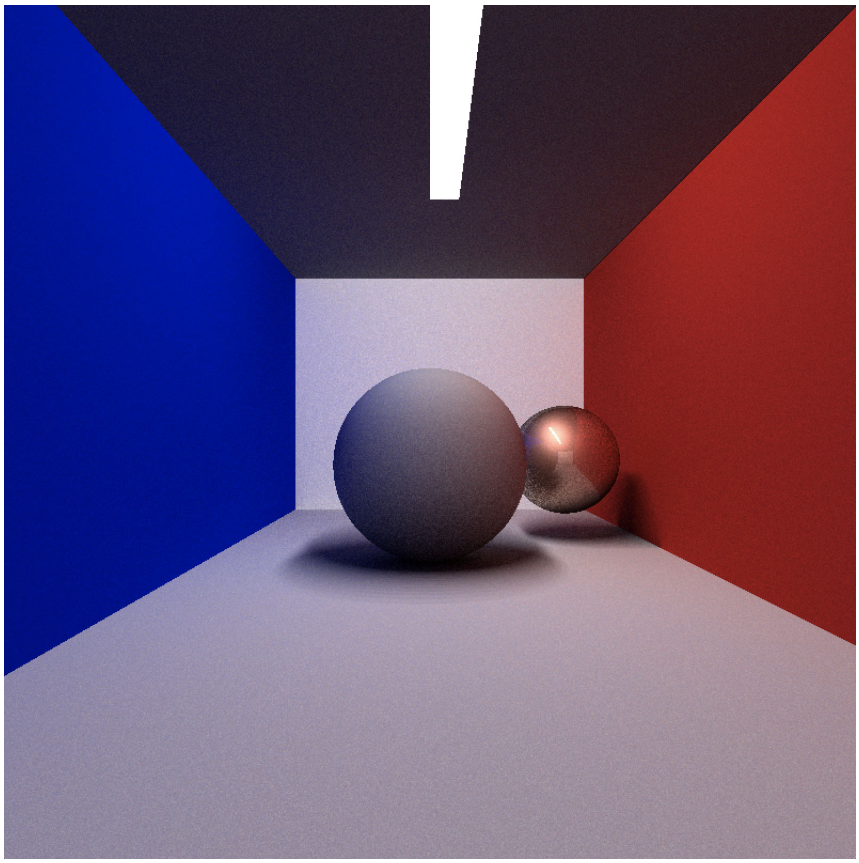


FIGURA 19.0.2. Sfera di gesso e sfera di rame, lampada disposta in profondità, rese con ray tracer stocastico

```
//
//  main.cpp
//  Progetto_Xcode
//
//  Created by Giuseppe Giordano on 18/12/2019.
//

/*

 Il programma genera una scena, una Cornell box:
 si creano due sfere, una totalemente diffusiva ed una riflettente.
 Aggiungiamo anche una clessidra, modificando l'equazione parametrica di una sfera.
 La scena e' visualizzata tramite un algoritmo di ray tracing stocastico:

 per l'illuminazione diretta si utilizza un ray tracing classico, che calcola il
     contributo del raggio d'ombra dal punto osservato ad un punto aleatorio scelto
     sulla sorgente di luce con probabilita' equidistribuita;
```

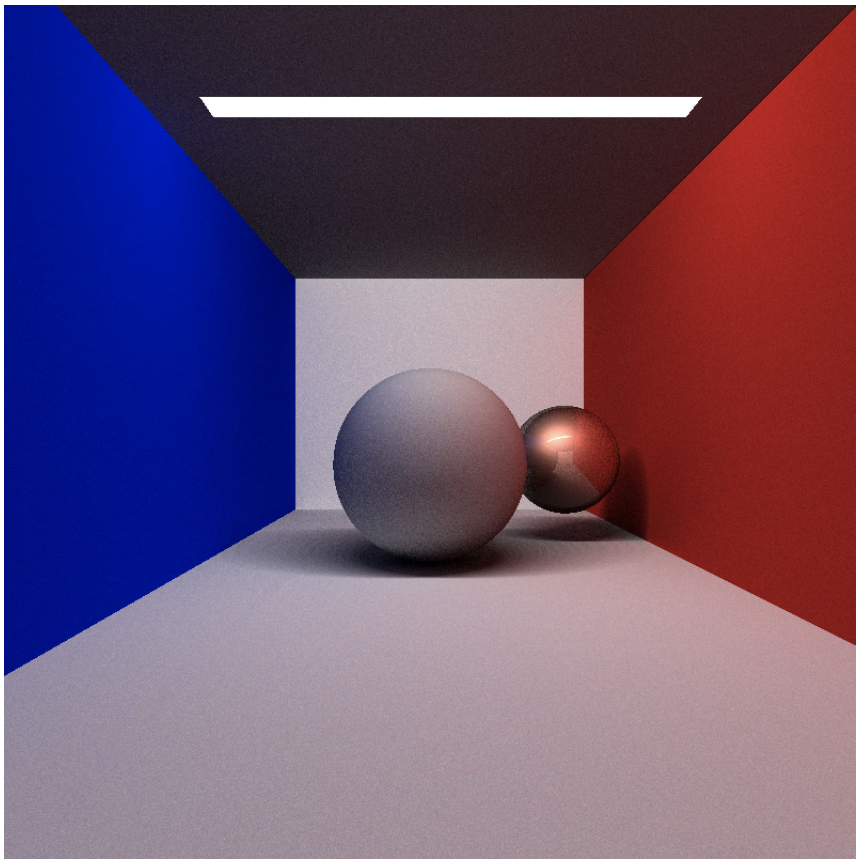


FIGURA 19.0.3. Sfera di gesso e sfera di rame, lampada disposta trasversalmente, rese con ray tracer stocastico

```
per l'illuminazione indiretta si utilizza un ray tracing stocastico, generando
    ricorsivamente nuovi raggi rimbalzo dopo rimbalzo con probabilita' date dalle
    rispettive brdf ai punti di rimbalzo, e calcolandone quindi i vari contributi
    con l'estimatore di Montecarlo.
Si utilizza il campionamento del coseno per le superficie diffusive, ma c'e'
    anche la possibilita' di usare un campionamento uniforme.
La ricorsione si ferma aleatoriamente con l'algoritmo della roulette russa, ma si
    puo' scegliere anche una soglia massima di rimbalzi.

La sfera riflettente e la clessidra sono gestite con la brdf di Cook-Torrance,
    mentre la sfera diffusiva solo con Lambert.
Questi parametri possono essere modificati proseguendo nel codice.

Le pareti della stanza sono gestite come fossero dei piani,
sono gestite tramite una brdf di Lambert.

Nel Main si possono modificare:
- dimensioni dell'immagine
```

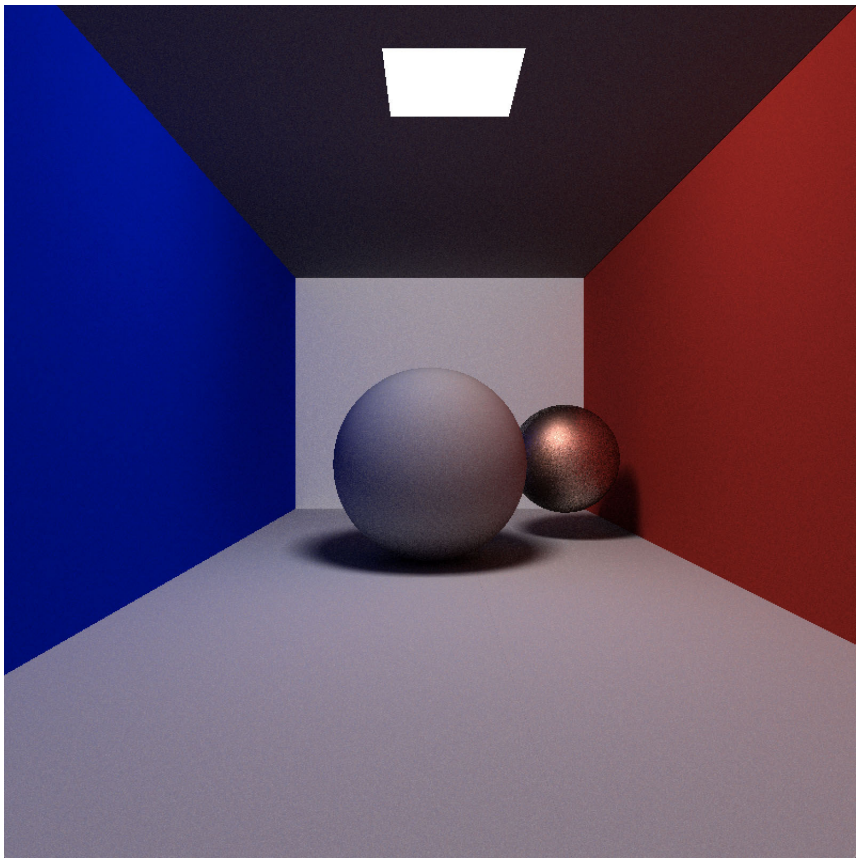


FIGURA 19.0.4. Sfera di gesso, sfera di rame corrugata con mappa di rilievo di Blinn, rese con ray tracer stocastico

```
 Nella classe Render:
 - numero di raggi d'ombra
 - numero di campioni per estimatore di montecarlo
 - numero di rimbalzi per illuminazione indiretta e per riflessioni
 - parametri dei vari oggetti della scena
 - se utilizzare campionamento uniforme o del coseno
 - se visualizzare la classidra di vetro

 Nella classe Sfera:
 - se visualizzare la mappa di rilievo con la formula di Blinn
 - parametri per la brdf di Cook-Torrance

 Nella classe Clessidra:
 - parametri per la brdf di Cook-Torrance
*/


#include <fstream>
#include <algorithm>
```

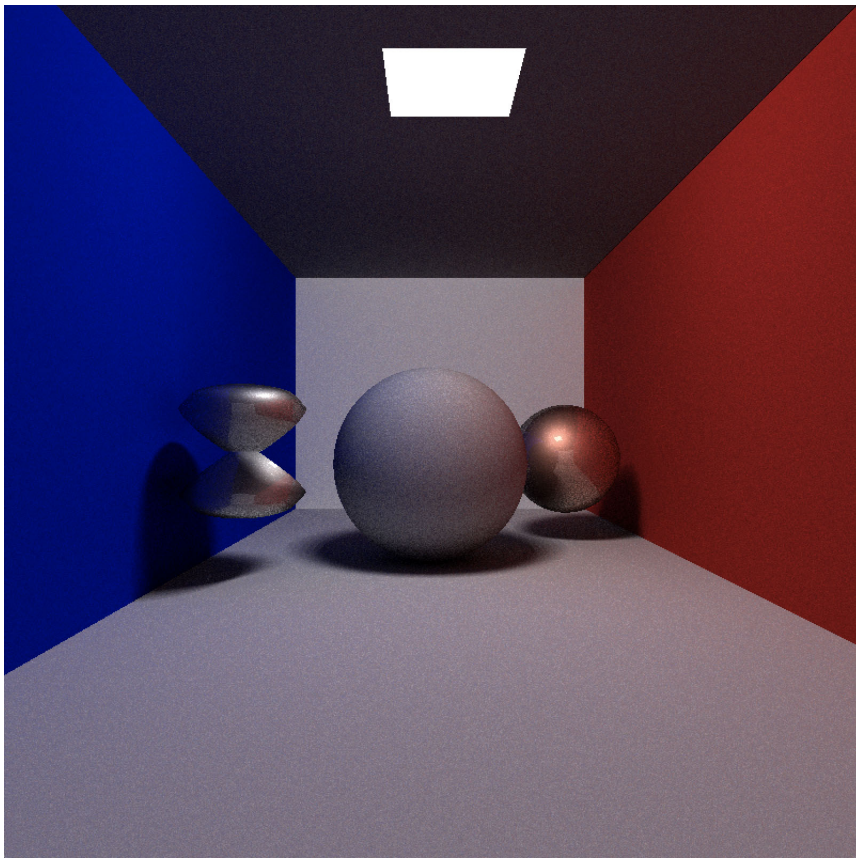


FIGURA 19.0.5. Sfera di gesso, sfera di rame e clessidra di argento, rese con ray tracer stocastico

```
#include "glm/glm.hpp"
#include "Ray.hpp"
#include <iostream>
#include <random>
#include "Render.hpp"

using namespace std;

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//DIMENSIONE IMMAGINE
const int width = 800;
const int height = 800;

int main()
{
    //Istanzio un oggetto della classe render
    Render* R = new Render();
```

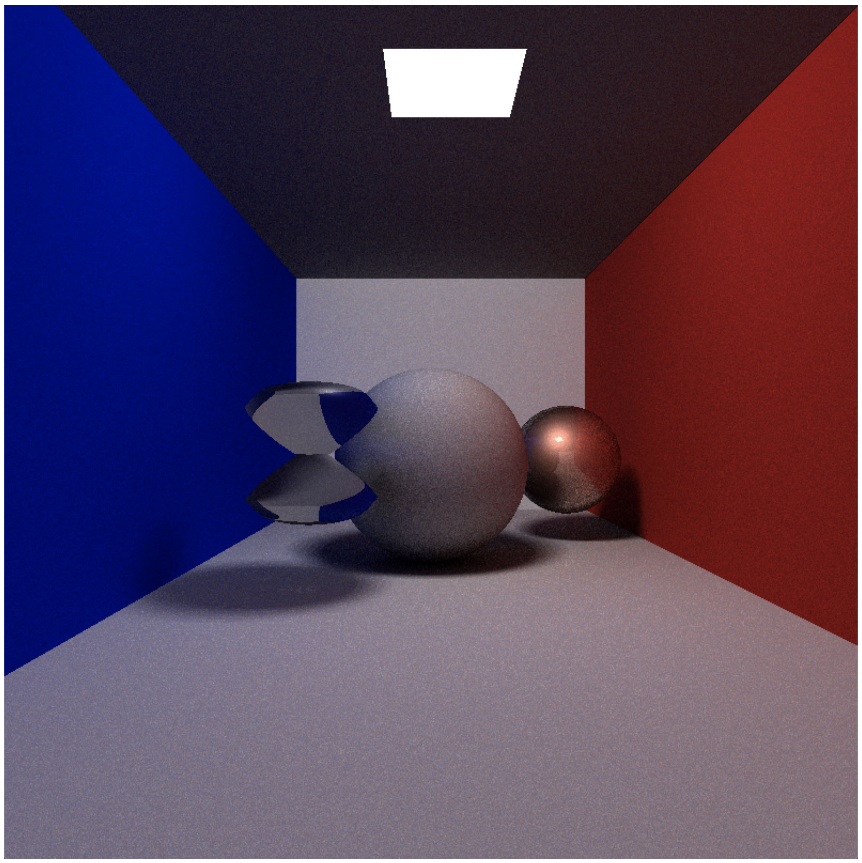


FIGURA 19.0.6. Sfera di gesso, sfera di rame e clessidra di vetro disposta lateralmente, rese con ray tracer stocastico

```
//Posizione osservatore
vec3 RayOrigin = vec3(0,0,0);

//Inizializazzione dell'immagine
vec3 **image = new vec3*[width];

//Salvataggio del clock iniziale
clock_t c_start = clock();

//Creazione immagine
image = R->computeImage(RayOrigin, width, height);

//Salvataggio del clock finale
clock_t c_end = clock();

//Calcolo della durata del rendering
int h, m, s, app;
app = (int) (c_end - c_start) / CLOCKS_PER_SEC;
```

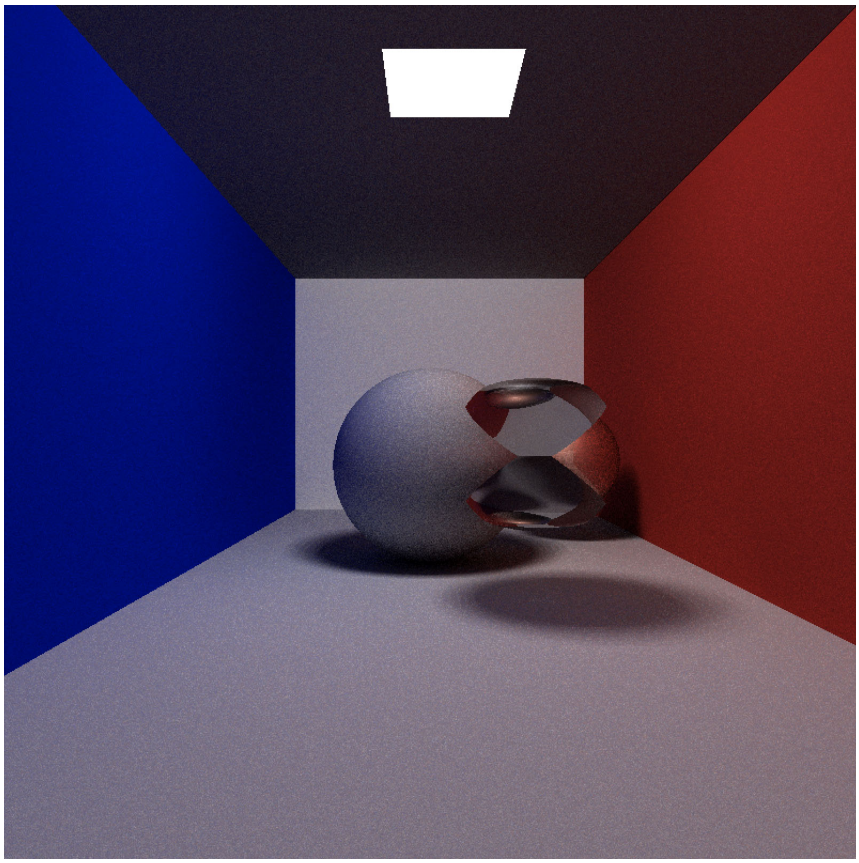


FIGURA 19.0.7. Sfera di gesso, sfera di rame e clessidra di vetro disposta lateralmente, rese con ray tracer stocastico

```
h = app/(3600);
app = app \% 3600;
m = app / 60;
s = app \% 60;
cout<<"Durata: "<<h<<" h, "<<m<<" min, "<<s<<" sec"<<endl;

//Scrittura del file img.ppm
ofstream ofs("./img.ppm", std::ios::out | std::ios::binary);
ofs<<"P6\n"<<width<<" "<<height<<"\n255\n";
for(int x=0; x<width; ++x)
{
     for(int y=0; y<height; ++y)
     {
         ofs<<(unsigned char)(std::min((float)1,(float) image[x][y].x)*255)<<
         (unsigned char)(std::min((float)1,(float) image[x][y].y)*255)<<
         (unsigned char)(std::min((float)1,(float) image[x][y].z)*255);
     }
}
//Apertura dell'immagine creata
```

```
    system("open img.ppm");
    ofs.close();
    return 0;
}
```

Shape.hpp:

```
#ifndef Shape_hpp
#define Shape_hpp

#include <stdio.h>
#include "glm/glm.hpp"
#include "Material.hpp"
#include "Light.hpp"
#include "Ray.hpp"

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//Classe per una generica forma
class Shape {
public:
    //Posizione del centro
    vec3 position;

    //Normale
    vec3 normal;

    //Indica se si sta calcolando illuminazione diretta o no
    bool dir;

    //Oggetto di tipo materiale
    Material mat;

    //Costruttori
    Shape(void);
    Shape(vec3 _position, vec3 _color , int _ks, vec3 _Cs);

    //Metodi virtual per assicurare che vengano definiti,
    //se si crea una classe figlio a partire da shape

    //Metodo per calcolare l'intersezione ->
    //restituisce vero se interseca superficie e t e' il parametro che indica la
        distanza
    virtual bool Intersection(float* t, vec3 rayOrigin, vec3 rayDirection);

    //Metodo che traccia il raggio e verifica se c'e' un'intersezione
    virtual bool Trace(vec3 rayOrigin, vec3 rayDirection);
```

```
    //Metodo per calcolare la normale
    virtual vec3 CalculateNormal(vec3 p);

    //Metodo per calcolo Brdf
    virtual vec3 BRDF(vec3, vec3, vec3);
};
#endif /* Shape_hpp */
```

Shape.cpp:

```
#include "Shape.hpp"
#include "glm/glm.hpp"
#include "Material.hpp"
#include "Light.hpp"
#include "Ray.hpp"


//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//Classe di una generica forma
//Serve solo da base per creare altre forme

//Costruttore Vuoto
Shape::Shape(void){
    position = vec3(0,0,0);
    normal = vec3(0,0,0);
    mat = Material(vec3(0,0,0), vec3(0.7,0.7,0.7),1000000);
}

//Costruttore con passaggio di parametri
Shape::Shape(vec3 _position, vec3 _color, int _ks, vec3 _Cs ){
    position = _position;
    mat = Material(_color, _Cs,_ks);

}

//Metodo per calcolare l'intersezione ->
//Restituisce vero se interseca superficie e t e' il parametro che indica la
    distanza
bool Shape::Intersection(float *t, vec3 rayOrigin, vec3 rayDirection){
    return false;
}

//Metodo che traccia il raggio e verifica se c'e' un'intersezione
bool Shape::Trace(vec3 rayOrigin, vec3 rayDirection){
    return false;
```

```
}

//Metodo per calcolare la normale
vec3 Shape::CalculateNormal(vec3 p)
{
    return normal;
}

//Metodo per calcolo brdf
vec3 Shape::BRDF(vec3 p, vec3 light, vec3 view)
{
    return mat.Cd;
}
```

Sphere.h:

```
#ifndef Sphere_h
#define Sphere_h

 #include <iostream>
 #include "glm/glm.hpp"
 #include "Shape.hpp"
 #include "Light.hpp"
 #include "Ray.hpp"


 using namespace std;

//Libreria glm per l'utilizzo della classe vec3
 using namespace glm;

//Classe Sfera
 class Sphere: public Shape{
    public:

    //Raggio
    float radius;

    //Costruttori
    Sphere(void);
    Sphere(vec3, vec3 , float , int);
    Sphere(vec3, vec3 , float , int, float);

    //Metodo per calcolare l'intersezione ->
    //restituisce vero se interseca superficie e t e' il parametro che indica la
        distanza
    bool Intersection(float *, vec3 , vec3 );
```

```
    //Metodo che traccia il raggio e verifica se c'e' un'intersezione
    bool Trace(vec3 rayOrigin, vec3 rayDirection);

    //Metodo che calcola la normale
    vec3 CalculateNormal(vec3 p);

    //Metodo per il calcolo della Brdf
    vec3 BRDF(vec3, vec3, vec3);
};
#endif /* Sphere_h */
```

Sphere.cpp:

```
#include "Sphere.h"
#include <iostream>
#include <algorithm>
#include "glm/glm.hpp"
#include "Light.hpp"
#include "Ray.hpp"

using namespace std;

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;


//Flag per attivare la mappa di rilievo con la formula di blinn
bool blinn = false;


//Classe Sfera

//Costruttori
Sphere::Sphere(void ){
    position = vec3(0,0,0);
    radius = 0;
    mat = Material(vec3(0,0,0), vec3(0.7,0.7,0.7), 0);
}

Sphere::Sphere(vec3 _position, vec3 _color, float _radius, int _type){
    position = _position;
    radius = _radius;
    dir = true;
    mat = Material(_color, vec3(0.7,0.7,0.7), _type);
}

Sphere::Sphere(vec3 _position, vec3 _color, float _radius, int _type, float _ks){
    position = _position;
```

```
    radius = _radius;
    dir = true;
    mat = Material(_color, _type, _ks);
}


//Parametrizzazione della superficie, restitusce l'angolo di langitudine e
    latitudine
void paramSup(float &a, float &b, vec3 normal)
{
    a = acosf(dot(vec3(0,1,0),normal));
    b  = acosf(normal.x/sinf(a));
    if(std::isnan(b))
        b = 0;
}


//Calcolo Intersezione
/*
    Si traccia il raggio dist che congiunge l'origine e il centro della clessidra.
    Successivamente si calcola la proiezione scalar di tale raggio sul vettore
    che indica la direzione in cui sta guardando l'osservatore.
    Se questa proiezione e' negativa significa che non c'e' intersezione
    nella direzione in cui guarda l'osservatore ma in quella opposta.
    Se e' positiva si verifica che la lunghezza del segmento che congiunge
    i due vettori dist e scalar sia minore del raggio del sfera.
    Se lo e' c'e' intersezione

    Sarebbe stato piu' efficiente considerare la superficie come una mesh
    di maglie triangolari, e calcolare
    l'intersezione per i triangoli che la compongono.
    Questo permetterebbe di traslare e ruotare simultaneamente punti della mesh,
    vettori normali e tessiture.
*/

bool Sphere::Intersection(float *t, vec3 rayOrigin, vec3 rayDir){

    //Vettore distanza tra il centro della clessidra e la posizione dell'osservatore
    vec3 dist = rayOrigin - position;

    //Proiezione del vettore dist sul vettore che rappresenta la direzione
        dell'osservatore
    float scalar = dot(dist, rayDir);

    //Se il prodotto scalare e' < 0 allora non c'e' intersezione
    if(scalar<0)
        return false;
    else
    {
```

```
        //calcolo la distanza f2 tra il vettore dist e scalar
        float f = dot(dist,dist)-(scalar*scalar);
        float f2 = sqrt(f);

        //verifico che f2 sia minore del raggio
        if(f2>radius)
            return false;
        else
        {
            float a = sqrt((radius*radius)-f);
            *t = scalar - a;
            return true;
        }

    }
}

//Traccia e verifica se raggio incontra superficie
//Stesso calcolo dell'intersezione ma non restituisce la distanza
bool Sphere::Trace(vec3 rayOrigin, vec3 rayDir){

    vec3 dist = rayOrigin - position;
    float scalar = dot(dist, rayDir);
    if(scalar>0)
        return false;
    else
    {
        float f = dot(dist,dist)-(scalar*scalar);
        float f2 = sqrt(f);
        if(f2>radius)
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}


//Calcolo Normale
/*
    La normale in un punto p di una sfera e'

                P - POS[centro]
              ---------------------
               ||P - POS[centro]||
```

```
    Se e' attivo il flag "Blinn",
    viene calcolata una nuova normale dalla formula di Blinn per la mappa di rilievo
*/
vec3 Sphere::CalculateNormal(vec3 p)
{
    vec3 normal;
    vec3 ris = p - position;

    if(mat.type==2 && blinn)
    {
            /*
                    Formula Di Blinn Per Mappa Di Rilievo

                    N' =  N  + D1b *(N x D2p) - D2b * (N x D1p)

                    Dove: N' = nuova normale N = normale effettiva
                    D1 = derivata rispetto prima variabile
                    b = mappa di rilievo scelta p = punto della sfera
                    D2 = derivata rispetto la seconda variabile


                    In questo caso b = cos(Theta) * sin(Phi) * Raggio / 100

                    quindi D1b = - sin(Theta) * cos(Phi) * Raggio / 100
                    D2b = - cos(Theta) * sin(Phi) * Raggio / 100

                    La parametrizzazione della superficie e'
                    x = sin(Theta) * cos(Phi)
                    y = cos(Theta)                    --->
                    z = sin(Theta) * sin(Phi)


                    D1p  = (cos(Theta) * cos(Phi), sin(Phi), cos(Theta) * sin(Phi))
                    D2p  = (sin(Theta) * sin(Phi), 0, sin(Theta) * cos(Phi))

                dove THETA e' l'angolo latitudine dal polo Nord e PHI l'angolo di
                    longitudine
            */

            //Calcolo la normale della sfera
            normal = normalize(ris);

            //Theta e Phi per parametrizzazione sfera
            float THETA = 0;
            float PHI = 0;

            paramSup(THETA, PHI, normal);
```

```
            //Theta e Phi casuali
            float THETA1 = drand48()*90;
            float PHI1 = drand48()*90;

            //Numero casuale tra 0 e 1
            float z = drand48();

            //I due termini della formula a cui poi andremo a sommare la normale
            vec3 term1 = -sinf(THETA1) * cosf(PHI1) * 2 * z * radius/10 *
                cross(normal, radius * vec3(-sinf(THETA)*sinf(PHI), 0,
                sinf(THETA)*cosf(PHI)));
            vec3 term2 = -cosf(THETA1) * sinf(PHI1) * 2 * z * radius/10 *
                cross(normal, radius * vec3(cosf(THETA)*cosf(PHI), -sinf(THETA),
                cosf(THETA)*sinf(PHI)));

            //Somma alla normale la differenza dei due termini
            ris = normal + term1 - term2;
    }



    normal = normalize(ris);
    return normal;
}



//Calcolo Brdf
/*
   Se il parametro type del materiale e' 1 viene gestito con la Brdf di Lambert
   permette di simulare materiale totalmente diffusivi


   Se il parametro type e' 2 viene gestito con la Brdf di Cook-Torrance
   questa Brdf e' basata sulla teoria delle micro-faccette.
   Si immagina quindi che la superficie sia composta da
   tante piccole superfici inclinate a seconda del parametro 'alpha',
   che indica quindi la rugosita' del materiale.
   Si considera inoltre la distribuzione delle microfaccette
   e il fattore di attenuazione geometrica,
   che prevede i casi in cui la microfaccetta sia coperta totalmente
   o parzialmente da un'altra.
   Il termine di Fresnel determina la quantita' di luce riflessa.

*/
vec3 Sphere::BRDF(vec3 p, vec3 lightRay, vec3 view)
{
    vec3 ris = vec3(0);
    normal = normalize(CalculateNormal(p));
```

```
//Calcolo Brdf di Cook-Torrance
if(mat.type == 2)
{
    /*
     Formule per Brdf Cook-Torrance

                                D * F * G
            BRDF_CT =        -----------------
                             4 * <N,V> * <N,L>

    D = distribzione delle microfaccette
    F = termine di Fresnel
    G = fattore di attenuazione geometrica

                    2 * <N,H> * <N,V>        2 * <N,H> * <N,L>
      G  =  min ( 1, ------------------   ,   ------------------ )
                          <V,H>                    <V,H>


                            alpha^2
      D(H)  =    ----------------------------------------
     (GGX)       pigreco * [ <N,H> ^2 * (alpha^2 - 1) + 1 ]^2


    alpha = rugosita' dell'oggetto, inclinazione delle microfaccette
    piu' e' alto piu' e rugoso l'oggetto (diffusivo)
    piu' e' basso piu' e liscio l'oggetto (riflettente)


             F = F0 + (1 - F0)(1 - <V,H> )
           (Schlick)

                              eta1 - eta2   2
         dove       F0  =  ( ------------ )
                              eta1 + eta2

       eta1 e' l'indice di rifrazione del materiale
       eta2 = 1 PER ARIA

         Oppure prendere F0 in base alla seguente tabella
    Water     0.02, 0.02, 0.02
    Plastic   0.03, 0.03, 0.03
    Glass     0.08, 0.08, 0.08
    Diamond   0.17, 0.17, 0.17
    Gold      1.00, 0.71, 0.29
    Silver    0.95, 0.93, 0.88
    Copper    0.95, 0.64, 0.54
    Iron      0.56, 0.57, 0.58
```

```
     Aluminum  0.91, 0.92, 0.92

     Fonte:
         (www.cs.cornell.edu/courses/cs5625/2013sp/lectures/Lec2ShadingModelsWeb.pdf)
    */

    //Calcolo halfway vector -> vettore tra v e l
    vec3 H = normalize((view + lightRay));
    float D, G;

    //Rugosita' del materiale
    float roughness = 0.4;
    float app, alfa2;

    //Calcolo vari prodotti scalari
    float dot_N_H = fabs(dot(normal,H));                // <N,H>
    float dot_N_L = fabs(dot(normal,lightRay));         // <N,L>
    float dot_V_H = fabs(dot(H, view));                 // <V,H>
    float dot_N_V = fabs(dot(normal,view));             // <N,V>


    //Calcolo FATTORE DI ATTENUAZIONE GEOMETRICA
    app = fmin((2 * dot_N_H * dot_N_L)/(dot_V_H), (2 * dot_N_H *
        dot_N_V)/(dot_V_H));
    G = fmin(1, app);


    //Calcolo distribuzione delle microfaccette -> uso distribuzione GGX
    alfa2 = roughness * roughness;
    D = alfa2 / (M_PI * pow(pow(dot_N_H,2) * (alfa2-1) + 1, 2));

    //Calcolo Fresnel -> utilizzo l'approssimazione si Schlick
    vec3 F0 = vec3(0.95, 0.64, 0.54);
    vec3 F = F0 + (vec3(1) - F0) * (float)pow(1 - dot_V_H,5);

    //Risultato
    float denom = (4 * dot_N_V * dot_N_L );
    vec3 cook = F * (float) (D * G / denom) ;

    ris = clamp(cook, vec3(0,0,0), vec3(1,1,1));

    return ris;
}

//Brdf Diffusiva
if(mat.type==1)
{
    /*
```

```
         Formula      Cd
         Brdf_L =  -------          (dove  Cd = Colore dell'oggetto)
                   pigreco
        */

        ris = mat.Cd / (float)M_PI ;
        ris = clamp(ris, vec3(0,0,0), vec3(1,1,1));
    }
    return ris;
}
```

Hourglass.hpp:

```
#ifndef Hourglass_hpp
#define Hourglass_hpp

#include <iostream>
#include "Shape.hpp"
#include "glm/glm.hpp"

using namespace std;

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

class Hourglass: public Shape{
    public:

    //Raggio massimo della clessidra
    float radius;

    //Costruttori
    Hourglass(void);
    Hourglass(vec3, vec3 , float , int);
    Hourglass(vec3, vec3 , float , int, float);

    //Metodo per calcolare l'intersezione ->
    //restituisce vero se interseca superficie e t e' il parametro che indica la
        distanza
    bool Intersection(float *, vec3 , vec3 );

    //Metodo che traccia il raggio e verifica se c'e' un'intersezione
    bool Trace(vec3 rayOrigin, vec3 rayDirection);

    //Metodo che calcola la normale
    vec3 CalculateNormal(vec3 p);

    //Metodo per il calcolo della Brdf
```

```
    vec3 BRDF(vec3, vec3, vec3);

};

#endif /* Hourglass_hpp */
```

Hourglass.cpp:

```
#include <stdio.h>
#include "Hourglass.hpp"
#include <algorithm>
#include "glm/glm.hpp"


using namespace std;

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

/*
   Classe Hourglass
   Classe che crea una clessidra modificando l'equazione della sfera
   considerando l'equazione di un generico punto della sfera

        x = r * sin(Theta) * cos(Phi)
        y = r * cos(Theta)
        z = r * sin(Theta) * sin(Phi)

    Dove Theta e' l'angolo di latitudine e phi quello di longitudine.

    moltiplicando alle cordinate x e z un fattore pari a

        cos(Theta) * (9*r/10) + r/10

   Questo fattore non diventa mai 0, ossia la clessidra non si strozza al centro;
   infatti nella parte centrale e' un disco di raggio r/10.

   il fattore cos(theta) puo' essere anche negativo poiche' Theta e' in [0,pigreco]
   quindi si considera |cos(Theta)|

*/

//Costruttori
Hourglass::Hourglass(void)
{
    position = vec3(0,0,0);
    radius = 0;
    mat = Material(vec3(0,0,0), vec3(0.7,0.7,0.7), 0);
```

```
}

Hourglass::Hourglass(vec3 _position, vec3 _color, float _radius, int _type){
    position = _position;
    radius = _radius;
    dir = true;
    mat = Material(_color, vec3(0.7,0.7,0.7), _type);
}

Hourglass::Hourglass(vec3 _position, vec3 _color, float _radius, int _type, float
    _ks){
    position = _position;
    radius = _radius;
    dir = true;
    mat = Material(_color, _type, _ks);
}

//Parametrizzazione della superficie, restitusce l'angolo di longitudine e
    latitudine
void param(float &c, float &d, vec3 normal, float radius, vec3 p, vec3 position)
{

    c = acosf(dot(vec3(0, 1, 0),normal));
    d = acosf( ((p.x - position.x) / (sinf(c) * radius)) );
    if(std::isnan(d))
        d = 0;

}

//Calcolo dell'intersezione
/*
    Si traccia il raggio dist che congiunge l'origine e il centro della clessidra.
    Successivamente si calcola la proiezione scalar di tale raggio sul vettore
    che indica la direzione in cui sta guardando l'osservatore.
    Se questa proiezione e' negativa significa che non c'e' intersezione
    nella direzione in cui guarda l'osservatore ma in quella opposta.
    Se e' positiva si verifica che la lunghezza del segmento che congiunge
    i due vettori dist e scalar
    sia minore del raggio del disco di sezione assiale della clessidra ad altezza
        raggio*Cos(Theta),
    dove Theta e' la deviazione assiale del punto osservato sulla clessidra rispeto
        al suo polo Nord.
    Se lo e' c'e' intersezione

    Sarebbe stato piu' efficiente considerare la superficie come una mesh
    di maglie triangolari, e calcolare
    l'intersezione per i triangoli che la compongono.
    Questo permetterebbe di traslare e ruotare simultaneamente punti della mesh,
    vettori normali e tessiture.
```

```
*/
bool Hourglass::Intersection(float *t, vec3 rayOrigin, vec3 rayDir){

    //Centro della sfera
    vec3 app = position;

    //Vettore distanza tra il centro della clessidra e la posizione dell'osservatore
    vec3 dist = rayOrigin - app;

    //Proiezione del vettore dist sul vettore che rappresenta la direzione
       dell'osservatore
    float scalar = dot(dist, rayDir);

    //Se il prodotto scalare e' < 0 allora non c'e' intersezione
    if(scalar<0)
        return false;
    else
    {
        //calcolo la distanza tra il vettore dist e scalar
        float f = dot(dist,dist)-(scalar*scalar);
        float f2 = sqrt(f);

        //se la distanza e' maggiore del raggio non c'e' intersezione
        //si escludono quindi le intersezioni con la sfera di raggio "radius".
        //Sarebbe stato piu' efficiente utilizzare come bounding box un cilindro
        //escludendo quindi le intersezione se f2 maggiore della distanza dall'asse
        if(f2>radius)
            return false;
        else
        {

                //altrienti si verifica che f2 sia minore del raggio del disco
                //di sezione assiale della clessidra ad altezza raggio*Cos(Theta)
                float a = sqrt((radius*radius)-f);
                *t = scalar - a;
                vec3  p = rayOrigin + (-*t * rayDir);
                vec3  n = normalize(p - position);
                float THETA = acosf(dot(vec3(0,1,0),n));
                float x = radius * sinf(THETA);

                float p1 = x * (abs(cosf(THETA)) * radius * 9/10 + radius * 1/10);


                if(f2>p1)
                {
                    return false;
                }

                a = sqrt((p1*p1)-f);
```

```
                *t = scalar - a;
                return true;


        }

    }

}



//Traccia e verifica se raggio incontra superficie
//Stesso calcolo dell'intersezione ma non restituisce la distanza
bool Hourglass::Trace(vec3 rayOrigin, vec3 rayDir)
{
    float t;
    vec3 dist = rayOrigin - position;
    float scalar = dot(dist, rayDir);
    if(scalar>0)
        return false;
    else
    {
        float f = dot(dist,dist)-(scalar*scalar);
        float f2 = sqrt(f);

        if(f2>radius)
            return false;
        else
        {

            float a = sqrt((radius*radius)-f);
            t = scalar - a;
            vec3  p = rayOrigin + (t * rayDir);
            vec3  n = normalize(p-position);
            float THETA = acosf(dot(vec3(0,1,0),n));

            float x = radius * sinf(THETA);

            float p1 = x * (abs(cosf(THETA))*radius * 9/10 + radius * 1/10);

            a = sqrt((p1*p1)-f);

            if(a>p1)
            {
                return false;
            }
            return true;
```

```
        }

    }
}



//Calcolo della normale nel punto p
vec3 Hourglass::CalculateNormal(vec3 p)
{
    vec3 normal;

    //Calcolo la normale della sfera
    vec3 ris = p - position;
    normal = normalize(ris);

    //Teta E Phi per parametrizzazione sfera
    float THETA = 0;
    float PHI = 0;
    param(THETA, PHI, normal, radius, p, position);

    /*
        Per calcolare la normale di una superficie uso la formula:

            N =  d2(P) x d1(P)

        Dove P e' un generico punto della superficie
        d1 e' derivata rispetto alla prima variabile e d2 rispetto la seconda

        Nel caso di una sfera il punto P e' indicato con

            x = r * sin(Theta) * cos(Phi)
            y = r * cos(Theta)
            z = r * sin(Theta) * sin(Phi)

        In questo caso si moltiplicano le coordinate x e z per il fattore
        cos(Theta) * (9*r/10) + r/10
        In modo da ottenere due sfere svasate. il generico punto e' quindi

            x = r * sin(Theta) * cos(Phi) * (cos(Theta) * 9 * r / 10 + r/10)
            y = r * cos(Theta)
            z = r * sin(Theta) * sin(Phi) * (cos(Theta) * 9 * r / 10 + r/10)

        Derivando rispetto Phi otteniamo

         d2(P) =
          (- r * sin(Theta) * sin(Phi) * (cos(Theta) * 9 * r/10 + r/10),
      0,
            r * sin(Theta) * cos(Phi) * (cos(Theta) * 9 * r/10 + r/10) )
```

```
        e rispetto a Theta

          d1(P) =
          (r/10 * cos(Phi) * ( cos(Teta) * ( 9 * r * cos(Theta) + r ) - 9 *
              sin(Theta)^2 ),
           - r * sin(Theta),
            r/10 * sin(Phi) * ( cos(Teta) * ( 9 * r * cos(Theta) + r ) - 9 *
                sin(Theta)^2 )

        Poiche' si considera il valore assoluto di cos(Theta) si divide in due casi
        */

    vec3 delta1, delta2;


    if(cosf(THETA)>=-0.1)
        delta1 = normalize(vec3(0.1f *radius* cosf(PHI) *
            (cosf(THETA)*(9*radius*cosf(THETA)+radius) - 9 *
            sinf(THETA)*sinf(THETA)),
                                -radius * sinf(THETA),
                            0.1f *radius* sinf(PHI) *
                                (cosf(THETA)*(9*radius*cosf(THETA)+radius) - 9 *
                                sinf(THETA)*sinf(THETA))));
    else
        delta1 = normalize(vec3(0.1f *radius* cosf(PHI) *
            (-(cosf(THETA))*(9*radius*cosf(THETA)+radius) + 9 *
            sinf(THETA)*sinf(THETA)),
                                -radius * sinf(THETA),
                            0.1f *radius* sinf(PHI) *
                                (-(cosf(THETA))*(9*radius*cosf(THETA)+radius) + 9
                                * sinf(THETA)*sinf(THETA))));

    delta2 = normalize(vec3( - radius * sinf(THETA) * sinf(PHI) * (abs(cosf(THETA))
        * radius*(0.9f) + (radius * 0.1f)),
                                        0,
                                radius * sinf(THETA) * cosf(PHI) * (abs(cosf(THETA)) *
                                    radius*(0.9f) + (radius * 0.1f))));


    ris = cross(delta2, delta1);

    normal = normalize(ris);

    return normal;
}


//Calcolo Brdf
```

```
/*
   Se il parametro type del materiale e' 1 viene gestito con la Brdf di Lambert
   permette di simulare materiale totalmente diffusivi.


    Se il parametro type e' 2 viene gestito con la Brdf di Cook-Torrance
    Questa Brdf e' basata sulla teoria delle micro-faccette.
    Si immagina quindi che la superficie sia composta da tante piccole superfici
    inclinate a seconda del parametro 'alpha' che indica quindi la rugosita' del
        materiale.
    Si considera inoltre la distribuzione delle microfaccette
    e il fattore di attenuazione geometrica.
    Che prevede i casi in cui la microfaccetta sia coperta totalmente o
        parzialmente da un'altra.
    Il termine di Fresnel determina la quantita' di luce riflessa

*/
vec3 Hourglass::BRDF(vec3 p, vec3 lightRay, vec3 view)
{
    vec3 ris = vec3(0);
    normal = normalize(CalculateNormal(p));

    //Calcolo Brdf con Cook-Torrance
    if(mat.type == 2)
    {
        /*
          Formule per Brdf Cook-Torrance


         Brdf_Ct = D * F * G /( 4 *(N,V)*(N,L) )

         D = distribzione delle microfaccette
         F = termine di Fresnel
         G = fattore di attenuazione geometrica


         G  =  min ( 1, 2 *(N,H)*(N,V)/ (V,H), 2 *(N,H)*(N,L )/ (V,H) )

         D(H)  =   alpha^2/ pigreco * ( (N,H)^2 * (alfa^2 - 1) + 1 )^2
        (GGX)


         alpha = rugosita' dell'oggetto, inclinazione delle microfaccette
         piu' e' alto piu' e rugoso l'oggetto (diffusivo)
         piu' e' basso piu' e liscio l'oggetto (riflettente)

            F = F0 + (1 - F0)*(1 -(V,H) )
         (Schlick)
```

```
 dove
 F0  =  (eta1 - eta2)/(eta1+eta2)^2,
 eta1 e' l'indice di rifrazione del materiale
 eta2 = 1 per aria

          Oppure prendere F0 in base alla seguente tabella
 Water     0.02, 0.02, 0.02
 Plastic    0.03, 0.03, 0.03
 Glass     0.08, 0.08, 0.08
 Diamond   0.17, 0.17, 0.17
 Gold      1.00, 0.71, 0.29
 Silver    0.95, 0.93, 0.88
 Copper 0.95, 0.64, 0.54
 Iron       0.56, 0.57, 0.58
 Aluminum  0.91, 0.92, 0.92

 Fonte: (www.cs.cornell.edu/courses/cs5625/2013sp
    /lectures/Lec2ShadingModelsWeb.pdf)
*/

//Calcolo halfway vector -> vettore tra V e L
vec3 H = normalize((view + lightRay));
float D, G;

//Rugosita' del materiale
float roughness = 0.2;
float app, alfa2;

//Calcolo vari prodotti scalari
float dot_N_H = fabs(dot(normal,H));              // (N,H)
float dot_N_L = fabs(dot(normal,lightRay));    // (N,L)
float dot_V_H = fabs(dot(H, view));               // (V,H)
float dot_N_V = fabs(dot(normal,view));        // (N,V)


//Calcolo fattore di attenuazione geometrica
app = fmin((2 * dot_N_H * dot_N_L)/(dot_V_H), (2 * dot_N_H *
   dot_N_V)/(dot_V_H));
G = fmin(1, app);


//Calcolo distribuzione delle microfaccette -> uso distribuzione GGX
alfa2 = roughness * roughness;
D = alfa2 / (M_PI * pow(pow(dot_N_H,2) * (alfa2-1) + 1, 2));

//Calcolo Fresnel -> utilizzo l'approssimazione di Schlick
vec3 F0 = vec3( 0.08, 0.08, 0.08);
vec3 F = F0 + (vec3(1) - F0) * (float)pow(1 - dot_V_H,5);
```

```
        //Risultato
        float denom = (4 * dot_N_V * dot_N_L );
        vec3 cook = F * (float) (D * G / denom) ;

        ris = clamp(cook, vec3(0,0,0), vec3(1,1,1));

        return ris;
    }

    //Brdf Diffusiva
    if(mat.type==1)
    {
        /*
           Brdf_L = Cd / pigreco         (Cd = Colore dell'oggetto)
        */

        ris = mat.Cd / (float)M_PI ;
        ris = clamp(ris, vec3(0,0,0), vec3(1,1,1));
    }
    return ris;
}
```

Plane.hpp:

```
#ifndef Plane_hpp
#define Plane_hpp

#include <stdio.h>
#include "glm/glm.hpp"
#include "Shape.hpp"
#include "Light.hpp"
#include "Ray.hpp"


//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//Classe per definire un piano a partire da una forma
class Plane : public Shape {
public:

    //Costruttore
    Plane(void);
    Plane(vec3 _position, vec3 _color, vec3 _Normal);

    //Metodo per calcolare l'intersezione ->
    //restituisce vero se interseca superficie e t e' il parametro che indica la
        distanza
```

```
    bool Intersection(float* t, vec3 rayOrigin, vec3 rayDirection);

    //Metodo che traccia il raggio e verifica se c'e' un'intersezione
    bool Trace(vec3 rayOrigin, vec3 rayDirection);

    //Metodo per calcolare la normale
    vec3 CalculateNormal(vec3 p);

    //Metodo per calcolo Brdf
    vec3 BRDF(vec3 p, vec3, vec3);
};

#endif /* Plane_hpp */
```

Plane.cpp:

```
#include "Plane.hpp"
#include "glm/glm.hpp"
#include "Shape.hpp"
#include "Light.hpp"
#include "Ray.hpp"

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

// Classe per definire un piano
// I parametri utilizzati sono : normale e posizione

//Costruttore Vuoto
Plane::Plane(void)
{
    normal = vec3(0,0,0);
    mat = Material(vec3(0,0,0), vec3(0.8,0.8,0.8), 1000000);
}

//Costruttore con passaggio parametri
Plane::Plane(vec3 _position, vec3 _color, vec3 _Normal)
{
    position=_position;
    normal = _Normal;
    dir = true;
    mat = Material(_color, vec3(0.8,0.8,0.8), 1000000);
}

//Calcola intersezione e restituisce vero se interseca una superficie,
//il parametro t indica la distanza
//Infatti t e' proporzionale alla lunghezza del vettore rayOrigin - position,
ossia alla lunghezza del raggio tracciato (a partire dal suo inizio).
```

```
Poiche' questo e' un vettore, dovremmo calcolarne la norma; ma e' piu' semplice
    proiettare perpendicolarmente alla normale alla superficie, dividendo poi per
    il versore di direzione del raggio anch'esso proiettato. Se il raggio giace
    sulla superficie piana allora questi prodotti scalari fanno zero, e la routine
    di intersezione restituisce false a causa del controllo di soglia.

bool Plane::Intersection(float *t, vec3 rayOrigin, vec3 rayDirection)
{
    float soglia= 1e-6;
    float scalar = dot(rayDirection, normal);
    if(scalar<soglia)
        return false;
    else
    {
        float ris = dot((rayOrigin - position),normal)/scalar;
        if(ris>=0)
        {
            *t=ris;
            return true;
        }
        else
            return false;
    }

}

//Traccia e verifica se raggio incontra superficie
bool Plane::Trace(vec3 rayOrigin, vec3 rayDirection)
{
    float soglia= 1e-6;
    float scalar = dot(rayDirection, normal);
    if(scalar>soglia)
        return false;
    else
    {
        float ris = dot((rayOrigin - position),normal)/scalar;
        if(ris>=0)
            return true;
        else
            return false;
    }

}

//Calcola la normale, in questo caso la restitusce semplicemente
//perche' definita nel costruttore e stessa per ogni punto
vec3 Plane::CalculateNormal(vec3 p)
{
    return normal;
```

```
}


//Calcolo Brdf
//per i piani gestione con brdf diffusiva utilizzando il modello di Lambert
vec3 Plane::BRDF(vec3 p, vec3 light, vec3 view)
{
    vec3 diffuse;

    /*
        Brdf_L = Cd / pigreco         (Cd = Colore dell'oggetto)
    */

    diffuse = mat.Cd / (float)M_PI ;
    diffuse = clamp(diffuse, vec3(0,0,0), vec3(1,1,1));

    return diffuse;
}
```

Material.hpp:

```
#ifndef Material_hpp
#define Material_hpp

#include <stdio.h>
#include "glm/glm.hpp"

using namespace std;

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

class Material {
public:

    //Colore Diffuso
    vec3 Cd;

    //Colore Speculare
    vec3 Cs;

    //Coefficiente Di Specularita'
    float ks;

    //Indica come gestire la brdf
    int type;

    //Costruttori
```

```
    Material(void);
    Material(vec3, vec3, int);
    Material(vec3, int);
    Material(vec3, int, float);
    Material(vec3);
};

#endif /* Material_hpp */
```

Material.cpp:

```
#include "Material.hpp"

#include "glm/glm.hpp"
#include <stdio.h>

using namespace std;

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;


//Classe Materiale
//Cd   -> colore diffuso
//Cs   -> colore riflesso
//Ks   -> fattore di riflessione
//Tipo -> variabile che indica quale brdf usare

//Costruttore vuoto
Material::Material(void)
{
    Cd = vec3(0.5,0.5,0.5);
    Cs = vec3(0.7,0.7,0.7);
    type = 0;
    ks = 0.3f;
}



//Costruttori con passaggio di parametri
Material::Material(vec3 _Cd, vec3 _Cs, int _type)
{
    Cd = _Cd;
    Cs = _Cs;
    type = _type;
}

Material::Material(vec3 _Cd, int _type)
```

```
{
    Cd = _Cd;
    Cs = vec3(0.7, 0.7, 0.7);
    type = _type;
}

Material::Material(vec3 _Cd, int _type, float _ks)
{
    Cd = _Cd;
    Cs = vec3(0.7, 0.7, 0.7);
    type = _type;
    ks = _ks;
}

Material::Material(vec3 _Cd)
{
    Cd = _Cd;
    Cs = vec3(0.7, 0.7, 0.7);
    type = 0;
}
```

Light.hpp:

```
#ifndef Light_hpp
#define Light_hpp

#include <stdio.h>
#include "glm/glm.hpp"

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

class Light{
    public:
    //Intensita' di luce
    vec3 LightInt;

    //Posizione della luce
    vec3 LightPos;

    //Dimensione della luce
    vec3 LightSize;


    //Costruttori
    Light(void);
    Light(vec3, vec3, vec3);
};
```

```
#endif /* Light_hpp */
```

Light.cpp:

```
#include "Light.hpp"
#include "glm/glm.hpp"

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//Classe per luci

//Costruttore vuoto per la classe luce, imposta dei valori predefiniti
Light::Light(void)
{
    LightInt = vec3(0.7,0.7,0.7);
    LightPos = vec3(0,0,0);
    LightSize = vec3(1,0.1f,1);
}

//Costruttore per la classe luce con passaggio di parametri
Light::Light(vec3 _LightInt, vec3 _LightPos, vec3 _LightSize)
{
    LightInt = _LightInt;
    LightPos = _LightPos;
    LightSize = _LightSize;
}
```

AreaLight.hpp:

```
#ifndef AreaLight_hpp
#define AreaLight_hpp

#include <stdio.h>
#include "glm/glm.hpp"


//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

/*
    Classe utilizzata per definire una sorgente di luce diffusa
*/
class AreaLight{
    public:
```

```
    //Posizione
    vec3 position;

    //Dimensione
    vec3 size;

    //Costruttori
    AreaLight(void);
    AreaLight(vec3 _position, vec3 _size);
};


#endif /* BoundingBox_hpp */
```

AreaLight.cpp:

```
#include "AreaLight.hpp"
#include "glm/glm.hpp"


//Libreria glm per l'utilizzo della classe vec3
using namespace glm;


//Classe per definire un'area light

//Costruttori
AreaLight::AreaLight(void){
    position = vec3(0,0,0);
    size = vec3(0,0,0);
}

AreaLight::AreaLight(vec3 _position, vec3 _size){
    position = _position;
    size = _size;
}
```

Ray.hpp:

```
#ifndef Ray_hpp
#define Ray_hpp

#include <stdio.h>
#include "glm/glm.hpp"
```

```
//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

class Ray{
public:

    //Orgine
    vec3 origin;

    //Direzione
    vec3 direction;

    //Costruttori
    Ray(void);
    Ray(vec3 _origin, vec3 _diredirection);
};

#endif /* Ray_hpp */
```

Ray.cpp:

```
#include "Ray.hpp"
#include "glm/glm.hpp"

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//Classe per raggio

//Costruttore vuoto
Ray::Ray(void)
{
    origin = vec3(0,0,0);
    direction = vec3(0,0,-1);
}

//Costruttore con passaggio di parametri
Ray::Ray(vec3 _origin, vec3 _diredirection)
{
    origin = _origin;
    direction = _diredirection;
}
```

Renderer.hpp:

```
#ifndef Render_hpp
```

```
#define Render_hpp

#include <stdio.h>
#include "Shape.hpp"
#include "Sphere.h"
#include "Plane.hpp"
#include <fstream>
#include <algorithm>
#include "glm/glm.hpp"
#include "AreaLight.hpp"
#include "Ray.hpp"
#include <iostream>
#include <random>
#include "Light.hpp"

using namespace std;

//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//Header della classe render, per dettagli vedere il file render.cpp

class Render{
public:
    //Numero elementi della scena
    int NumEL;

    //Vettore degli elementi della scena
    Shape* shapeList[10];

    //Area per definire luce
    AreaLight* areaLight;

    //Luce
    Light* light;

    //Costruttore
    Render(void);

    //Calcola la radianza in un punto, calcolando il contributo dell'illuminazione
        diretta e indiretta
    vec3 computeRadiance(int shapeHit, vec3 p, Ray* view, int depth, int depth1);

    //Calcolo illuminazione diretta
    vec3 directIllumination(int shapeHit, vec3 p, Ray* view, int depth);

    //Crea un sistema di coordinate locali a partire dalla normale
    void createCoordinateSystem(vec3 N, vec3 &v1, vec3 &v2);
```

```
    //Campionamento uniforme dell'emisfero
    vec3 uniformSampleHemisphere(float &u1, float &u2, float &pdf);

    //Campionamento per coseno dell'emisfero
    vec3 CosineSampleHemisphere(float &u1, float &u2, float &pdf);

    //Calcolo illuminazione indiretta
    vec3 indirectIllumination(int shapeHit, vec3 p, Ray* view, int depth);

    //Genera l'immagine
    vec3** computeImage(vec3 RayOrigin, int, int);
};


#endif /* Render_hpp */
```

Renderer.cpp:

```
#include "Render.hpp"
#include <stdio.h>
#include "Shape.hpp"
#include "Sphere.h"
#include "Plane.hpp"
#include <fstream>
#include <algorithm>
#include "glm/glm.hpp"
#include "AreaLight.hpp"
#include "Ray.hpp"
#include <iostream>
#include <random>
#include "Light.hpp"
#include "Hourglass.hpp"


using namespace std;

/*
  In questa classe avviene il rendering dell'immagine.
  Istanziando un oggetto di Render e richiamando il metodo computeimage
  si crea una videocamera (ossia un osservatore) e un sistema di coordinate.
  Per ogni pixel dell'immagine si calcola l'intersezione con gli oggetti della
      scena.
  Si utilizza il metodo computeradiance per calcolare la radianza nel punto trovato
  computeradiance calcola separatamente l'illuminazione diretta e indiretta.

        (Commenti piu' dettagliati accompagnano le singole righe di codice)

*/
```

```
//Libreria glm per l'utilizzo della classe vec3
using namespace glm;

//Numero elementi scena
const int NumEl = 9;

//Numero raggi d'ombra per lato
float sampleShadows = 6.0f;
float sampleShadowsInd = 1.0f;

//Numero campioni per estimatore Montecarlo
float sapleMC = 25.0f;

//Soglia massima di rimbalzi per l'estimatore
int MaxDepth = 1;

//Soglia massima di rimbalzi per riflessione
int MaxDepthRif = 3;

//Flag che attiva il vetro per la clessidra
bool glass = true;

//Variabile che indica se sta calcolando illuminazione diretta o no (non
    modificare)
bool DIR = true;


//Costruttore della classe Render
Render::Render()
{

    //Definizione degli elementi della scena

    //Scatola -> Box
    //Costruttore Piano: new Plane( Posizione , Colore , Normale)

    shapeList[0] = new Plane(vec3(0,-4,0), vec3(0.7,0.7,0.7), vec3(0,1,0)); //piano
        base
    shapeList[1] = new Plane(vec3(8,0,0), vec3(0.7,0.1,0.1), vec3(-1,0,0)); //piano
        DX
    shapeList[2] = new Plane(vec3(0,0,-35), vec3(0.7,0.7,0.7), vec3(0,0,1));
        //Piano Retro
    shapeList[3] = new Plane(vec3(-7,0,0), vec3(0,0,0.7), vec3(1,0,0));   //piano
        SX
    shapeList[4] = new Plane(vec3(0,8,0), vec3(0.5,0.5,0.5), vec3(0,-1,0)); //piano
        Sup
    shapeList[6] = new Plane(vec3(0,0,2), vec3(0.7,0.7,0.7), vec3(0,0,-1)); //Piano
        Retro
```

```
    //Sfere
    /*
     Gestione Brdf (Parametro Type)
        il parametro type del colore viene gestito nella maniera seguente:
        se = 1 BRDF di Lambert
        se = 2 BRDF Cook Torrance

     Se si sceglie Cook-Torrance, si puo' scegliere il materiale nella classe
         corrispondente all'oggetto
    */

    //Costruttore Sfera:
    //new Sphere( Centro, Colore, Raggio, Tipo, Coefficienti Di Specularita')

    //Costruttore Clessidra:
    //new Hourglass( Centro, Colore, Raggio Massimo, Tipo, Coefficienti Di
        Specularita')

    shapeList[8] = new Sphere(vec3(0,-1,-21), vec3(0.7,0.7,0.7), 3, 1);
        //sfera grande
    shapeList[5] = new Hourglass(vec3(-4, -0.5 ,-17), vec3(0.5,0.5,0.5), 2, 2,
        0.9f); //clessidra
    shapeList[7] = new Sphere(vec3(5,-1,-25), vec3(0.4627,0.235,0.157), 2, 2,
        0.3f); //sfera piccola


    vec3 _LightInt,_LightPos,_LightSize;

    //Definisco una luce a partire da un'area light
    //Costruttore Area Light : new Arealight(Posizione, Dimensione);

    //Luce rettangolare in lunghezza
//    areaLight = new AreaLight(vec3(0,8,-23),vec3(1, 0.1f, 12));

    //Luce rettangolare in larghezza
//    areaLight = new AreaLight(vec3(-5.5,8,-17),vec3(12, 0.1f, 1));

    //Luce quadrata
    areaLight = new AreaLight(vec3(-1,8,-17),vec3(3, 0.1f, 3));

    //Parametri di luce
    _LightInt = vec3(0.7,0.7,0.7);
    _LightPos =
        vec3(areaLight->position.x,areaLight->position.y,areaLight->position.z);
    _LightSize = vec3(areaLight->size.x,areaLight->size.y,areaLight->size.z);

    //Definizione della luce
```

```
    light = new Light(_LightInt, _LightPos, _LightSize);
}




/*
   Per calcolare l'illuminazione diretta si utilizza il Ray Tracing classico
   - se il raggio mandato dall'osservatore colpisce un oggetto in un punto p,
   si controlla se quel punto e' in luce, tracciando un raggio d'ombra che ci da'
   il valore dell'illuminazione del punto in coordinate RGB.
   Per fare cio' si campiona uniformemente la sorgente:
   per ogni lato della sorgente si mandano sampleshadows raggi,
   trovando cosi' sampleshadows*sampleshadows punti.
   Per ognuno di essi si traccia il raggio che li congiunge con il punto p.
   Se il raggio interseca un'altro oggetto allora e' in ombra.
   Altrimenti si calcola l'illuminazione diretta in quel punto con la formula:

                          Le * BRDF * G
                         --------------
                               p

       Le = Intensita' di luce entrante in P dalla sorgente
       G  = Fattore geometrico
       p  = Funzione distribuzione di probabilita', per area light = 1/A[sorg]


    Alla fine di questo procedimento si divide per il numero totale di raggi
       d'ombra.
*/

vec3 Render::directIllumination(int shapeHit, vec3 p, Ray* view, int depth)
{
    //Colore risultante
    vec3 ris = vec3(0);

    //Riflessione
    vec3 reflectColor = vec3(0);

    //Ae supera il numero di rimabalzi restuturisce vec3(0)
    if(depth > MaxDepthRif) return ris;

    //Se colpisce una superficie
    if(shapeHit!=-1)
    {

        //Visualizzazione luce
```

```
if(shapeHit==4 && p.x<areaLight->position.x+areaLight->size.x &&
    p.x>areaLight->position.x && p.z>areaLight->position.z &&
    p.z<areaLight->position.z+areaLight->size.z)
{
        return vec3(1);
}


//Normale della superfice nel punto p
vec3 normal = normalize(shapeList[shapeHit]->CalculateNormal(p));

//Calcolo riflessione
if((shapeList[shapeHit]->mat.type == 2 && depth<=MaxDepthRif && DIR &&
    shapeList[shapeHit]->mat.ks>0 && !glass) || (shapeHit==7 &&
    depth<=MaxDepthRif && DIR && shapeList[shapeHit]->mat.ks>0))
{
    //Raggio riflesso con formula -> R = 2 * (N,V)* N - V
    //Dove N e' la normale della superifice
    //e V il vettore che indica la direzione osservata.
    vec3 ReflectionRayDir = normalize(2 *
        (dot(normal,view->direction))*normal - view->direction);
    vec3 ReflectionRayOrigin = p + normal * 1e-4f;
    Ray *ReflectionR = new Ray(ReflectionRayOrigin,-ReflectionRayDir);

    //Calcolo intersezione in direzione del raggio riflesso
    float minT= INFINITY;
    int refHit=-1;
    float t0 = 0.0f;
    for(int k=0; k<NumEl;k++)
    {
        bool hit = shapeList[k]->Intersection(&t0, ReflectionRayOrigin, -
            ReflectionRayDir);
        if(hit && t0<minT)
        {
           minT=t0;
           refHit=k;
        }
    }
    //Punto colpito dal raggio riflesso
    vec3 r = ReflectionR -> origin + ( -minT * ReflectionR -> direction);

    //Calcolo l'illuminazione in quel punto
    reflectColor = reflectColor + shapeList[shapeHit]->mat.Cs *
        directIllumination(refHit, r, ReflectionR, depth+1);

}

//Calcolo rifrazione
```

```
if(shapeHit == 5 && depth<=MaxDepthRif && DIR &&
    shapeList[shapeHit]->mat.ks>0 && glass && shapeList[shapeHit]->mat.type
    == 2)
{
    //Indice di rifrazione del materiale
    float etar = 0.5f;

    //Calcolo del raggio rifratto -> da legge di Snell
    //T = (n * - sqrt(1 - n^2 * ( 1 - (N,V)^2 ))) * N - n * V
    //con N normale nel punto
    //V raggio di visuale
    //n indice di rifrazione del materiale
    vec3 RifractionRayDir = normalize((etar * (dot(normal,view->direction))
        - sqrtf(1 - etar * etar * (1.0f -
        pow(dot(normal,view->direction),2))))*normal - etar *
        view->direction);
    vec3 RifractionRayOrigin = p + normal * 1e-4f;
    Ray *RifractionR = new Ray(RifractionRayOrigin,-RifractionRayDir);

    //Calcolo intersezione in direzione del raggio rifratto
    float minT= INFINITY;
    int refHit=-1;
    float t0 = 0.0f;
    for(int k=0; k<NumEl;k++)
    {
        bool hit = shapeList[k]->Intersection(&t0, RifractionRayOrigin, -
            RifractionRayDir);
        if( k != 5)
        {
            if(hit && t0<minT)
            {
                minT=t0;
                refHit=k;
            }
        }
    }

    //Punto colpito dal raggio rifratto
    vec3 r = RifractionR -> origin + ( - minT * RifractionR -> direction);


    //Calcolo l'illuminazione in quel punto
    reflectColor = reflectColor + shapeList[shapeHit]->mat.Cs *
        computeRadiance(refHit, r, RifractionR, depth+1, 0);
}

//Numero campionamenti per raggi d'ombra
/*
 Se sto calcolando l'illuminazione diretta utilizzo
```

```
 piu' raggi d'ombra per ottenere ombre piu precise e realistiche.

 Se sto calcolando l'illuminazione indiretta utilizzo meno raggi d'ombra,
 questo aumentera' un po' la varianza ma diminuira'
 notevolmente la durata del calcolo.
*/

float Nss;
if(DIR)
{
    shapeList[shapeHit]->dir=true;
    Nss = sampleShadows;
}
else
{
    shapeList[shapeHit]->dir=false;
    Nss = sampleShadowsInd;
}



//Totale raggi d'ombra
float totalRays = Nss * Nss;

//Definizione distribuzione dei raggi per lato
float softIncrementX = light->LightSize.x / Nss;
float softIncrementZ = light->LightSize.z / Nss;


//Verifico che l'oggetto sia in luce
vec3 app = light->LightPos;
vec3 ShadowsRay;
if(sampleShadows>0)
{
       //Per ogni campione sulla sorgente
       for(float m=light->LightPos.x;
           m<light->LightSize.x+light->LightPos.x; m += softIncrementX)
       {
           for(float n=light->LightPos.z;
               n<light->LightSize.z+light->LightPos.z; n += softIncrementZ)
           {


                  //Definisco raggio d'ombra a partire dal punto campionato
                      sulla sorgente
                  app = vec3(m,app.y,n);
                  ShadowsRay = normalize(app - p);
```

```
                //Variabile booleana che indica se il raggio colpisce
                    altri oggetti della scena
                bool lightShapeHit=false;
                int ShadowHit = -1;

                //Verifico se il raggio interseca una superficie
                for(int l = 0; l < NumEl && lightShapeHit == false; l++)
                {
                    bool lightHit = shapeList[l]->Trace(p , ShadowsRay);
                    if(lightHit)
                    {
                        lightShapeHit=true;
                        ShadowHit = l;
                    }
                }

                //Se non colpisce un oggetto
                if(!lightShapeHit)
                {
                    //Calcolo l'illuminazione indiretta

                    //Versore della luce
                    vec3 lightRay = normalize(app - p);

                    //Funzione distribuzione di probabilita' per area
                        light
                    float Area = light->LightSize.x * light->LightSize.z;
                    float py = 1/Area;

                    //Fattore geometrico
                    float G = glm::
                        max(0.0f,dot(lightRay,normal))/(float)M_PI;
/*

                  L[dir] =
                     (1/Ns)* Somma di [Le * BRDF * G * V/p(y)]

                dove la somma e' su tutti gli Ns campioni y, e
                 Ns   = numero raggi d'ombra  G = fattore geometrico
                 Le   = Intensita' di luce entrante in x dalla sorgente
                 p(Y) = funzione distribuzione di probabilita'-> per area
                    light = 1/A[sorg]
*/


                    ris += light->LightInt * shapeList[shapeHit]->BRDF(p,
                        lightRay, view->direction)*G/py;
                }
                //Se l'oggetto e' di vetro lascia passare parte del raggio
```

```
                        else if(ShadowHit==5 && shapeHit!=5 && glass)
                        {
                            ris = ris + shapeList[shapeHit]->mat.Cd*0.2f;
                        }


                    }
                }

            //Si divide per il numero di raggi totali
            ris = ris / totalRays + shapeList[shapeHit]->mat.ks * reflectColor;
        }
     }
    return ris;
}

//Creo un sistema di coordinate locali in base alla normale
void Render::createCoordinateSystem(vec3 N, vec3 &v1, vec3 &v2)
{

    //Creo vettore ortonormale alla normale
    if(fabs(N.y)<fabs(N.x))
    {
        v1 = vec3(N.z, 0, -N.x)/sqrt(N.x * N.x + N.z * N.z);
    }
    else
    {
        v1 = vec3(0, -N.z, N.y)/sqrt(N.y * N.y + N.z * N.z);
    }

    //Prodotto vettoriale per creare terzo vettore e lo normalizzo
    v2= cross(N,v1);
    v2 = normalize(v2);
}



/*
   Campionamento Uniforme
   Genera campioni uniformemente distribuiti sull'emisfero.
   e' il metodo piu' semplice da implementare, ma genera anche molto rumore
      nell'immagine.
*/
vec3 Render::uniformSampleHemisphere(float &u1, float &u2, float &pdf)
{
    /*
     Generico punto sull'emisfero

        x = sin(Theta) * cos(Phi)
```

```
        y = cos(Theta)
        z = sin(Theta) * sin(Phi)
    */

    // distribuzione di THETA = cos^-1(1-u1) --> THETA = cos^-1(v1) --> v1 =
        cos(THETA)
    // cos(Theta) = u1 = y
    // sin(Theta)^2 = 1 - cos(Theta)^2
    float sinTheta = sqrtf(1 - u1 * u1);

    //Distribuzione di phi = 2 pigreco * u2
    float phi = 2 * M_PI * u2;

    float x = sinTheta * cosf(phi);
    float z = sinTheta * sinf(phi);

    //Distribuzione di probabilita' per campionamento uniforme
    pdf = 1/(2 * M_PI);

    return vec3(x, u1, z);
}




/*
   Campionamento Per Coseno
   A differenza del campionamento uniforme,
   le direzioni generate sono proporzionali al coseno.
   In questo modo non vengono considerati i campioni vicino all'equatore
   perche' danno un contributo quasi nullo.
   Questo procedimento comporta una notevole diminuzione del rumore nell'immagine.
*/
vec3 Render::CosineSampleHemisphere(float &u1, float &u2, float &pdf)
{
    /*
     Generico punto sull'emisfero

        x = sin(Theta) * cos(Phi)
        y = cos(Theta)
        z = sin(Theta) * sin(Phi)
    */

    // distribuzione di Theta = arccos(sqrt((1-u1)) --> Theta = arccos(sqrt(v1))
        --> sqrt(v1) = cos(Theta)
    // poiche' v1 e' in [0,1] posso cosiderare u1 = v1
    // cos(Theta) = sqrt(u1);
    // sin(Theta)^2 = 1 - cos(Theta)^2
    float cosTheta = sqrtf(u1);
```

```
    float sinTheta = sqrtf(1-u1);

    //Distribuzione di Phi = 2 pigreco * u2
    float phi = 2 * M_PI * u2;

    float x = sinTheta * cosf(phi);
    float z = sinTheta * sinf(phi);

    //Distribuzione di probabilita' per campionamento del coseno
    pdf = u1 / (float) M_PI;

    return vec3(x, cosTheta, z);
}



/*
   Per calcolare l'illuminazione indiretta si utilizza l'estimatore di Montecarlo
   -sia p il punto osservato
    a partire dalla normale in tale punto, si crea un sistema di coordinate locali.
    Si generano saplemc campioni di Montecarlo utilizzando
    il campionamento del coseno o il campionaemnto uniforme.
    Per ognuno di questi raggi si calcola l'intersezione con la scena, trovando un
        punto y.
    Si calcola la radianza in y ottenendo cosi' un algoritmo ricorsivo.
    La ricorsione si ferma se si raggiunge una soglia predefinita di rimabalzi
    o tramite l'algoritmo della roulette russa, che sfrutta la riflettanza
        dell'oggetto.
    Per ogni campione si calcola quindi

                    L[ind] = cos(THETA) * BRDF * Lr / PDF

            dove
                    THETA = angolo tra normale e campionamento
                    Lr  = radianza entrante in x in direzione del campionamento
                    PDF = funzione distribuzione di probabilita'

    Alla fine di questo processo si divide per il numero totale di campioni.
*/

vec3 Render::indirectIllumination(int shapeHit, vec3 p, Ray* view, int depth)
{
    //Risultato
    vec3 ris = vec3(0);

    //Variabile random per roulette russa
    float U = drand48();

    //Assorbimento
```

```
    float abs;

    //Assorbimento proporzionale alla Brdf
    vec3 lightRay = normalize(light->LightPos - p);
    vec3 fr = shapeList[shapeHit]->BRDF(p, lightRay, view->direction);
    abs = fr.x > fr.y && fr.x > fr.z ? fr.x : fr.y > fr.z ? fr.y : fr.z;

    //Numero massimo rimbalzi
    if(depth>MaxDepth) return ris;

    //Roulette Russa
    if(U >= (1- abs)) return ris;

    //Vettori per coordinate locali
    vec3 v1,v2, normal;

    //Calcolo normale
    normal = shapeList[shapeHit]->CalculateNormal(p);

    //Creo sistema di coordinate in base alla normale
    createCoordinateSystem(normal,v1,v2);

    //Funzione distribuzione di probabilita'
    float pdf;

    //Per ogni campione di montecarlo
    for(int i=0;i<sapleMC;++i)
    {
        vec3 sample;

        //genero 2 numeri casuali tra 0 e 1
        float u1 = rand() / (RAND_MAX+1.0);
        float u2 = rand() / (RAND_MAX+1.0);

        //Campionamento uniforme dell'emisfero secondo u1 e u2
//        sample = uniformSampleHemisphere(u1,u2, pdf);

        //Campionamento del coseno dell'emisfero secondo u1 e u2 con u1 = cos(Theta)
        sample = CosineSampleHemisphere(u1,u2,pdf);

        //Trasformazione del vettore campionato con coordinate locali in coordinate
            globali,
        //Per definire una direzione nello spazio
        /*
                                          | V2x V2y V2z |    | Sx*V2x + Sy*Nx +
                                             Sz*V1x |
           sample[WORLD] = sample *  | Nx   Ny   Nz | =  | Sx*V2y + Sy*Ny + Sz*V1y |
                <--   prod righe * colonne
                                        | V1x V1y V1z |    | Sx*V2z + Sy*Nz + Sz*V1z |
```

```
*/

vec3 sampleHemisphere = vec3(sample.x * v2.x + sample.y * normal.x +
    sample.z * v1.x,
                            sample.x * v2.y + sample.y * normal.y + sample.z
                                * v1.y,
                            sample.x * v2.z + sample.y * normal.z + sample.z
                                * v1.z);

sampleHemisphere = normalize(sampleHemisphere);

float minT= INFINITY;
int shapeHit_Sec=-1;
float t = 0.0f;

//Calcolo intersezione da p lungo samplehemisphere
for(int k=0; k<NumEl; k++)
{
    bool hit = shapeList[k]->Intersection(&t,p,-sampleHemisphere);
    if(hit && t<minT)
    {
        minT=t;
        shapeHit_Sec=k;
    }
}

//Se non interseca niente return vec3(0)
if (shapeHit_Sec == -1) return ris;

//Nuovo raggio da p in direzione samplehemisphere
Ray *psi = new Ray(p,-sampleHemisphere);

//Nuovo punto
vec3 y = psi->origin + (-minT * psi->direction);

//Calcolo della radianza nel nuovo punto
DIR = false;
shapeList[shapeHit]->dir=true;
vec3 radianza = computeRadiance(shapeHit_Sec,y,psi,0,depth+1);
DIR = true;

//Calcolo radianza risultante nel punto iniziale
/*
     L[ind] = (1/N)* Somma (su tutti gli N campioni) di [cos(THETA) * BRDF *
         Lr / PDF]

     dove
          N   = numero campionamenti MC  THETA = angolo tra normale e
              campionamento
```

```
                Lr  = radianza entrante in x in direzione del campionamento
                PDF = funzione distribuzione di probabilita'
        */

        ris += (u1 * radianza * shapeList[shapeHit]->BRDF(p, -psi->direction,
            view->direction))/pdf;
        shapeList[shapeHit]->dir=DIR;
    }

    //Si divide per il numero di campioni
    ris = ris / (float)(sapleMC);

    //Si divide per 1 - assorbimento
    ris = ris / (float)(1 - abs);

    return ris;
}


/*
   Per calcolare l'illuminazione globale si agisce nel seguente modo:
   calcolare il contributo di illuminazione diretta,
   calcolare il contributo di illuminazione indiretta,
   sommare i due contributi.
*/
vec3 Render::computeRadiance(int shapeHit, vec3 p, Ray* view, int depth, int
    depht1)
{
    vec3 radianza = vec3(0,0,0);
    //Se colpisce un elemento calcola la radianza nel punto
    if(shapeHit!=-1)
    {
//       if (depht1>0)   //Decommentare per calcolare solo illuminazione indiretta
        //Calcolo Illuminazione Diretta
        radianza += directIllumination(shapeHit, p, view, depth);

        //Calcolo Illuminazione Indiretta
        radianza += indirectIllumination(shapeHit, p, view, depht1); //Commentare
            per calcolare solo illuminazione diretta

    }
    return radianza;
}


/*
   Per ogni pixel dell'immagine si definisce il raggio che parte dall'osservatore
   e passa per il centro del pixel.
   Dato questo raggio si verica l'intersezione con gli oggetti della scena.
```

```
   Se c'e' intersezione si va a calcolare la radianza nel punto trovato.
*/
vec3** Render::computeImage(vec3 RayOrigin,int width,int height)
{

    //Definizione di una matrice di vec3 che rappresenta l'immagine
    vec3 **image = new vec3*[width];
    for(int i=0; i<width;i++) image[i] = new vec3[height];

    //Contatore percentuale
    int c=0;

    //Per ogni pixel dell'immagine
    for(int i=0; i<width; i++)
        {
            for(int j=0; j< height; j++)
            {
                //Definizione dei parametri dei pixel per ottenere un sistema di
                    coordinate
                //normalizzo le coordinate in modo che siano in [0,1]
                float pixelNormX = (i + 0.5) / width;
                float pixelNormY = (j + 0.5) / height;
                float AspectRatio = width / height; //rapporto dell'immagine
                //mappo i pixel su [-1,1]
                float pixelMapX=(2 * pixelNormX - 1) * AspectRatio;
                //inverto l'asse y
                float pixelMapY= 1 - 2 * pixelNormY;
                //setto i parametri della camera
                float pixelCameraX = -pixelMapY * tan(radians(65.0) / 2);
                float pixelCameraY = -pixelMapX * tan(radians(65.0) / 2);
                vec3 CameraSpace = vec3(pixelCameraX, pixelCameraY,-1);

                /*
                     SISTEMA DI COORDINATE:
                           y
                           |
                           |         x
                           . - - - - -
                          /
                       z /
                */

                //Creo raggio passante dall'origine e dal pixel che sto considerando
                vec3 RayDirection = CameraSpace - RayOrigin;
                RayDirection = - normalize(RayDirection);
                Ray *view = new Ray(RayOrigin,RayDirection);

                //Si cerca intersezione piu' vicina
                float minT= INFINITY; //Distanza minima
```

```
                int shapeHit=-1;     //Indice elemento colpito
                float t = 0.0f;      //Distanza punto colpito

                for(int k=0; k<NumEl;k++)
                {
                    bool hit = shapeList[k]->Intersection(&t, RayOrigin,
                        RayDirection);
                    if(hit && t<minT)
                    {
                        minT=t;
                        shapeHit=k;
                    }
                }

                //Punto trovato
                vec3 p = view->origin + (-minT * view->direction);

                //Calcolo della radianza nel punto trovato
                image[i][j] = computeRadiance(shapeHit,p,view,0,0);

            }
            //Calcolo percentuale
            c = c + height;
            cout<< c*100/(width*height)<<"\%"<<endl;
        }
        return image;
}
```

CAPITOLO 20

Appendice: Versione in Java del renderer di illuminazione globale della Parte 3

Nei capitoli 11, 12, 13 e 14 abbiamo presentato un codice completo di Illuminazione Globale, scritto per semplicità nel linguaggio C, a parte il modulo principale main/cpp in C++. Per comodità del lettore inesperto, qui ne diamo invece una variante scritta in Java, con commenti dettagliati soprattutto nella classe principale Main.
Il codice è stato elaborato in versione preliminare in collaborazione con Laura Laurenti [**30**] e poi espanso, approfondito ed ottimizzato da Domenico Verde [**47**], e sviluppa un renderer di radiosità stocastica seguito da Final Gathering. Come in alcune parti del codice C++/C illustrato nei capitoli 11, 12, 13 e 14, al fine di modificare la radiosità per simulare effetti di trasparenza, viene aggiunto un raggio rifratto nel corso del Final Gathering: questo crea immagini non fotorealistiche. Il raggio rifratto può essere invece utilizzato in un renderer di Ray Tracing, oppure in un renderer Multipass nella fase del Ray Tracing. Il package di interfaccia utente è alla fine del codice.

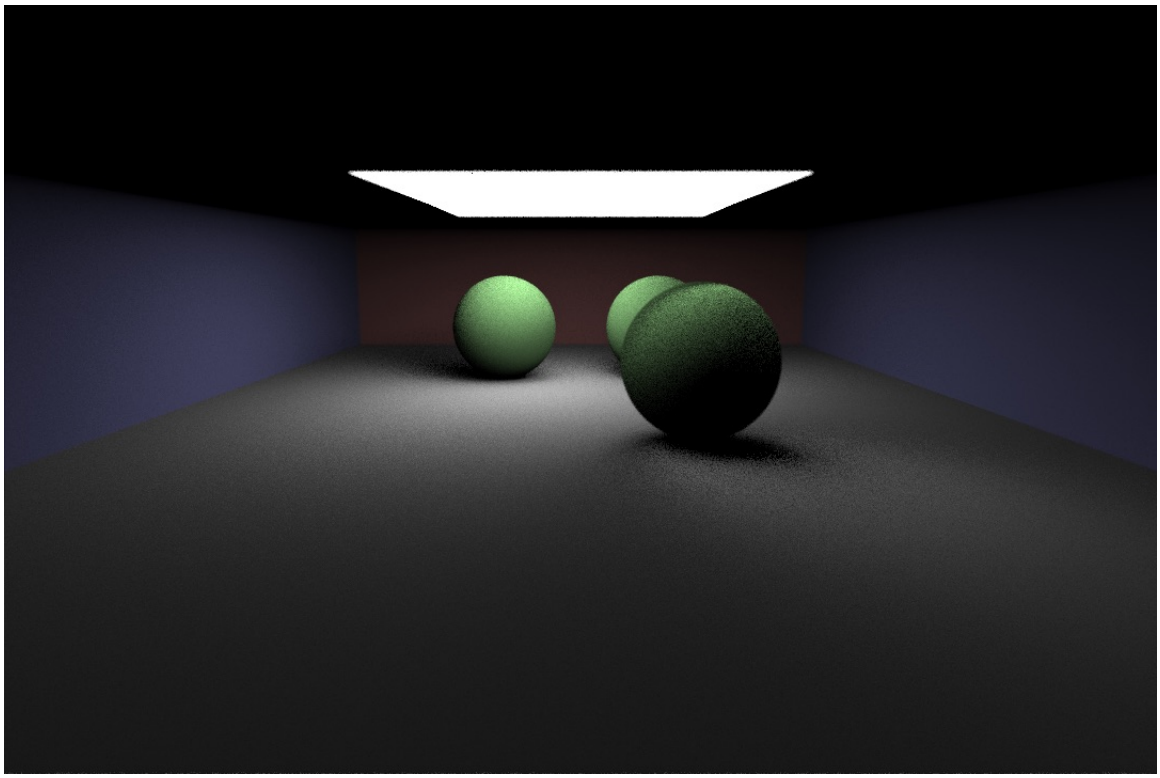


FIGURA 20.0.1. Tre sfere diffusive, rese con algoritmo di radiosità stocastica seguito da Final Gathering e raggio rifratto aggiuntivo.

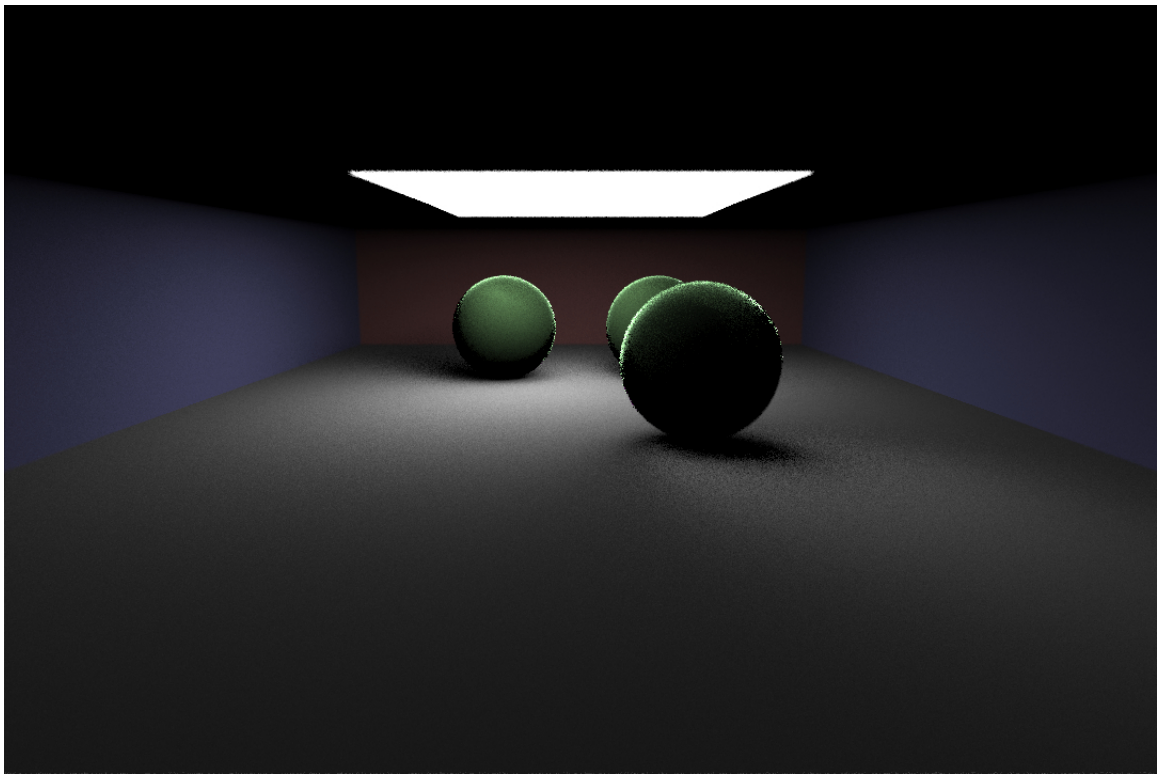


FIGURA 20.0.2. Tre sfere traslucenti con BRDF data dall'algoritmo empirico del dipolo di Jensen, rese con Final Gathering.

Classe Main:

```
//Questo codice calcola la radiosita' di una scena in
//base al metodo iterativo di Jacobi stocastico.
//Indicazioni per l'utente:
//Ci sono due scelte per la visualizzare dell'immagine
//finale: una di radiosita' pura ed una di radiosita'
//seguita da illuminazione riflessa indiretta
//(impostando true il parametro doFinalGathering) che permette di
//ottenere un'immagine fotorealistica
//Si puo' scegliere se visualizzare sfere
//di vetro o di giada impostando true o false
//(solo uno puo' essere true) i parametri
//jade e glass.
//Si puo' inoltre scegliere se visualizzare allineate
//o sovrapposte impostando true o false il parametro
//aligned (la scelta true e' consigliata per
//le sfere in vetro, cosi' da poter apprezzare le
//multiriflessioni).
```

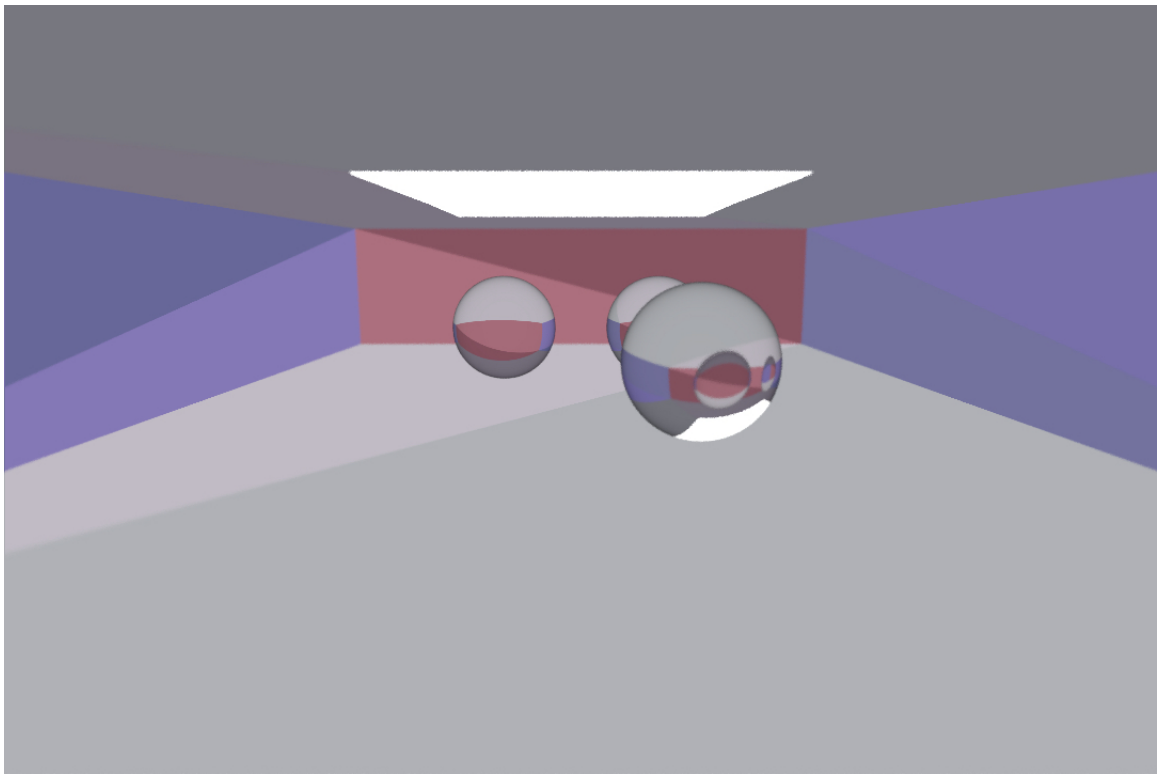


FIGURA 20.0.3. Tre sfere di vetro, rese con radiosità stocastica ed in aggiunta tracciando un raggio rifratto. Per risparmiare tempo nel rendering, le pareti della stanza sono modellate con solo due elementi ciascuna, e quindi, senza un passaggio di Final Gathering, appaiono come triangoli di colore piatto.

```
//Tutti i parametri tra cui scegliere si possono
//trovare nelle prime righe di codice
//Si specifica che agendo sul codice e' possibile
//comunque apportare le volute modifiche sui
//materiali, sulla posizione delle sfere, etc.

import ui.InterfaceInitialiser;

public class Main {
  public static void main(String[] args) {
    new InterfaceInitialiser();
  }
}
```

Classe Point3D:

```
package primitive;
```

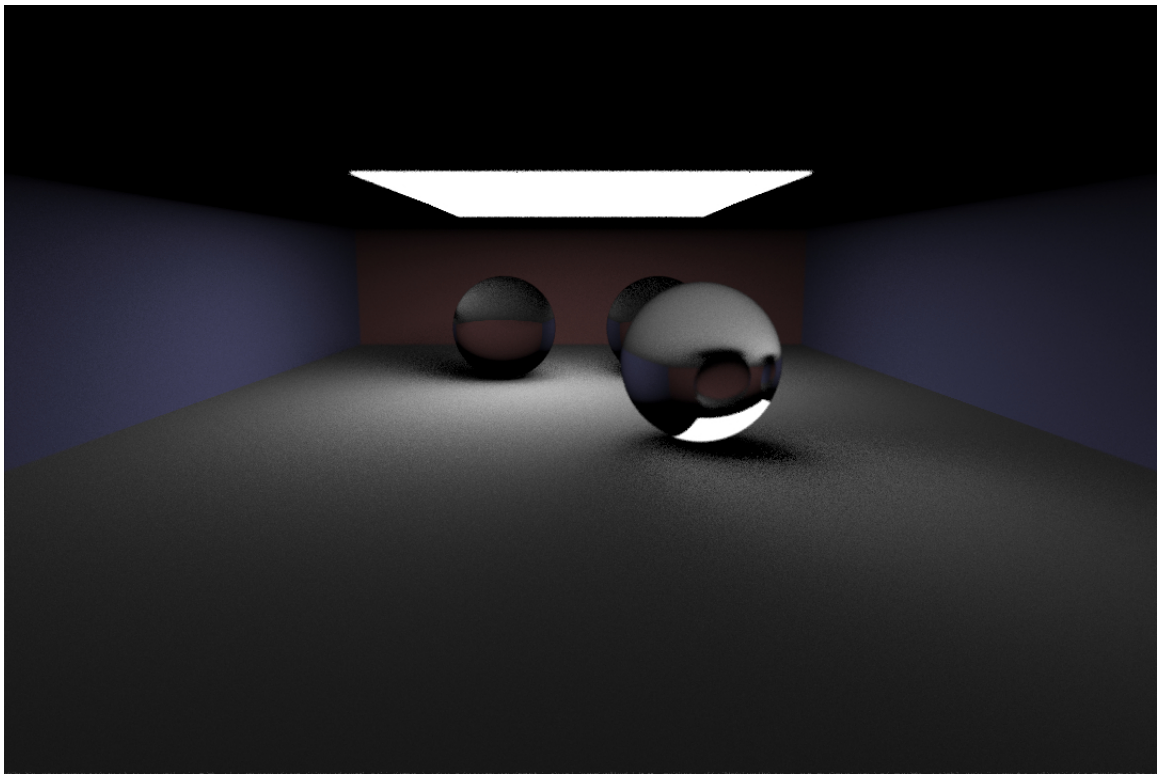


FIGURA 20.0.4. Tre sfere di vetro, rese con algoritmo di radiosità stocastica seguita da Final Gathering, e tracciando anche un raggio rifratto. Poiché alle sfere di vetro sono stati assegnata alti coefficienti di trasparenza e riflettività speculare ma basso coefficiente di diffusione, esse appaiono scure con questo rendering basato sulla radiosità, quindi sulla capacità diffusiva.

```
//questa classe definisce un float3: esso puo'
//rappresentare un punto o un vettore in R3 a seconda
//delle esigenze
//Dal momento che la scena che vogliamo rendere
//e' una stanza nello spazio R3, questa classe sara'
//molto utilizzata, insieme anche alle sue funzioni che si
//occupano di varie trasformazioni o operazioni con i
//vettori.

import renderer.Utilities;

import java.awt.*;

public class Point3D {
   //ha come parametri soltanto le tre componenti di un
   //float in uno spazio a 3 dimensioni
   public double x;
   public double y;
   public double z;
```

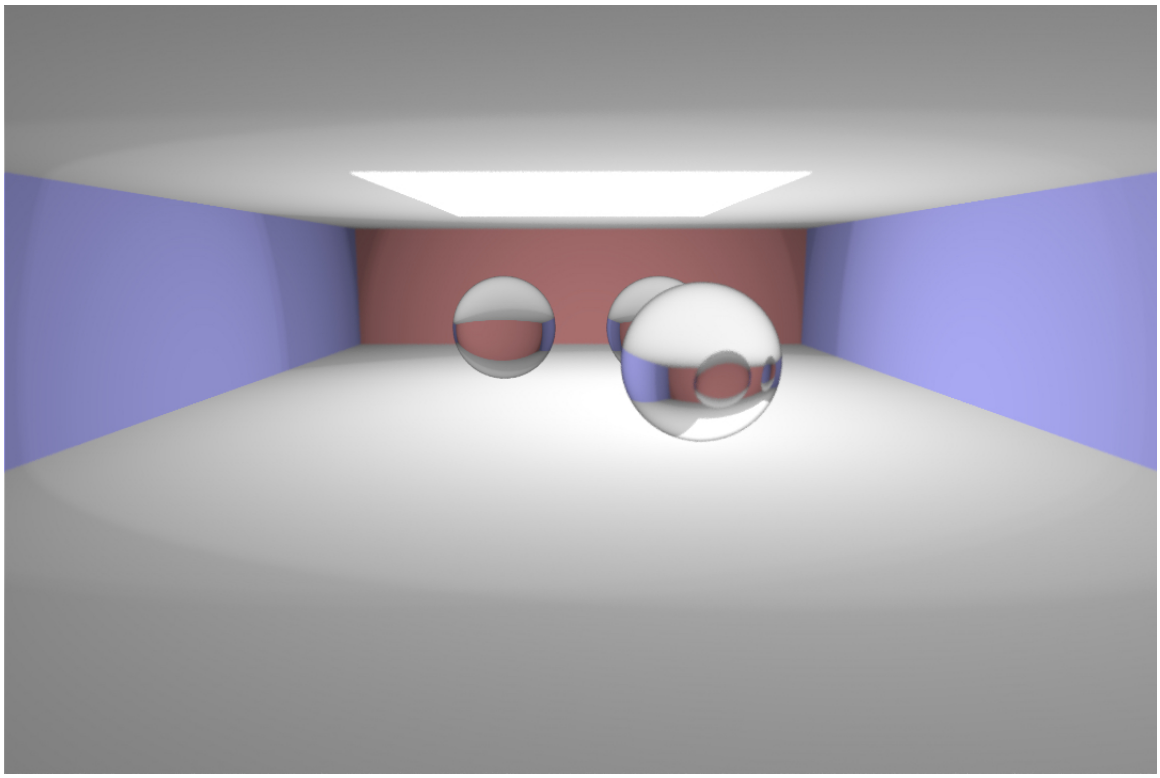


FIGURA 20.0.5. Tre sfere di vetro, rese con Photon Mapping: 80 fotoni generici, 100 fotoni nella mappa delle caustiche, 60 fotoni per le caustiche indirette; le caustiche dirette si riferiscono al trasporto ella luce direttamente da lampade a superficie speculari o rifrangenti (specchi e bocce di vetro); le caustiche indirette si riferiscono a nuovi percorsi che vengono iniziati quando un fotone colpisce una sfera di vetro o uno specchio. Tempo di esecuzione: 41 minuti su Intel Core i9 8-Core 2.4 GHz.

```
//costruttori
public Point3D() {
   x = 0;
   y = 0;
   z = 0;
}

public Point3D(double x_, double y_, double z_) {
   x = x_;
   y = y_;
   z = z_;
}

public Point3D(double v) {
   x = v;
   y = v;
   z = v;
```

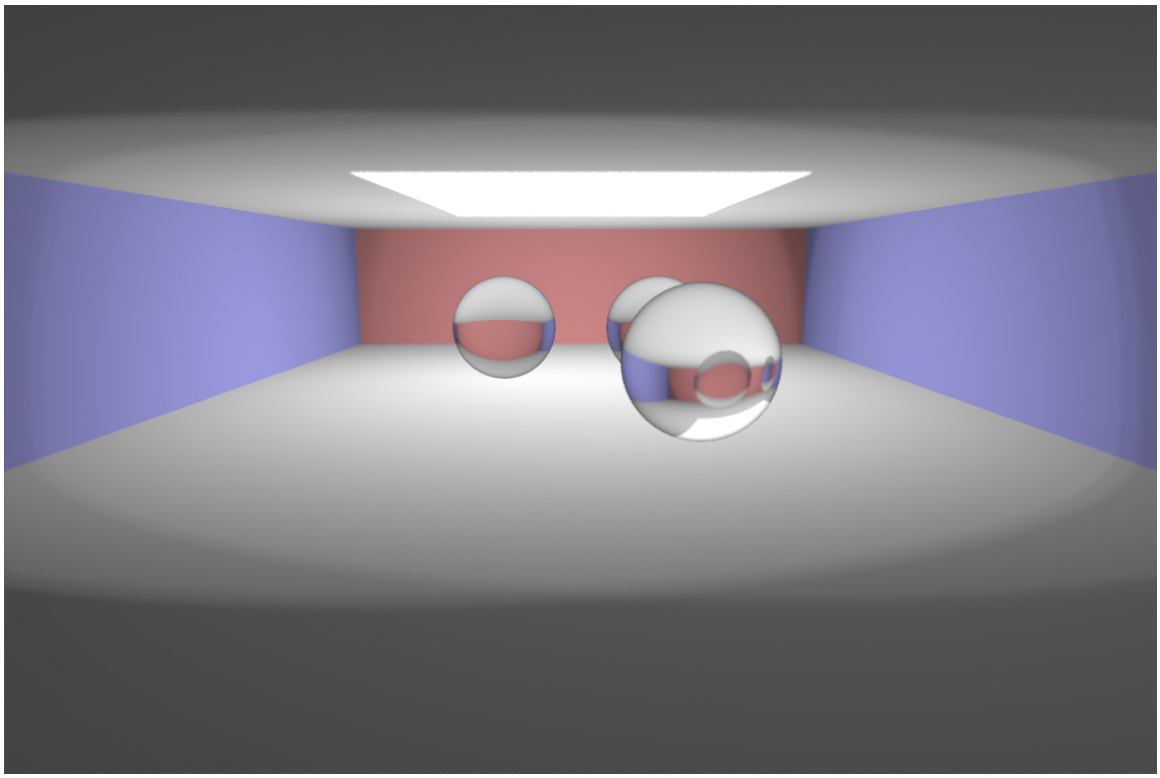


FIGURA 20.0.6. Tre sfere di vetro, rese con Photon Mapping: 90 fotoni generici, 100 fotoni nella mappa delle caustiche, 80 fotoni per le caustiche indirette. Tempo di esecuzione 2 ore e 47 minuti su Intel Core i9 8-Core 2.4 GHz. Si noti come il maggior numero di fotoni rende le penombre di quella dellpiù. graduali che nella precedente Figura 20.0.5.

```
}

//metodi set e get
public void setX(double newX) {
   x = newX;
}

public void setY(double newY) {
   y=newY;
}

public void setZ(double newZ) {
   z=newZ;
}

public double getX() {
   return x;
}

public double getY() {
```

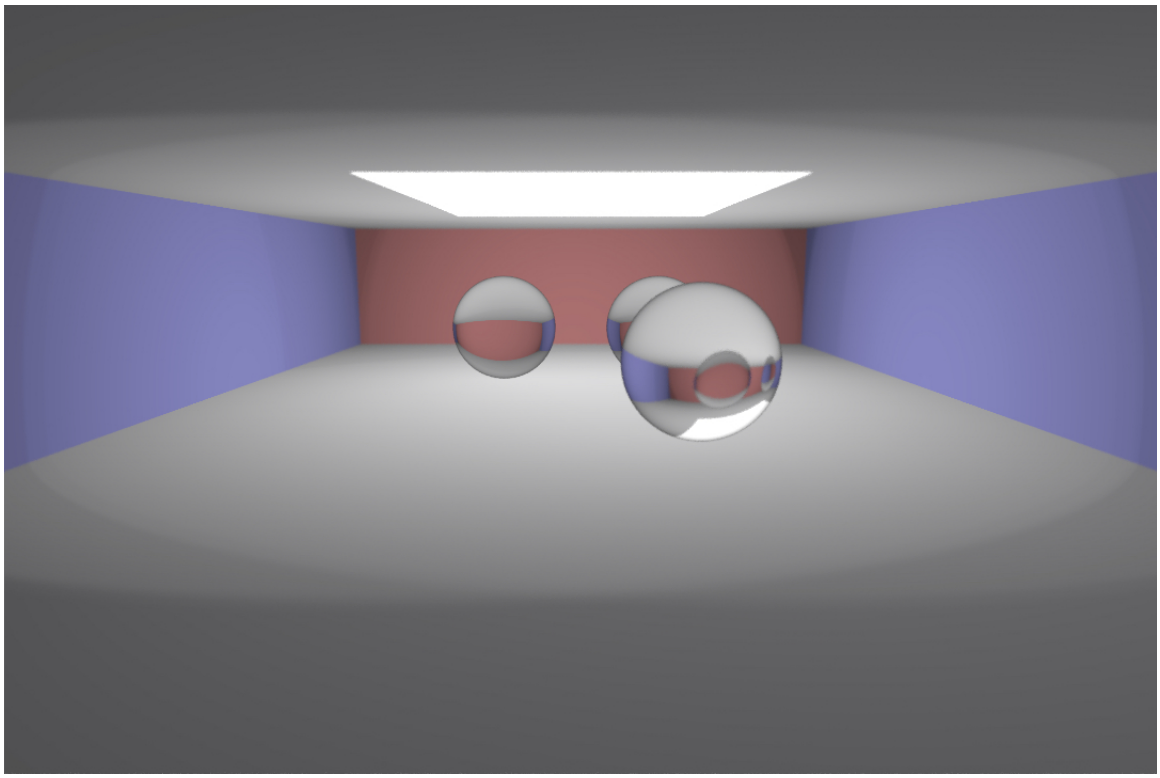


FIGURA 20.0.7. Tre sfere di vetro, rese con Photon Mapping: 100 fotoni generici, 150 fotoni nella mappa delle caustiche, 100 fotoni per le caustiche indirette. Si noti come il maggior numero di fotoni rende le penombre assai meglio sfumate di quelle delle precedenti Figura 20.0.5 e 20.0.6, ma a prezzo di un notevole aumento del tempo di calcolo. Infatti il tempo di esecuzione è stato di circa 20 ore su Intel Core i9 8-Core 2.4 GHz.

```
   return y;
}

public double getZ() {
   return z;
}

// Sum applica una traslazione al punto
void sum(Point3D point3D) {
   x += point3D.x;
   y += point3D.y;
   z += point3D.z;
}

//addizione componente componente tra 2 vettori in R3
// Non modifica il punto
public Point3D add(Point3D b) {
   Point3D a = new Point3D(x,y,z);
   a.x += b.x;
```

```
      a.y += b.y;
      a.z += b.z;
      return a;
   }

   //sottrazione componente componente tra 2 vettori in R3
   // Non modifica il punto
   public Point3D subtract(Point3D b) {
      Point3D a=new Point3D(x,y,z);
      a.x=a.x-b.x;
      a.y=a.y-b.y;
      a.z=a.z-b.z;
      return a;
   }

   //moltiplicazione tra ogni componente di un vettore
   //in R3 con uno scalare b
   public Point3D multiplyScalar(double b) {
      Point3D a=new Point3D(x,y,z);
      a.x=a.x*b;
      a.y=a.y*b;
      a.z=a.z*b;
      return a;
   }

   //divisione tra ogni componente di un vettore in R3
   //con uno scalare b
   public Point3D divideScalar(double b) {
      Point3D a=new Point3D(x,y,z);
      a.x=a.x/b;
      a.y=a.y/b;
      a.z=a.z/b;
      return a;
   }

   //moltiplicazione componente componente tra 2
   //vettori in R3
   public Point3D multiplyComponents(Point3D b) {
      Point3D a=new Point3D(x,y,z);
      a.x=a.x*b.x;
      a.y=a.y*b.y;
      a.z=a.z*b.z;
      return a;
   }

   //divisione componente componente tra 2 vettori in R3
   public Point3D divideComponents(Point3D b) {
      Point3D a=new Point3D(x,y,z);
      a.x=a.x/b.x;
```

```
   a.y=a.y/b.y;
   a.z=a.z/b.z;
   return a;
}

//modulo
public void abs() {
   if (x<0) x=-x;
   if (y<0) y=-y;
   if (z<0) z=-z;
}

//norma euclidea
public double normalize() {
   Point3D a=new Point3D(x,y,z);
   return Math.sqrt(a.x*a.x+a.y*a.y+a.z*a.z);
}

//norma al quadrato
public double squareNorm() {
   Point3D a=new Point3D(x,y,z);
   return a.x*a.x+a.y*a.y+a.z*a.z;
}

//media delle componenti di un vettore in R3
public double average() {
   Point3D a=new Point3D(x,y,z);
   return (a.x+a.y+a.z)/3;
}

//norma del vettore
public Point3D getNormalizedPoint() {
Point3D a=new Point3D(x,y,z);
double n = a.normalize();
double ax=(a.x)/n;
double ay=(a.y)/n;
double az=(a.z)/n;
   return new Point3D(ax,ay,az);
}

// Prodotto scalare
public double dotProduct(Point3D b) {
   Point3D a=new Point3D(x,y,z);
   return a.x*b.x+a.y*b.y+a.z*b.z;
}

// Prodotto vettoriale
public Point3D crossProduct(Point3D b) {
   Point3D a=new Point3D(x,y,z);
```

```
      return new Point3D(a.y*b.z-a.z*b.y,
            a.z*b.x-a.x*b.z,a.x*b.y-a.y*b.x);
   }

   //Cerca la componente massima di un vettore in R3
   public double max() {
      double max=x;
      if(y>x){max=y;}
      if(z>max){max=z;}
      return max;
   }

   //copia del vettore b passato come parametro
   public void copy(Point3D b) {
      x = b.getX();
      y = b.getY();
      z = b.getZ();
   }

   //verificare se sono uguali due vettori
   @Override
   public boolean equals(Object b) {
      if (b instanceof Point3D) {
         Point3D b1 = (Point3D) b;
         return (b1.x == x) && (b1.y == y) && (b1.z == z);
      } else {
         return false;
      }
   }

   //radice del vettore in R3 passato come parametro
   //(fa la radice di ogni componente)
   public static Point3D getSquareCompPoint(Point3D b) {
      Point3D a=new Point3D();
      a.x=(float) Math.sqrt(b.x);
      a.y=(float) Math.sqrt(b.y);
      a.z=(float) Math.sqrt(b.z);
      return a;
   }

   // riflessione di un vettore i rispetto alla normale
   //n: R=2<N,I>N-I
   public static Point3D reflect(Point3D i, Point3D n){
      return n.multiplyScalar(n.dotProduct(i))
                  .multiplyScalar(2.0f)
                  .subtract(i);
   }

  // rifrazione di un vettore i rispetto ad una normale
```

```
//n: T=N(ior<N,I>-sqrt(1-(ior)^2(1-(<N,I>)^2)))-ior*I
// in questa funzione la rifrazione non varia con la
//lunghezza d'onda

public static Point3D getRefraction(Point3D direction, Point3D normal,
      double refractionIndex){
   // direction definito come "i" nei commenti
   // normal definito come "n" nei commenti

 //ci si accerta che il vettore in entrata sia
 //normalizzato
 direction.getNormalizedPoint();

 //si calcola il coseno tra la normale e il vettore
 //entrante i: <n,i>
 double cosThetaI = normal.dotProduct(direction);

 //si prende in esame l'indice di rifrazione ior
 //del materiale. Per semplificare il calcolo viene
 //considerato solamente il passaggio dal vuoto,
 //non e' quindi possibile con questa funzione
 //modellare il passaggio di luce tra due materiali
 //con indice di rifrazione differente
 //indice di rifrazione nel vuoto (=1) / indice di
 //rifrazione del materiale
 double eta=1/refractionIndex;

 //se il coseno dell'angolo tra il vettore i e n e'
 //minore di 0 allora dobbiamo invertire la normale
 //e l'indice di rifrazione, ovvero il passaggio
 //avverra' dal mezzo denso fino al vuoto
 // (di conseguenza il coseno diventera' positivo)
 if (cosThetaI<0) {
    cosThetaI=-cosThetaI;
    eta=(float) (1.0/eta);
    normal = normal.multiplyScalar(-1);
 }

 //calcolo dei fattori necessari
 double sin2ThetaI = 1-(cosThetaI*cosThetaI);
 double sin2ThetaT = eta*eta*sin2ThetaI;

 // se questo coefficiente e' minore di 0 allora
 //avviene una riflessione totale e il resto del
 //calcolo non viene effettuato
 double K= 1 - sin2ThetaT;
 if (K<0) {
    //riflessione totale
       return new Point3D(-1.0f);
```

```
  } else {
     //altrimenti si procede con il calcolo del
     //vettore rifratto
     float cosThetaT= (float) Math.sqrt(K);
     double angle = eta*cosThetaI - cosThetaT;
        return normal.multiplyScalar(angle)
                   .subtract(direction.multiplyScalar(eta));
  }
 }

 public static Point3D exponent(Point3D point) {
     return new Point3D(
                Math.exp(point.getX()),
                Math.exp(point.getY()),
                Math.exp(point.getZ())
     );
  }

 // rifrazione di un vettore i rispetto ad una normale
 //n: T=N(ior<N,I>-sqrt(1-(ior)^2(1-(<N,I>)^2)))-ior*I
 //restituisce primitive.Ray[]t in cui t[0]=R, t[1]=G, t[2]=B
 public static Ray[] getRefraction(Ray[] rays, Point3D direction, Point3D normal,
     Point3D refractionIndex) {
     //utilizziamo la proprieta' maxDepth del raggio per
          //verificare se c'e' riflessione totale
     rays[0].depth = 1;
     rays[1].depth = 1;
     rays[2].depth = 1;

     direction.getNormalizedPoint();
     // <n, i>
  double cosThetaI = normal.dotProduct(direction);
  Point3D eta = new Point3D(1.0f).divideComponents(refractionIndex);
  //se l'angolo di incidenza e' superiore a pi/2
  //allora devo cambiare il verso della normale e
  //cambiare indice di rifrazione (prendendo il suo
  //inverso)

  if (cosThetaI < 0.0f) {
     cosThetaI = -cosThetaI;
     normal = normal.multiplyScalar(-1.0f);
     eta=new Point3D(1.0f).divideComponents(eta);
  }

  Point3D sin2ThetaT = eta.multiplyComponents(eta)
       .multiplyScalar(1 - (cosThetaI*cosThetaI));

   Point3D K = new Point3D(1.0f).subtract(sin2ThetaT);
```

```
 //componente rossa
   if (K.x<0.0f) {
   // TIR
      //radice negativa niente rifrazione
   rays[0].depth=0;
} else {
      float cos_theta_t= (float) Math.sqrt(K.x);
      double angle=eta.x*cosThetaI-cos_theta_t;
   rays[0].d= normal.multiplyScalar(angle);
   rays[0].d= rays[0].d.subtract(direction.multiplyScalar(eta.x));
}

   //componente verde
   if (K.y<0.0f) {
   rays[1].depth=0;
} else {
   float cos_theta_t= (float) Math.sqrt(K.y);
   double angle =eta.y*cosThetaI-cos_theta_t;
   rays[1].d= normal.multiplyScalar(angle);
   rays[1].d= rays[1].d.subtract(direction.multiplyScalar(eta.y));
}

   //componente blu
   if (K.z<0.0f) {
   rays[2].depth=0;
} else {
   float cos_theta_t= (float) Math.sqrt(K.z);
   double angle=eta.z*cosThetaI-cos_theta_t;
   rays[2].d= normal.multiplyScalar(angle);
   rays[2].d= rays[2].d.subtract(direction.multiplyScalar(eta.z));
   }

   return rays;
}

//costringe un numero in [0,1]
 public static double clamp(double v) {
   if(v<0)
      return 0;
   else if (v>1)
      return 1;
   else
      return v;
}

 //costringe le componenti di un vettore in R3 in [0,1]
public static Point3D clamp3(Point3D f) {
 double xf=clamp(f.x);
 double yf=clamp(f.y);
```

```
   double zf=clamp(f.z);
      return new Point3D(xf,yf,zf);
  }

  @Override
   public String toString() {
      return x + " " + y + " " + z;
   }

   public Color toColor() {
      return new Color(Utilities.toInt(x), Utilities.toInt(y), Utilities.toInt(z));
   }
}
```

Classe Triangle:

```
package primitive;

import renderer.Utilities;

//classe che definisce un triangolo attraverso un array di
//float3 contenente i 3 vertici del triangolo, e la sua
//normale.
//La classe ha la funzione che calcola l'intersezione
//del triangolo con un raggio, e la funzione che
//restituisce la normale al triangolo

/* Un problema presente Ăš l'aggiunta di colori sui
 *  punti del triangolo (oppure associati alla patch o
 *  mesh della scena) per permettere l'interpolazione di
 *  Gouraud (per Jacobi stocastico)
 */

/* Come in altre classi, qui si accede al materiale
 *  dell'oggetto tramite un indice che si interfaccia
 *  alla classe statica RenderAction. Sarebbe necessario
 *  associare a ogni oggetto creato il suo materiale,
 *  senza doversi ricollegare a una classe statica.
 * Il secondo rilevante problema Ăš la necessitĂ  di
 *  aggiungere una mesh dinamica per gli oggetti,
 *  cosicchĂš sia piĂč efficiente la traslazione e la
 *  rotazione per oggetti generici.
 */

public class Triangle {
  //array dei vertici
  public Point3D[] vertices;
```

```
//Normale al triangolo
private Point3D n;

int matId;

public boolean isBorderMeshScene = false;

//costruttore:i parametri in input sono i tre vertici
//del triangolo
public Triangle(Point3D v0, Point3D v1, Point3D v2, int matId) {
  // triangolo definito da un array di 3 float3
  vertices = new Point3D[3];
  vertices[0]=v0;
  vertices[1]=v1;
  vertices[2]=v2;

  //calcoliamo la normale con un metodo apposito
  n=tNormalCalc();

  this.matId = matId;
}

//vettore normale e' ottenuto tramite prodotto
//vettoriale di a e b
Point3D tNormalCalc() {
  //calcolo dei vettori appartenenti al piano del
  //triangolo a e b
  Point3D a=vertices[1].subtract(vertices[0]);
  Point3D b=vertices[2].subtract(vertices[0]);
  return (a.crossProduct(b)).getNormalizedPoint();
}

//funzione di intersezione con un raggio: Return
//distanza o -1.0f se non c'e' intersezione
float intersect(Ray r) {
  //risolvo il sistema o+Td=a1+ beta(b1-a1) +
  //gamma(c1-a1) => riscritto come [ beta(a1-b1)+
  //gamma(a-c1)+Td1 = a-o ] dove T,beta,gamma sono
  //le incognite e a1 b1 e c1 vertici del triangolo,
  //d1 e' invece la direzione del raggio

  //Componenti note:

  //componenti X

  // (a1-b1).x
  double a = vertices[0].x-vertices[1].x;
  // (a1-c1).x
  double b=vertices[0].x-vertices[2].x;
```

```
    //d1.x
    double c=r.d.x;
    // (a-o).x
    double d=vertices[0].x-r.o.x;

    //componenti Y
    double e=vertices[0].y-vertices[1].y;
    double f=vertices[0].y-vertices[2].y;
    double g=r.d.y;
    double h=vertices[0].y-r.o.y;

    //componenti Z
    double i=vertices[0].z-vertices[1].z;
    double j=vertices[0].z-vertices[2].z;
    double k=r.d.z;
    double l=vertices[0].z-r.o.z;

    //ora ho tutte le componenti del sistema , lo
    //risolvo (inizio a calcolare beta, gamma e t)
    // | a b   c  ||d|
    // | e f   g  ||height|
    // | i j   k  ||l|

    double m=f*k-g*j;
    double n=h*k-g*l;
    double p=f*l-h*j;

    double q=g*i-e*k;
    double s=e*j-f*i;

    double inv_denom=1.0/(a*m+b*q+c*s);

    double e1=d*m-b*n-c*p;
    double beta=e1*inv_denom;

    if(beta<0.0){
      return(-1.0f);
    }

    double r1=e*l-h*i;
    double e2=a*n+d*q+c*r1;
    double gamma=e2*inv_denom;

    if(gamma<0.0) {
      return(-1.0f);
    }

    if(beta+gamma>1.0){
      return(-1.0f);
```

```
  }

  double e3=a*p-b*r1+d*s;
  float t=(float) (e3*inv_denom);

  if(t< Utilities.EPSILON)
    return -1.0f;
  else
    return t;
}

//funzione che restituisce la normale del triangolo
Point3D normal(){
  return n;
}

Point3D calculateCenter() {
  Point3D center = new Point3D();

  for (Point3D v : vertices) {
    center.sum(v);
  }

  return center.multiplyScalar(1/ (double) 3);
}

void translate(Point3D direction) {
  for (Point3D vertex : vertices) {
    vertex.sum(direction);
  }
}

void rotate(Point3D axis, double phi, boolean onlyTriangle) {
  // Rotazione in R3 secondo teorema di Eulero per rotazione di corpi rigidi
  Point3D center;

  if (onlyTriangle) {
    center = calculateCenter();
  } else {
    center = new Point3D();
  }

  for (int i = 0; i < vertices.length; i++) {
    vertices[i] = vertices[i].subtract(center);

    // Uso della formula
    // R(axis, phi)(v) = cos(phi)*v + (1 - cos(phi))*vParallelo + sen(phi)*(axis
        X v)
    // Con X prodotto vettoriale; con vParallelo = <axis, v>*axis
```

```
      double cosPhi = Math.cos(phi);
      Point3D p1 = vertices[i].multiplyScalar(cosPhi);
      Point3D p2 = axis.multiplyScalar(vertices[i].dotProduct(axis))
          .multiplyScalar(1 - cosPhi);
      Point3D p3 = vertices[i].crossProduct(axis)
          .multiplyScalar(Math.sin(phi));

      vertices[i] = p1.add(p2).add(p3).add(center);
    }
  }

  @Override
  public String toString() {
    return "Triangolo, punti: (" + vertices[0] + ") (" + vertices[1] + ") (" +
        vertices[2] + ")";
  }
}
```

Classe Obj:

```
package primitive;

import renderer.Utilities;

import java.util.ArrayList;

//La classe primitive.Obj funge da "padre" delle varie geometrie
//presenti nel programma (primitive.Triangle e primitive.Sphere); grazie a
//questa i vari algoritmi agiscono allo stesso modo su
//ogni geometria.
//la classe contiene i seguenti metodi:
//calcolo della normale all'oggetto
//calcolo per ottenere un punto random sulla superficie
//dell'oggetto
//calcolo dell'area dell'oggetto
//verifica dell'intersezione tra un oggetto e un raggio
//calcolo dei valori min e max del bounding box
//dell'oggetto: i due valori min e max delimitano il
//volume entro cui e' contenuto l'oggetto in questione

/* Il problema principale Ăš l'erronea gestione degli oggetti
 *  da questa classe. Programmata inizialmente secondo
 *  un'ottica di programmazione strutturata, sarebbe l'ideale
 *  creare una superclasse per gli oggetti, con i metodi principali
 *  da richiamare, per poi estendere le classi alle sottoclassi
 *  dell'oggetto in questione. Altrimenti la gestione dalle
```

```
 *  varie classi richiederÄ  sempre un controllo preventivo
 *  dell'oggetto.
 */

public class Obj {
  public Sphere s = null;
   public Triangle t = null;
   Hourglass h = null;
   //TODO cambiare organizzazione delle variabili

   //bounding box dell'oggetto:
   public Point3D min=new Point3D();
   public Point3D max=new Point3D();

   //area dell'oggetto
   public double areaObj;

   //potenza emessa dall'oggetto: viene utilizzata
   //all'interno dell'algoritmo che calcola la radiosita'
   public Point3D P=new Point3D();

   //matId e' l'indice che fara' riferimento alla lista
    //dei materiali, quindi determina il materiale dell'primitive.Obj
   public int matId;

   //costruttore per primitive.Obj sfera
   public Obj(Object obj) {
      if (obj instanceof Sphere) {
         s = (Sphere) obj;
         matId = s.matId;

         calculateBoundingBox();

         //calcolo il bounding box dell'oggetto
         Point3D r= new Point3D(s.rad);
         min=(s.p).subtract(r);
         max=(s.p).add(r);
      } else if (obj instanceof Triangle) {
         t = (Triangle) obj;
         matId = t.matId;
      } else if (obj instanceof Hourglass) {
         h = (Hourglass) obj;
         matId = h.matId;

         //calcolo il bounding box dell'oggetto
         //primitive.Point3D r= new primitive.Point3D(h.rad);
         //min=(h.p).subtract(r);
         //max=(h.p).add(r);
      }
```

```
      //calcolo l'area dell'oggetto
      areaObj = area();
   }
   void setMaterial(int m){
      matId=m;
   }

   //richiamo rispettivamente i metodi di triangle o
   //sphere
   public Point3D normal(Point3D iP){
      if (s != null) {
         return s.normal(iP);
      } else if (t != null) {
         return t.normal();
      } else if (h != null) {
         return h.normal(iP);
      }

      return new Point3D();
   }

   //metodo che restituisce un punto random sulla
   //superficie dell'oggetto
   public Point3D randomPoint(float rnd1, float rnd2, float rnd3){
      if(s!=null){
         double cos1= Math.cos(rnd1);
         double cos2= Math.cos(rnd2);
         double sin1= Math.sin(rnd1);
         double sin2= Math.sin(rnd2);
         Point3D r=new Point3D(cos1*cos2,cos1*sin2,sin1);
         return (s.p).add(r);
      }
      else if(t!=null){
         float d=rnd1+rnd2+rnd3;
         rnd1 /= d;
         rnd2 /= d;
         rnd3 /= d;
         Point3D ret1=t.vertices[0].multiplyScalar(rnd1);
         Point3D ret2=t.vertices[1].multiplyScalar(rnd2);
         Point3D ret3=t.vertices[2].multiplyScalar(rnd3);
         Point3D ret=ret1.add(ret2);
         ret=ret.add(ret3);

         return ret;
      } else if (h != null) {
         double phi = Math.acos(Math.random());
         double theta = Math.acos(Math.random());
         // Distribuzione di probabilitÃ  uniforme in [0, 1]
```

```
      double senPhi = Math.sin(phi);
      double cosPhi = Math.cos(phi);
      double senTheta = Math.sin(theta);
      double cosTheta = Math.cos(theta);

      double rad10 = h.rad/(double) 10;
      double multiplier = (rad10 + cosPhi*(h.rad - rad10));

      Point3D r = new Point3D(senPhi*cosTheta*multiplier,
            senPhi*senTheta*multiplier, h.rad*cosPhi);

      return h.p.add(r);
   }

   return new Point3D();
}

//metodo che calcola l'area dell'oggetto
public double area() {
   if (s != null) {
      //area sfera: 4*pigreco*r
      return 4* Utilities.MATH_PI *s.rad;
   }  else if (t != null) {
      //prendo i vertici del triangolo
      Point3D[] v= t.vertices;
      //calcolo dell'area del triangolo
      Point3D l1=v[1].subtract(v[0]);
      Point3D l2=(v[2].subtract(v[0]));
      //utilizziamo l'altezza che e' data da il
      //primo lato per il seno dell'angolo compreso
      //tra i due vettori
      float l1_norma=(float) Math.sqrt(l1.x*l1.x+l1.
            y*l1.y+l1.z*l1.z);
      float l2_norma=(float) Math.sqrt(l2.x*l2.x+l2.
            y*l2.y+l2.z*l2.z);
      double cosl1_l2=l1.dotProduct(l2)/(l1_norma*l2_norma);
      float sinl1_l2=(float) Math.sqrt(1-cosl1_l2*
            cosl1_l2);

      return l1_norma*sinl1_l2*l2_norma;
   } else if (h != null) {
      // Estimatore di Montecarlo con N campioni in [0, 1] a campionamento libero
      int samples = 200;
      double sum = 0;
      // x: cos(phi); y: sen(phi)*cos(theta)*(r/10 - (1-r/10)*cos(phi));
            z = sen(phi)*sen(theta)*(1-r/10)*cos(phi))

      for (int i = 0; i < samples; i++) {
```

```
            double phi = Math.acos(Math.random());
            double theta = Math.acos(Math.random());
            // Distribuzione di probabilitĂ  uniforme in [0, 1]

            double senPhi = Math.sin(phi);
            double cosPhi = Math.cos(phi);
            double senTheta = Math.sin(theta);
            double cosTheta = Math.cos(theta);

            double rad10 = h.rad/(double) 10;
            double multiplier = (rad10 + cosPhi*(h.rad - rad10));

            Point3D derPhiFormula =
               new Point3D(cosPhi*cosTheta*multiplier,
                  cosPhi*senTheta*multiplier, -h.rad*senPhi);
            Point3D derThetaFormula =
               new Point3D(-senPhi*senTheta*multiplier,
                  senPhi*cosTheta*multiplier, 0);

            double norm = derPhiFormula
               .crossProduct(derThetaFormula).normalize();

            sum += norm;
         }

         sum /= 1 / (double) (4 * Utilities.MATH_PI * Utilities.MATH_PI);

         return sum / (double) samples;
      }

      return 0.0f;
   }

   //controlla se il raggio passato come parametro
   //interseca l'oggetto
   public double intersect(Ray raggio) {
      //richiamo rispettivamente i metodi di triangle o
      //sphere
      if (s != null) {
         return s.intersect(raggio);
      } else if ( t != null) {
         return t.intersect(raggio);
      } else if (h != null) {
         return h.intersect(raggio);
      }

      return 0.0f;
   }
```

```
//aggiorna (considerando il paramero oldMax) il valore
//massimo del bound che circonda gli oggetti contenuti
//nell'array objects
public static Point3D getBoundMax(ArrayList<Obj> objects, Point3D oldMax){
   //per ogni oggetto
   for (Obj object : objects) {
      //se l'oggetto e' un triangolo
      if (object.t != null) {

         Triangle t = object.t;

         //per ogni vertice
         for (int j = 0; j < 3; j++) {
            //si controlla se il vertice e' il massimo
            //tra i vertici controllati finora (se lo
            //e' lo imposto come massimo)
            if (t.vertices[j].x > oldMax.x)
               oldMax.x = t.vertices[j].x;
            if (t.vertices[j].y > oldMax.y)
               oldMax.y = t.vertices[j].y;
            if (t.vertices[j].z > oldMax.z)
               oldMax.z = t.vertices[j].z;
         }
      }

      //se l'oggetto e' una sfera
      if (object.s != null) {
         //carico il centro della sfera
         Point3D sp = object.s.p;
         //carico il raggio della sfera
         float rad = object.s.rad;

         //viene preso in esame il bounding box
         //della sfera quello cioe' il cubo di
         //lato 2*rad centrato sulla sfera
         if (sp.x + rad > oldMax.x) oldMax.x = sp.x + rad;
         if (sp.y + rad > oldMax.y) oldMax.y = sp.y + rad;
         if (sp.z + rad > oldMax.z) oldMax.z = sp.z + rad;
      }

      if (object.h != null) {
         //carico il centro della clessidra
         Point3D sp = object.h.p;
         //carico il raggio della clessidra
         float rad = object.h.rad;

         /* Viene preso in esame il bounding box di due sfere di raggio rad/2
          *  allineate con l'asse della clessidra.
          * Il tutto facendo attenzione alla sua inclinazione
```

```
            * rispetto all'asse y (verticale)
            */
           if (sp.x + rad > oldMax.x) oldMax.x = sp.x + rad;
           if (sp.y + rad > oldMax.y) oldMax.y = sp.y + rad;
           if (sp.z + rad > oldMax.z) oldMax.z = sp.z + rad;
        }
     }

     return oldMax;
  }

  //aggiorna (considerando il paramero oldMin) il valore
  //minimo del bound che circonda gli oggetti contenuti
  //nell'array objects
  public static Point3D getBoundMin(ArrayList<Obj> objects, Point3D oldMin){
     //per ogni oggetto
     for (Obj object : objects) {

        //se l'oggetto e' un triangolo
        if (object.t != null) {

           Triangle t = object.t;

           //per ogni vertice
           for (int j = 0; j < 3; j++) {
              //si controlla se il vertice e' il minimo
              //tra i vertici controllati finora (se lo
              //e' lo imposto come minimo)
              if (t.vertices[j].x < oldMin.x)
                 oldMin.x = t.vertices[j].x;
              if (t.vertices[j].y < oldMin.y)
                 oldMin.y = t.vertices[j].y;
              if (t.vertices[j].z < oldMin.z)
                 oldMin.z = t.vertices[j].z;
           }
        }
        //se l'oggetto e' una sfera
        if (object.s != null) {
           //carico il centro della sfera
           Point3D sp = object.s.p;
           //carico il raggio della sfera
           float rad = object.s.rad;

           //viene preso in esame il bounding box
           //della sfera quello cioe' il cubo di
           //lato 2*rad centrato sulla sfera
           if (sp.x - rad < oldMin.x)
              oldMin.x = sp.x - rad;
           if (sp.y - rad < oldMin.y)
```

```
            oldMin.y = sp.y - rad;
         if (sp.z - rad < oldMin.z)
            oldMin.z = sp.z - rad;
      }

      if (object.h != null) {
         //carico il centro della sfera
         Point3D sp = object.h.p;
         //carico il raggio della sfera
         float rad = object.h.rad;

         //viene preso in esame il bounding box
         //della sfera quello cioe' il cubo di
         //lato 2*rad centrato sulla sfera
         if (sp.x - rad < oldMin.x)
            oldMin.x = sp.x - rad;
         if (sp.y - rad < oldMin.y)
            oldMin.y = sp.y - rad;
         if (sp.z - rad < oldMin.z)
            oldMin.z = sp.z - rad;
      }
   }

   return oldMin;
}

public void rotateTriangleOnly(Point3D axis, double phi) {
   if (t != null) {
      t.rotate(axis, phi, true);

      calculateBoundingBox();
   }
}

void rotateTriangleInSpace(Point3D axis, double phi) {
   if (t != null) {
      t.rotate(axis, phi, false);

      calculateBoundingBox();
   }
}

private void calculateBoundingBox() {
   if (t != null) {
      //calcolo il bounding box dell'oggetto
      max.x=Math.max(t.vertices[0].x, Math.max(t.vertices[1].x,
          t.vertices[2].x));
      max.y=Math.max(t.vertices[0].y, Math.max(t.vertices[1].y,
          t.vertices[2].y));
```

```
            max.z=Math.max(t.vertices[0].z, Math.max(t.vertices[1].z,
                t.vertices[2].z));

            min.x=Math.min(t.vertices[0].x, Math.min(t.vertices[1].x,
                t.vertices[2].x));
            min.y=Math.min(t.vertices[0].y, Math.min(t.vertices[1].y,
                t.vertices[2].y));
            min.z=Math.min(t.vertices[0].z, Math.min(t.vertices[1].z,
                t.vertices[2].z));
        }
    }

    public void setNewPosition(Point3D point) {
        if (s != null) {
            s.p.sum(point);
        } else if (t != null) {
            t.translate(point);

            calculateBoundingBox();
        }
    }

    @Override
    public String toString() {
        return (s != null ? s.toString() : t.toString());
    }

    public Point3D getPosition() {
        if (t != null) {
            return t.calculateCenter();
        } else if (s != null) {
            return s.p;
        }

        return null;
    }
}
```

Classe Material:

```
package primitive;

//classe contenente la definizione del materiale dell'
//oggetto.
//All'interno di questa classe sono definiti anche i
//metodi per il calcolo del coefficiente di Fresnel
//(che indica la quantita' di energia luminosa che viene
```

```
//riflessa o rifratta quando incontra il materiale) e per
//il calcolo della BRDF (funzione che indica la la frazione
//della radianza differenziale incidente da una direzione
//psi che viene riflessa in una direzione theta)

import renderer.Utilities;

public class Material {
   // Diffuse color
   public Point3D diffusionColor = new Point3D();

   // Reflection color
   public Point3D reflectionColor = new Point3D();

   // Refraction color
   public Point3D refractionColor = new Point3D();

   //potenza luce emessa dall'oggetto
   public Point3D emittedLight;

   // IOR (eta): indice di rifrazione rispetto alle
   //lunghezza d'onda RGB
   public Point3D refractionIndexRGB;

   //coefficiente di assorbimento per materiali conduttori
   //(anche esso varia in base alla lunghezza d'onda RGB)
   public Point3D absorptionCoefficient = new Point3D();

   //Cook-Torrance model

   //slope: coefficient di rugosita' della superficie:
   //scarto quadratico medio della pendenza delle
   //microsfaccettature. Se meshes e' piccolo allora
   //l'inclinazione delle microsfaccettature varia poco
   //rispetto alla normale della superficie e quindi la
   //riflessione e' molto a fuoco sulla direzione
   //speculare. Se meshes e' grande l'inclinazione e' elevata
   //e la superficie e' ruvida
   public static float slope=0;

   //precisione della riflessione/rifrazione (il parametro
   //deve essere minore di 1)
   public float refImperfection=0;

   //boolean che specifica se il materiale in questione
   //e' permeabile o meno: serve per capire se e'
   //necessario calcolare la BRDF o la BSSRDF
   public boolean translucent=false;
```

```
   // Nome per la scelta nella creazione dell'oggetto
   public String name;

   //materiale di default: diffusivo bianco senza
   //riflessione
   public Material(String name) {
      diffusionColor = new Point3D(1.0f);
      reflectionColor = new Point3D(0.0f);
      refractionColor = new Point3D(0.0f);
      refractionIndexRGB = new Point3D(0.0f);
      emittedLight = new Point3D(0.0f);

   }

   //costruttore materiale 1: materiali diffusivi e riflessivi
   public Material(Point3D diffusionColor, Point3D reflectionColor, String name) {
      this.diffusionColor = diffusionColor;
      this.reflectionColor = reflectionColor;
      refractionColor =new Point3D(0.0f);
      refractionIndexRGB =new Point3D(0.0f);
      emittedLight =new Point3D(0.0f);
      refImperfection=0;

      this.name = name;
   }

   //costruttore materiali lucidi (tramite il modello di Cook-Torrance)
   public Material(Point3D diffusionColor, Point3D refractionIndexRGB,
                        float slope_, String name) {
      this.diffusionColor = diffusionColor;
      slope = slope_;
      this.refractionIndexRGB =refractionIndexRGB;
      emittedLight =new Point3D(0.0f);

      this.name = name;
   }

   //costruttore materiale generico
   public Material(Point3D diffusionColor, Point3D reflectionColor,
                Point3D refractionColor, Point3D refractionIndexRGB,
                Point3D absorptionCoefficient, float slope_,
                float refImperfection_, boolean translucent_, String name) {

      emittedLight =new Point3D(0.0f);
      //normalizzazione: Kd+Kr deve essere <1
      double Ftot= diffusionColor.max()+reflectionColor.max();
      if( Ftot>1){
         this.diffusionColor =diffusionColor.divideScalar(Ftot);
         this.reflectionColor =reflectionColor.divideScalar(Ftot);
```

```
   }
   else{
      this.reflectionColor =reflectionColor;
      this.diffusionColor =diffusionColor;
   }
   this.refractionColor =refractionColor;
   this.refractionIndexRGB =refractionIndexRGB;
   this.absorptionCoefficient =absorptionCoefficient;
   slope=slope_;
   refImperfection=refImperfection_;
   translucent=translucent_;

   this.name = name;
}

//materiali che emettono luce: Le corrisponde al colore della luce emessa
public Material(Point3D emittedLight, String name) {
   diffusionColor =new Point3D(0.0f);
   reflectionColor =new Point3D(0.0f);
   refractionColor =new Point3D(0.0f);
   refractionIndexRGB =new Point3D(0.0f);
   this.emittedLight = emittedLight
         .multiplyScalar(Utilities.MATH_1_DIV_PI);

   this.name = name;
}

//metodo che calcola il coefficiente di Fresnel in base
//al coseno dell'angolo di incidenza del raggio visuale
//con la normale
public Point3D getFresnelCoefficient(double cosI){
   Point3D etat= refractionIndexRGB;
    //viene calcolato solamente per materiali con
   //indice di rifrazione maggiore di 0
   if(refractionIndexRGB.max()>0) {

      //calcolo coefficiente di Fresnel:
      //F=(|Rparal|^2+|Rperp|^2)/2
      //verifico se il materiale e' un conduttore o
      //un dielettrico tramite il parametro k
      if(absorptionCoefficient.max()<=0){
         //formula per materiali dielettrici

         //indice di rifrazione del vuoto
         Point3D etai= new Point3D(1.0f);

         //si verifica che il coseno tra la normale
         //e il raggio entrante nella superficie sia
         //maggiore di 0
```

```
//se e' minore di 0 si considera il raggio
//come uscente dalla superficie e vengono
//quindi invertiti gli indici di rifrazione
if(cosI<0){
   etai=etat;
   etat=new Point3D(1.0f);
   cosI=-cosI;
}
//calcolo del coefficienti di Fresnel:

Point3D et = etai.divideComponents(etat);
Point3D sinT2= (et.multiplyComponents(et))
      .multiplyScalar(1-cosI*cosI);
//se il seno e' maggiore di 1 allora la
//radice sara' negativa di conseguenza
// si effettuera' una riflessione totale
//ovvero il coefficiente di Fresnel deve
//essere posto uguale ad 1 (solo in questo
//caso infatti tutta la luce viene
//completamente riflessa) poiche' qui si
//considerano indici di rifrazioni
//differenti in base alla lunghezza d'onda
//se la riflessione totale non avviene per
//ogni lunghezza d'onda il metodo continua
//restringendo il seno in tutte le
//lunghezze d'onda tra 0 e 1. una volta
//finito il metodo si controllano le
//lunghezze d'onda per cui c'e' stata
//riflessione totale e si pone il
//coefficiente di Fresnel per quelle
//lunghezze d'onda uguale ad 1

if((sinT2.x>1)&&(sinT2.y>1)&&(sinT2.z>1)){
   return new Point3D(1.0f) ;
}

// restrizione in [0,1]
Point3D sint2_= Point3D.clamp3(sinT2);

//calcolo del coseno
Point3D one=new Point3D(1.0f);
Point3D cosT= Point3D.getSquareCompPoint(one.subtract(sint2_));

//formula per materiali dielettrici

//Rparal=(eta2cos1-eta1cos2)/(eta2cos1+
//+eta1cos2)
Point3D etatCosI = etat.multiplyScalar(cosI);
Point3D Rparal = (etatCosI.subtract(etai.multiplyComponents(cosT)))
```

```
        .divideComponents(etatCosI.add(etai.multiplyComponents(cosT)));

    //Rperp=(eta1cos1-eta2cos2)/(eta1cos1+
    //+eta2cos2)
    Point3D etaiCosI=etai.multiplyScalar(cosI);
    Point3D Rperp=(etaiCosI.subtract(etat.multiplyComponents(cosT)))
        .divideComponents(etaiCosI.add(etat.multiplyComponents(cosT)));

    //F=(|Rparal|^2+|Rperp|^2)/2
    Point3D result= (Rparal.multiplyComponents(Rparal).
                add(Rperp.multiplyComponents(Rperp))).
                multiplyScalar(0.5f);

    if(result.average() < Utilities.EPSILON)
        result=new Point3D(0.0f);

    //controllo della riflessione totale su
    //ogni componente RGB
    if(sinT2.x>1) {
        result.x=1;
    }

    if(sinT2.y>1) {
        result.y=1;
    }

    if(sinT2.z>1) {
        result.z=1;
    }
    return result;
} else {
    //formula per materiali conduttori
    etat= refractionIndexRGB;

    //Rparal=((etat^2+k^2)cos1^2-2*etat*cos1+
    //+1)/((etat^2+k^2)cos1^2+2*etat*cos1+1)
    Point3D tmp= (etat.multiplyComponents(etat))
          .add(absorptionCoefficient.multiplyComponents(absorptionCoefficient));
    Point3D tmp2= tmp.multiplyScalar(cosI).multiplyScalar(cosI);
    Point3D ior2cos_i= etat.multiplyScalar(2.0f).multiplyScalar(cosI);
    Point3D Rparal_2= (tmp2.subtract(ior2cos_i).add(new Point3D(1.0f)))
          .divideComponents(tmp2.add(ior2cos_i).add(new Point3D(1.0f)));

    //Rperp=((etat^2+k^2)-2*etat*cos1+
    //+cos1^2)/((etat^2+k^2)+2*etat*cos1+cos1^2)
    Point3D cos2i=new Point3D(cosI*cosI);
    Point3D Rperp_2=(tmp.subtract(ior2cos_i).add(cos2i)).
                divideComponents(tmp.add(ior2cos_i).add(cos2i));
    return (Rparal_2.add(Rperp_2)).multiplyScalar(0.5f);
```

```
        }
      } else {
        //negli altri casi poniamo il coefficiente di
        //Fresnel uguale a 1
        return new Point3D(1.0f);
      }
    }

    //BRDF per materiali riflettenti
    public Point3D S_BRDF(Point3D fresnel){
      return fresnel.multiplyComponents(reflectionColor);
    }

    //BRDF per materiali trasparenti
    public Point3D T_BRDF(Point3D fresnel){
      Point3D one=new Point3D(1.0f);
      return refractionColor .multiplyComponents(one.subtract(fresnel));
    }

    //BRDF di Cook-Torrance=kd+(F*D*G)/(pi*<psi,n>*<teta,n>)
    //psi e' il raggio in ingresso, theta e' il raggio in uscita, n e' la normale
        dell'oggetto
    public Point3D C_T_BRDF(Ray psi, Ray theta, Point3D n) {
      //parte diffusiva
      Point3D fr= diffusionColor.multiplyScalar(Utilities.MATH_1_DIV_PI);

      //la parte riflettente viene considerata solo se
      //lo slope e' stato inizializzato (ovvero e'
      //diverso da 0)
      if(slope!=0) {
        //dati:

        //halfway vector
        Point3D H = (psi.d.add(theta.d)).getNormalizedPoint();

        //<psi,H>=<theta,H>
        double c = psi.d.dotProduct(H);
        //<n,H>
        double cNH = n.dotProduct(H);
        //<psi,n>
        double cPsiN = psi.d.dotProduct(n);
        //<teta,n>
        double cThetaN = theta.d.dotProduct(n);


        //calcolo coefficiente di Fresnel
        // !!! si potrebbe anche fare con il metodo
        //getFresn(cos_i) !!!
```

```
Point3D F;

Point3D ior2 = new Point3D(
      refractionIndexRGB.x * refractionIndexRGB.x,
      refractionIndexRGB.y * refractionIndexRGB.y,
      refractionIndexRGB.z * refractionIndexRGB.z);

Point3D g = ior2.add(new Point3D(c * c - 1));
g.abs();

//g+c
Point3D c3 = new Point3D(c, c, c);
Point3D gc = g.add(c3);
//g-c
Point3D g_c = g.subtract(c3);

// (g-c)^2/(g+c)^2
Point3D a = ((g_c).multiplyComponents(g_c))
      .divideComponents(gc.multiplyComponents(gc));

//(c*(g+c)-1)/(c*(g+c)+1)
Point3D one = new Point3D(1.0f);
Point3D b = ((gc.multiplyScalar(c)).subtract(
      one)).divideComponents((gc.multiplyScalar(c)).
      add(one));

//F= 1/2 * ((g-c)^2/(g+c)^2) * (1+ ((c*(g+c)-
//-1)/(c*(g+c)+1))^2)
F = a.multiplyComponents(one.add(b.multiplyComponents(b))).
      multiplyScalar(0.5f);


//Distribuzione di Beckmann
double sNH = Math.sqrt(1 - Math.pow(cNH, 2));

double tan = sNH / (cNH * slope);
double e = Math.exp(Math.pow(tan, 2));
double _D = e * (slope * slope * Math.pow(cNH, 4));
double D = 1 / _D;
float Df = (float) D;
//fattore geometrico

double G = 1;
if (2 * cPsiN * cNH / c < G) {
   G = 2 * cPsiN * cNH / c;
}
if (2 * cThetaN * cNH / c < G) {
   G = 2 * cThetaN * cNH / c;
```

```
        }
        float Gf = (float) G;

        //modello di Cook-Torrance
        fr = fr.add(F.multiplyScalar(Df * Gf *
             Utilities.MATH_1_DIV_PI * 1 / (cPsiN * cThetaN)));
      }

      return fr;
   }

   //metodo empirico (non fotorealistico) per avere
   //un valore per la BSSRDF
   //Fpsi e Ftheta sono i coefficiente di Fresnel passati
   //come parametro, calcolati nel main considerando
   //come psi il raggio in ingresso e theta quello in
   //uscita
   Point3D BSSRDF(Point3D Fpsi, Point3D Ftheta) {
      Point3D result;
      Point3D one=new Point3D(1.0f);

      float zv=0.005f;
      float dv;//=0.0125f;
      float zr=0.0025f;
      float dr;//=0.01030f;
      dv=(float) (Math.random() * (150-90)+90);
      dv=dv/10000f;
      float l2=dv*dv-zv*zv;
      dr=(float) Math.sqrt(l2+zr*zr);
      Point3D pi4=new Point3D(4* Utilities.MATH_PI);
      //i valori si sigmas e sigmaa sono specifici per
      //la giada
      Point3D sigmas=new Point3D(0.657f,0.786f,0.9f);
      Point3D sigmaa=new Point3D(0.2679f,0.3244f,0.1744f);
      Point3D sigmat=sigmaa.add(sigmas);
      Point3D sigmatr=(sigmaa.multiplyComponents(sigmat)).
           multiplyScalar(3.0f);
      sigmatr= Point3D.getSquareCompPoint(sigmatr);
      Point3D alpha=sigmas.divideComponents(sigmat);

      Point3D expdr=sigmatr.multiplyScalar(dr*-1.0f);
      float dr3=dr*dr*dr;
      Point3D edivdr=(Point3D.exponent(expdr)).
           divideScalar(dr3);

      Point3D expdv=sigmatr.multiplyScalar(dv*-1.0f);
      float dv3=dv*dv*dv;
      Point3D edivdv=(Point3D.exponent(expdv)).
           divideScalar(dv3);
```

```
      Point3D rPart=(((sigmatr.multiplyScalar(dr)).
            add(one)).multiplyComponents(edivdr)).
            multiplyScalar(zr);
      Point3D vPart=(((sigmatr.multiplyScalar(dv)).
            add(one)).multiplyComponents(edivdv)).
            multiplyScalar(zv);

      Point3D Rd=(alpha.divideComponents(pi4)).multiplyComponents(rPart.
            add(vPart));

      result=(Rd.multiplyComponents(Fpsi).multiplyComponents(Ftheta)).
            divideScalar(Utilities.MATH_PI);

      return result;
   }

   @Override
   public String toString() {
      return name;
   }
}
```

Classe StandardMaterial:

```
package primitive;

public interface StandardMaterial {
  /* Interfaccia per la selezione di materiali
   * Cruciale per la scelta di un materiale senza doverlo ricreare e nella
       selezione del materiale dal modellatore
   */
  Material MATERIAL_LIGHT_WHITE = new Material(new Point3D(5.0f),
      "MATERIAL_LIGHT_WHITE");

  Material MATERIAL_DIFFUSIVE_RED = new Material(
      new Point3D(0.1f,0.0f,0.0f),
      new Point3D(), "MATERIAL_DIFFUSIVE_RED");
  Material MATERIAL_DIFFUSIVE_GREEN = new Material(
      new Point3D(0.05f,0.3f,0.0f),
      new Point3D(), "MATERIAL_DIFFUSIVE_GREEN");
  Material MATERIAL_DIFFUSIVE_BLUE = new Material(
      new Point3D(0.45f,0.45f,1.0f),
      new Point3D(), "MATERIAL_DIFFUSIVE_BLUE");
  Material MATERIAL_DIFFUSIVE_GRAY = new Material(
      new Point3D(0.7f),
      new Point3D(), "MATERIAL_DIFFUSIVE_GRAY");
  Material MATERIAL_DIFFUSIVE_BLACK = new Material(
```

```
    new Point3D(),
    new Point3D(), "MATERIAL_DIFFUSIVE_BLACK");
Material MATERIAL_DIFFUSIVE_PINK = new Material(
    new Point3D(1.0f,0.4f,0.4f),
    new Point3D(), "MATERIAL_DIFFUSIVE_PINK");
Material MATERIAL_DIFFUSIVE_DEEP_GRAY = new Material(
    new Point3D(0.2f,0.15f,0.15f),
    new Point3D(), "MATERIAL_DIFFUSIVE_DEEP_GRAY");

Material MATERIAL_REFLECTIVE_GLASS = new Material(
    new Point3D(),
    new Point3D(1.0f), "MATERIAL_REFLECTIVE_GLASS");
Material MATERIAL_REFLECTIVE_PERFECT_GLASS = new Material(
    new Point3D(),
    new Point3D(),
    new Point3D(1.0f),
    new Point3D(1.55f),
    new Point3D(),
    0.0f,0.0f,false, "MATERIAL_REFLECTIVE_PERFECT_GLASS");
Material MATERIAL_COOK_TORRANCE_VIOLET = new Material(
    new Point3D(0.6f,0.1f,0.2f),
    new Point3D(5.3f,1.485f,1.485f),
    0.9f, "MATERIAL_COOK_TORRANCE_VIOLET");
Material MATERIAL_STEEL = new Material(
    new Point3D(),
    new Point3D(1.0f),
    new Point3D(),
    new Point3D(2.485f),
    new Point3D(3.433f),
    0.0f ,0.0f,false, "MATERIAL_STEEL");
Material MATERIAL_IMPERFECT_STEEL = new Material(
    new Point3D(),
    new Point3D(1.0f),
    new Point3D(),
    new Point3D(1.485f,2.885f,2.885f),
    new Point3D(3.433f,1.433f,1.433f),0.0f,0.01f, false,
       "MATERIAL_IMPERFECT_STEEL");
Material MATERIAL_DEEP_RED = new Material(
    new Point3D(0.5f,0.12f,0.2f),
    new Point3D(), "MATERIAL_DEEP_RED");
Material MATERIAL_TRANSLUCENT_JADE = new Material(
    new Point3D(0.31f,0.65f,0.246f),
    new Point3D(),
    new Point3D(),
    new Point3D(1.3f,1.3f,1.3f),
    new Point3D(),
    0,0,true, "MATERIAL_TRANSLUCENT_JADE");
Material MATERIAL_DIFFUSIVE_JADE = new Material(
    new Point3D(0.31f,0.65f,0.246f),
```

```
      new Point3D(),
      new Point3D(),
      new Point3D(1.3f,1.3f,1.3f),
      new Point3D(),
      0,0,false, "MATERIAL_DIFFUSIVE_JADE");
}
```

Classe Mesh:

```
package primitive;

import renderer.Utilities;
import ui.RenderAction;

import java.util.ArrayList;

//classe per definire una mesh: una raccolta di vertici,
//lati e facce che definiscono la forma, quindi la
//modellazione solida di un oggetto poliedrico.
//Le facce di solito sono costituite da triangoli
//(maglia triangolare), quadrilateri o altri poligoni
//convessi semplici, poiche' cio' semplifica il rendering,
//ma puo' anche essere composto da poligoni concavi piu'
//generali e sfere.
//Nel nostro caso considereremo solo sfere o triangoli.
//La classe contiene anche un metodo per assegnare il
//materiale di una mesh e un metodo per spezzare in
//due parti uguali ogni triangolo della mesh (e' ovvio che
//il metodo non funziona per mesh di sfere)

/* Come in altre classi, qui si accede al materiale
 *  dell'oggetto tramite un indice che si interfaccia
 *  alla classe statica RenderAction. Sarebbe necessario
 *  associare a ogni oggetto creato il suo materiale,
 *  senza doversi ricollegare a una classe statica.
 * Il secondo rilevante problema Ăš la necessitĂ  di
 *  aggiungere una mesh dinamica per gli oggetti,
 *  cosicchĂš sia piĂč efficiente la traslazione e la
 *  rotazione per oggetti generici.
 */

public class Mesh {
  Utilities utilities;

  //array di puntatori ad oggetti
  public ArrayList<Obj> objects;
```

```
int[] matIdRoom = {
    //vettore per gli indici dei materiali delle pareti della stanza
    3, //sinistra
    4, //inferiore
    13, //posteriore
    3, //destra
    4, //superiore
    4  //frontale
};

private Mesh() {
  utilities = new Utilities();
}

public Mesh(ArrayList<Sphere> spheres) {
  this();
  loadSphere(spheres);
}

public Mesh(Point3D max, Point3D min) {
  this();
  createScene(max, min);
}

//si crea la stanza con grandezza dipendente da max e min
//per stanza si intende un ambiente parallelepipedico
//avente pareti (composte da due triangoli per ciascuna
//parete) di un materiale scelto tra quelli presenti
//nell'array dei materiali, una luce nel nostro caso
//collocata sul soffitto
void createScene(Point3D max, Point3D min){
  //uso le variabili maxx e minn che hanno i valori di
  //max e min per non aggiornarli dopo l'utilizzo del
  //metodo
  Point3D maxx=new Point3D();
  maxx.copy(max);
  Point3D minn=new Point3D();
  minn.copy(min);

  double maxhroom = maxx.y + RenderAction.hroom;
  maxx.y = max.y;

  //definisco un array per gli 8 vertici della stanza
  //a max y aggiungo sempre hroom
  //vertici: sinistra=min.x, destra=max.x; basso=min.y,
  //alto=max.y; dietro=min.z, davanti=max.z
  Point3D[] v ={
        //sinistra, in basso, dietro
  minn,
```

```
//sinistra, in alto, dietro
new Point3D(minn.x,maxhroom,minn.z),
//sinistra, in basso, davanti
new Point3D(minn.x,minn.y,maxx.z),
//destra, in basso, dietro
new Point3D(maxx.x,minn.y,minn.z),
//destra, in alto, davanti
new Point3D(maxx.x,maxhroom,maxx.z),
//sinistra, in alto, davanti
new Point3D(minn.x,maxhroom,maxx.z),
//destra, in alto, dietro
new Point3D(maxx.x,maxhroom,minn.z),
//destra, in basso, davanti
new Point3D(maxx.x,minn.y,maxx.z),
};

//Dal momento che frontWall=false, si creano 12
//triangoli per comporre la stanza: 10 per la stanza
//(a cui manca appunto la faccia frontale per
//permettere di vedere all'interno) e 2 per la luce
//parallela al soffitto (perche' frontL=false)
ArrayList<Triangle> tRoom = new ArrayList<>();
//array per gli oggetti che compongono la stanza
objects = new ArrayList<>(12);

//inseriamo la luce
//c e' la posizione centrale della stanza
Point3D c=((maxx.add(minn)).multiplyScalar(
       0.5f));
//dichiaro l'array Lv che serve per i vertici della
//luce e lo inizializzo a (0,0,0)
Point3D[] Lv=new Point3D[8];
for(int i=0; i<8; i++) {
  Lv[i]=new Point3D();
}

for(int i=0; i<8; i++) { //dilatazione della luce
  Lv[i].copy(v[i]); //copio in Lv i vertici della stanza
  //faccio poi le dovute dilatazioni nella x
  //Lv[i].x=Lv[i].x+(Lv[i].x-c.x)*scaleL.x;

  //frontL e' false quindi entra in questo if (la
  //luce non e' frontale): si fanno quindi le
  //dovute dilatazioni nella z

  /*
  if(!Main.frontL){
    //Lv[i].z=Lv[i].z+(Lv[i].z-c.z)*scaleL.z;
  }
```

```
        else{
            if( (i==4 )|| (i==5) ){
                Lv[i].y=Lv[i].y+hroom*scaleL.y;
            }
        }
      */
    }

    //anche in questo caso copio i valori in delle
    //variabili che non si aggiornano
    Point3D translateLL;
    Point3D Lv1=new Point3D();
    Lv1.copy(Lv[1]);
    Point3D Lv4=new Point3D();
    Lv4.copy(Lv[4]);
    Point3D Lv5=new Point3D();
    Lv5.copy(Lv[5]);
    Point3D Lv6=new Point3D();
    Lv6.copy(Lv[6]);

    if(!RenderAction.frontL){
      //definisco i vertici dei triagoli della stanza,
      //traslandoli sulla y del valore
      //utilities.EPS=0.01f per evitare di avere
      //problemi di aliasing dati dal linee
      //perfettamente dritte nella fase di rendering
      translateLL = new Point3D(0, -Utilities.EPSILON, 0);
      Point3D Lv1tr = (Lv1).add(translateLL);
      Point3D Lv4tr = (Lv4).add(translateLL);
      Point3D Lv5tr = (Lv5).add(translateLL);
      Point3D Lv6tr = (Lv6).add(translateLL);
      //si creano e si aggiungono i triangoli per la
      //luce
      Triangle Tr0=new Triangle(Lv4tr,Lv5tr,Lv6tr, RenderAction.matIdL);
      tRoom.add(0,Tr0);
      Triangle Tr1=new Triangle(Lv1tr,Lv6tr,Lv5tr, RenderAction.matIdL);
      tRoom.add(1,Tr1);
    }

    //dilatazione delle pareti
    for (int i = 0; i < 8; i++) {
      v[i].x=v[i].x+(v[i].x-c.x)*1.5f;
      v[i].z=v[i].z+(v[i].z-c.z)*1.5f;
    }

    //si creano e si aggiungono i triangoli per le
    //pareti
    //faccia laterale sinistra
    Triangle Tr2=new Triangle(v[0],v[1],v[2], matIdRoom[0]);
```

```
  tRoom.add(2,Tr2);
  Triangle Tr3=new Triangle(v[5],v[2],v[1], matIdRoom[0]);
  tRoom.add(3,Tr3);
  //faccia inferiore
  Triangle Tr4=new Triangle(v[0],v[2],v[3], matIdRoom[1]);
  tRoom.add(4,Tr4);
  Triangle Tr5=new Triangle(v[7],v[3],v[2], matIdRoom[1]);
  tRoom.add(5,Tr5);
  //faccia posteriore
  Triangle Tr6=new Triangle(v[0],v[3],v[1], matIdRoom[2]);
  tRoom.add(6,Tr6);
  Triangle Tr7=new Triangle(v[6],v[1],v[3], matIdRoom[2]);
  tRoom.add(7,Tr7);
  //faccia laterale destra
  Triangle Tr8=new Triangle(v[4],v[6],v[7], matIdRoom[3]);
  tRoom.add(8,Tr8);
  Triangle Tr9=new Triangle(v[7],v[6],v[3], matIdRoom[3]);
  tRoom.add(9,Tr9);
  //faccia superiore
  Triangle Tr10=new Triangle(v[4],v[5],v[6], matIdRoom[4]);
  tRoom.add(10,Tr10);
  Triangle Tr11=new Triangle(v[1],v[6],v[5], matIdRoom[4]);
  tRoom.add(11,Tr11);

  for (Triangle t : tRoom) {
    t.isBorderMeshScene = true;
    objects.add(new Obj(t));
  }
}

void loadSphere(ArrayList<Sphere> spheres) {
  //crea una mesh costituita da n sfere
  //creo un array di primitive.Obj di n elementi
    objects = new ArrayList<>();

    //per ogni elemento creo un oggetto che ha l'elemento
    //i-esimo dell'array di primitive.Sphere[] spheres e l'elemento
    //i-esimo di int[] matIdSphere (che considerera'
    //l'i-esimo materiale)
  for (Sphere sphere : spheres) {
    objects.add(new Obj(sphere));
  }

    //viene restituita una mesh delle sfere create
}

//metodo per impostare il materiale degli oggetti
//presenti nella mesh
```

```
  void setMaterial(int m) {
    for (Obj object : objects) {
      object.setMaterial(m);
    }
  }

  //funzione valida solo per primitive.Mesh di triangoli
  //il metodo suddivide i triangoli della primitive.Mesh in due,
  //duplicando quindi il numero di triangoli della primitive.Mesh
  //nel nostro caso sceneDepth=0, quindi non si usa mai
  //questo metodo
  public void splitMeshes(){
    //nuovo array dei triangoli
    ArrayList<Obj> objects2 = new ArrayList<>(objects.size()*2);
    for (Obj object : objects) {
      //valido solamente su triangoli
      if (object.t != null) {
        //carico i vertici del triangolo
        Point3D[] v = object.t.vertices;
        //creo i lati del triangolo
        Point3D l[] = {v[2].subtract(v[0]),
            v[0].subtract(v[1]), v[1].
            subtract(v[2])};
        //cerco il lato piu' lungo del triangolo
        int pos = 0;
        if (l[1].normalize() > l[pos].normalize()) pos = 1;
        if (l[2].normalize() > l[pos].normalize()) pos = 2;

        //cerco il punto a meta' del lato piu'
        //lungo del triangolo
        float val = (float) (l[pos].normalize() * 0.5);
        //normalizzo il lato piu' lungo
        l[pos].getNormalizedPoint();
        //carico i nuovi vertici del triangolo
        Point3D[] nv = {v[(pos + 1) \% 3], v[(pos + 2) \%3],
            v[pos], l[pos].multiplyScalar(val).
            add(v[pos])};
        //creo i nuovi due triangoli
        objects2.add(new Obj(new Triangle(nv[0], nv[1], nv[3], object.matId)));
        objects2.add(new Obj(new Triangle(nv[0], nv[3], nv[2], object.matId)));
      }
    }

    //aggiorno l'array degli oggetti e il numero di
    //oggetti della mesh
    objects= objects2;
  }
}
```

Classe ModelerProperties:

```
package primitive;

public interface ModelerProperties {
  /* primitive.ModelerProperties gestisce l'inizializzazione del programma e delle
     istanze.
   * Le istanze variano tra l'avviare una anteprima (per spostare o creare oggetti)
      e avviare il render finale.
   * ENABLE_MODELER avvia l'interfaccia di anteprima e il modellatore
   * PREVIEW_ONLY aggiorna il modellatore in caso di varie modifiche
   * START_RENDERING deallora il modellatore per avviare il render finale
   */
  int ENABLE_MODELER = 0;
  int START_RENDERING = 1;
  int PREVIEW_ONLY = 2;
}
```

Classe Photon:

```
package primitive;

/* Classe per la creazione di un fotone, con una posizione
 * di partenza, l'energia del fotone e la sua direzione.
 * Il fotone verrÃ  poi gestito nella PhotonBox, ovvero nella
  * suddivisione dello spazio secondo KDTree per gestirne
 *  la hitting distribution e calcolare l'illuminazione
 */
public class Photon {
   public Point3D position=new Point3D();
   public Point3D power=new Point3D();
   public Point3D direction=new Point3D();

   public Photon(Point3D ip, Point3D d, Point3D p){
      position.copy(ip);
      power.copy(p);
      direction.copy(d);
   }
}
```

Classe Ray:

```
package primitive;
```

```
// Classe che definisce un raggio attraverso i parametri
// origine, direzione e profonditĂ
public class Ray {
   // Origine del raggio
   public Point3D o;
   // Direzione del raggio
   public Point3D d;
   // Profondita' del raggio
   public float depth;

   public Ray() {
      o = new Point3D();
      d = new Point3D();
      depth = 0.0f;
   }

   public Ray(Point3D or, Point3D di) {
      o = or;
      d = di;
      depth = 0.0f;
   }

   public void setDepth(float newDepth)
   {
      depth=newDepth;
   }
}
```

Classe Sphere:

```
package primitive;

//classe che costruisce una sfera attraverso i parametri
//posizione (centro) e raggio
//La classe ha la funzione che calcola l'intersezione
//della sfera con un raggio, e la funzione che restituisce
//la normale alla sfera in un punto dato

/* Il problema principale di questa classe Ăš dato dalla
 *  natura dell'oggetto, basato su formule matematiche invece
 *  che meshes, il che rende difficile applicare mappe
 *  di tessitura (textures) o lo stesso metodo di Jacobi.
 * Sarebbe necessario strutturare l'oggetto in modo da avere
 *  una creazione di meshes dinamico, in base al grado di
 *  approssimazione.
 */
```

```
import renderer.Utilities;
import ui.RenderAction;

public class Sphere {
  // raggio
  public float rad;
  // posizione (centro)
  public Point3D p;

  int matId;

  //costruttore
  public Sphere(float nrad, Point3D np, int matId) {
    rad=nrad;
    p=np;
    this.matId = matId;
  }

  //metodo che imposta, a seconda della scelta
  //effettuata dall'utente, la posizione
  //appropriata alle prime tre sfere
  //Prende come parametro l'indice dell'array
  //spheres di cui si deve settare la posizione
  public static Point3D setSpheresPosition(int index) {
    if(RenderAction.aligned) {
      switch(index) {
        case 0:
          return new Point3D(-1.0f,0.0f,0.0f);
        case 1:
          return new Point3D(-5.0f,0.3f,0.8f);
        case 2:
          return new Point3D(3.5f,0.5f,3.3f);
        default:
          return new Point3D(0.0f);
      }
    } else {
      switch(index) {
        case 0:
          return new Point3D(-4.0f,0.0f,0.0f);
        case 1:
          return new Point3D(-7.0f,0.0f,0.0f);
        case 2:
          return new Point3D(-4.0f,0.0f,5.3f);
        default:
          return new Point3D(0.0f);
      }
    }
  }
```

```
  //funzione di intersezione con un raggio: Return
  //distanza o -1.0f se non c'e' intersezione
  double intersect(Ray r) {
    //sostituendo il raggio o+td all'equazione della
    //sfera (td+(o-p)).(td+(o-p)) -R^2 =0 si ottiene
    //l'intersezione. Si deve risolvere
    //t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
    // A=d.d=1 (r.d e' normalizzato)
    // B=2*(o-p).d
    // C=(o-p).(o-p)-R^2
    Point3D op = p.subtract(r.o);
    // calcolo della distanza
    double t;
    // 2*t*B -> semplificato usando b/2 invece di B
    double B=op.dotProduct(r.d);
    double C=op.dotProduct(op)-rad*rad;
    //determinante equazione quadratica
    double det=B*B-C;
    // se e' negativo non c'e' intersezione reale,
    //altrimenti assegna a det il risultato
    if (det<0)
      return -1.0f;
    else {
      det = Math.sqrt(det);
      //ritorna la t piu' piccola se questa e' >0
      //altrimenti vedi quella piu' grande se e' >0
      //se sono tutte negative non c'e' intersezione
      if((t=B-det) > Utilities.EPSILON)
        return t;
      else {
        if((t=B+det) > Utilities.EPSILON)
          return t;
        else
          return -1.0f;
      }
    }
  }

  //funzione che calcola la normale in un punto iP della sfera
  Point3D normal(Point3D iP) {
    //vettore dal centro all'intersezione normalizzato
    return (iP.subtract(p)).getNormalizedPoint();
  }

  @Override
  public String toString() {
    return "Sfera, centro: " + p.toString();
  }
}
```

Classe HourGlass:

```
package primitive;

import primitive.Point3D;
import primitive.Ray;

public class Hourglass {
  // raggio
  public float rad;
  // posizione (centro)
  public Point3D p;

  int matId;

  //costruttore
  public Hourglass(float nrad, Point3D np, int matId) {
    rad=nrad;
    p=np;

    this.matId = matId;
  }

  Point3D normal(Point3D iP) {
    double phi = Math.acos(iP.x);

    double senPhi = Math.sin(phi);
    double cosPhi = Math.cos(phi);

    double rad10 = rad/(double) 10;
    double multiplier = (rad10 + cosPhi*(rad - rad10));

    double theta = Math.acos(iP.y / (senPhi*multiplier));

    double senTheta = Math.sin(theta);
    double cosTheta = Math.cos(theta);

    Point3D derPhiFormula = new Point3D(cosPhi*cosTheta*multiplier,
        cosPhi*senTheta*multiplier, -rad*senPhi);
    Point3D derThetaFormula = new Point3D(-senPhi*senTheta*multiplier,
        senPhi*cosTheta*multiplier, 0);

    return derPhiFormula.crossProduct(derThetaFormula);
  }

  double intersect(Ray r) {
    Point3D op = p.subtract(r.o);
    double t;
```

```
    return 0;
  }
```

Classe Camera:

```
package primitive;

import primitive.Point3D;

//classe che definisce una fotocamera e i suoi parametri
public class Camera {

   // Posizione della fotocamera
   public Point3D eye;
   // Direzione di vista
   private Point3D lookAt;
   // Up vector (generalmente Y)
   private Point3D up;
   // risoluzione sulla X
   private int width;
   // risoluzione sulla Y
   private int height;
   // Distanza del piano di messa a fuoco
   public float d;
   // coordinate locali della fotocamera
   //(si intende il sistema di riferimento con la
   //fotocamera nell'origine)
   public Point3D U;
   public Point3D V;
   public Point3D W;

   //apertura diaframma della camera
   public float aperture = 5;
   public double fuoco = 0;

   //costruttore della fotocamera in cui il
   //sistema ortonormale viene prima impostato a 0
   //per essere poi calcolato basandosi sulla posizione
   //della fotocamera eye (come centro) e sulla
   //direzione in cui si guarda lookAt (come assi)
   public Camera(Point3D neye, Point3D nlookAt,
               Point3D nup, int nwidth, int nheight,
               float nd) {
      eye=neye;
      lookAt=nlookAt;
      up=nup;
```

```
      width=nwidth;
      height=nheight;
      d=nd;
      U=new Point3D();
      V=new Point3D();
      W=new Point3D();

      // creo il sistema ortonormale della fotocamera
      //(normalizzando tutti gli assi del sistema con
      //il metodo .norm())
        //direzione verso cui guardo invertito per
      //convenzione della computer graphics
      //corrisponde alla profondita' z
      W=eye.subtract(lookAt);
      W=W.getNormalizedPoint();

        // prendo -W (direzione sguardo) e faccio il
      //prodotto vettoriale con il vettore up (che da'
      //l'inclinazione della camera) cosi da ottenere
      //un vettore ortogonale ad entrambi
      U=(W.multiplyScalar(-1.0f));
      U=U.crossProduct(up.getNormalizedPoint());
      U=U.getNormalizedPoint();
      //creo l'ultimo vettore ortogonale che e' gia'
      //normalizzato poiche' i vettori di cui facciamo
      //il prodotto vettoriale sono normalizzati
      //(non potevamo prendere esattamente l'up
      //poiche' non e' per forza ortogonale alla
      //direzione dello sguardo)
      V=U.crossProduct(W);
   }
}
```

Classe Box:

```
import java.util.*;

//classe che definisce un Box (e i suoi parametri):
//un contenitore a forma di parallelepipedo con
//dentro gli elementi della scena, che puo'
//essere ulteriormente suddiviso in altri box figli.
//In questo modo si puo' avere una partizione
//sempre piu' fine, in box sempre piu' piccoli
//che contengono gli elementi della scena
public class Box {
   //array dei vertici che definiscono il box
```

```
Point3D[] V = new Point3D[2];
//il parametro lato ci dice il piano con cui il
//box sara' diviso in due
//corrispondenze piano:valore
//xy:0 yz:1 xz:2  cioe'
//lato=0 taglio con il piano z,
//l=1 taglio con il piano x,
//l=2 taglio con il piano y
private int side;

//array degli oggetti che il box contiene
public ArrayList<Obj> objects;

//partition.Box figli: i due "sottobox" in cui il box
//iniziale viene diviso. Nel caso rimangano Null il
//partition.Box non ha nessun figlio
public Box leaf1;
public Box leaf2;

//costruttore box
public Box(Point3D min, Point3D max, int l){
    V[0]=min;
    V[1]=max;
    side = l;
}

//BSP (binary space partition): dato un box lo ripartisce
//in 2 box identici, tagliando il box di partenza con un
//piano indicato dal parametro "lato" della classe partition.Box
//restituisce lo stesso box di partenza, a cui assegna
//pero' i due sottobox generati ai parametri leaf1 e
//leaf2, che in partenza erano settati a NULL
static Box makeChild(Box B) {
  Point3D min= B.V[0];
  Point3D max= B.V[1];
  //l indica il piano con cui si vuole tagliare a
  //meta' il box (vedere la classe partition.Box per la
  //definizione di lato)
  int l= B.side;

  //taglio con il piano z=(min.z+max.z)/2 a meta' del
  //Bound
  if(l==0){
      //vengono costruiti i box leaf1 e leaf2 del box
    B.leaf1= new Box(min,new Point3D(max.x,max.y,(
           min.z+max.z)/2),(short)1);
    B.leaf2= new Box(new Point3D(min.x,min.y,(
           min.z+max.z)/2),max,(short)1);
  }
```

```
  //taglio con il piano x=(min.x+max.x)/2 a meta' del
  //Bound
  if(l==1){
      //vengono costruiti i box leaf1 e leaf2 del box
    B.leaf1= new Box(min,new Point3D((min.x+max.x)/2,
           max.y,max.z),(short)2);;
    B.leaf2= new Box(new Point3D((min.x+max.x)/2,min.y,
           min.z),max,(short)2);
  }
  //taglio con il piano y=(min.y+min.y)/2 a meta' del
  //bound
  if(l==2){
      //vengono costruiti i box leaf1 e leaf2 del box
    B.leaf1= new Box(min,new Point3D(max.x,(min.y+
           max.y)/2,max.z),(short)0);;
    B.leaf2= new Box(new Point3D(min.x,(min.y+max.y)/2,
           min.z),max,(short)0);
  }

  //restituisce il partition.Box di partenza, ma con le leaf
  //aggiornate
  return B;
}

//metodo iterativo che crea una partizione spaziale della
//scena e divide gli oggetti di un box padre tra i suoi
//due box figli.
//il metodo continua fino a quando non si raggiunge il
//valore maxDepth
//notiamo che la variabile liv (livello di profondita'
//all'interno dell'albero) e' globale e viene
//continuamente aggiornata nei passaggi
public static Box setPartition(Box b) {
  //procede solo se il box non e' nullo, il livello di
  //profondita' non ha superato il parametro maxDepth e se
  //ci sono almeno un numero di oggetti maggiore del
  //valore di boxThreshold dentro al box
  if((RenderAction.depthLevel < RenderAction.maxDepth)
      && (b!=null) && (b.objects.size() > RenderAction.boxThreshold)) {
    //aumentiamo il livello di profondita' dell'albero
    RenderAction.depthLevel++;

    //crea i figli del box padre
    b = makeChild(b);
    //assegna ai figli (foglie dell'albero) gli
    //oggetti appartenti al box padre
    b.setLeafObj();

    //continua la partizione iterando il procedimento
```

```
      ///finche' non si raggiunge il livello di massima
      //profondita'
      setPartition(b.leaf1);
      setPartition(b.leaf2);

      //variabile utilizzata per visualizzare lo stato
      //di caricamento dei box
      RenderAction.loadedBoxes++;

      //una volta finita la partizione dei figli
      //ritorna al livello iniziale
      RenderAction.depthLevel--;
    }
    //Il box di ritorno e' aggiornato con le partizioni
    //richieste
    return b;
  }

  //settiamo gli oggetti contenuti nel box
  public void setObjects(ArrayList<Obj> o){
      objects=o;
  }

  //in questo metodo si verifica se esiste
  //l'intersezione di un raggio con il box
  public boolean intersect(Ray r){
   //inizializziamo a + e - infinito i piani
   //near e far
    double tNear = Float.NEGATIVE_INFINITY;
    double tFar = Float.POSITIVE_INFINITY;
    //array con le componenti della direzione del
    //raggio
    double[] direction = {
           r.d.getX(),
           r.d.getY(),
           r.d.getZ()
    };
    //array con le componenti dell'origine del
    //raggio
    double[] origin = {
           r.o.getX(),
           r.o.getY(),
           r.o.getZ()
    };

    //array con le componenti dei vertici del box
    //V[0] e V[1]
    double[] min ={
           V[0].getX(),
```

```
        V[0].getY(),
        V[0].getZ()
};
double[] max = {
        V[1].getX(),
        V[1].getY(),
        V[1].getZ()
};

//per ogni componente faccio vari controlli
//per capire se c'e' intersezione:
for (int i = 0; i < 3; i++) {
  //se nella direzione i non c'e' variazione
  //e l'origine e' fuori dal box non puo'
  //esserci intersezione
  if (direction[i] == 0){
    if((origin[i] < min[i]) || (origin[i] > max[i])) {
      return false;
    }
  } else {
    //si definiscono le intersezioni piu'
    //vicina e piu' lontana
    double T1= (min[i] -origin[i])/direction[i];
    double T2= (max[i] -origin[i])/direction[i];
    //ordina dal piu' piccolo (T1) al piu'
    //grande (T2)
    if(T1>T2){
      //swap(T1,T2);
      double app=T2;
      T2=T1;
      T1=app;
    }
    //voglio il t vicino piu' grande
    if(T1>tNear)
      tNear=T1;
    //voglio il t lontano piu' piccolo
    if(T2<tFar)
      tFar=T2;
    //non c'e' intersezione tra i due
    //segmenti
    if(tNear>tFar){
      return false;
    }
    //il raggio interseca nella direzione
    //opposta
    if(tFar<0){
      return false;
    }
  }
```

```
    }

    return true;
  }
 //in questo metodo si determinano quali
  //oggetti siano contenuti all'interno di uno dei
  //figli del box
  void setLeafObj() {
    //creazione di due arrayList (non uso array
   //perche' non posso sapere dall'inizio quanti
   //oggetti conterranno le suddivisioni del box)
   ArrayList<Obj> leaf1Obj = new ArrayList<>();
   ArrayList<Obj> leaf2Obj = new ArrayList<>();

   //per ogni oggetto presente nel box padre
    for (Obj object : objects) {
      //vogliamo scoprire in quale leaf e'
      //l'oggetto i-esimo
      boolean inleaf1 = false; //flag primo figlio leaf1
      boolean inleaf2 = false; //flag secondo figlio leaf2

      if (object.t != null) {
        //se l'oggetto e' un triangolo
        //si carica il triangolo
        Triangle t = object.t;
        //per ogni vertice
        for (int j = 0; j < 3; j++) {
          //se il box e' stato dimezzato
          //rispetto all'asse z
          //cio' che discriminera' in quale
          //leaf sta' il vertice e' la
          //coordinata z (le altre restano
          //uguali)
          if (side == 0) {
            if (t.vertices[j].z < leaf1.V[1].z) {
              //il vertice e' nel box leaf1
              inleaf1 = true;
            } else {
              //il vertice e' nel box leaf2
              inleaf2 = true;
            }
          }
          //se il box e' stato dimezzato rispetto
          //all'asse x
          if (side == 1) {
            if (t.vertices[j].x < leaf1.V[1].x) {
              inleaf1 = true;
            } else {
              inleaf2 = true;
```

```
      }
    }
    //se il box e' stato dimezzato rispetto
    //all'asse y
    if (side == 2) {
      if (t.vertices[j].y < leaf1.V[1].y) {
        inleaf1 = true;
      } else {
        inleaf2 = true;
      }
    }
  }
}

// se l'oggetto e' una sfera si procede
//diversamente
if (object.s != null) {
  //si carica la sfera
  Sphere s = object.s;
  //si carica il raggio della sfera
  float rad = s.rad;
  //se il box e' stato dimezzato rispetto
  //all'asse z
  if (side == 0) {
    //se mi trovo all'interno della leaf1
    if (s.p.z - rad < leaf1.V[1].z) {
      inleaf1 = true;
    }
    //se mi trovo all'interno della leaf2
    if (s.p.z + rad > leaf1.V[0].z) {
      inleaf2 = true;
    }
  }
  //se il box e' stato dimezzato rispetto
  //all'asse x
  if (side == 1) {
    if (s.p.x - rad < leaf1.V[1].x) {
      inleaf1 = true;
    }
    if (s.p.x + rad > leaf1.V[0].x) {
      inleaf2 = true;
    }
  }
  //se il box e' stato dimezzato rispetto
  //all'asse y
  if (side == 2) {
    if (s.p.y - rad < leaf1.V[1].y) {
      inleaf1 = true;
    }
```

```
          if (s.p.y + rad > leaf1.V[0].y) {
            inleaf2 = true;
          }
        }
      }

      //l'oggetto e' caricato dentro l'array del box
      //figlio in cui si trova
      //nel caso che questo fosse presente in
      //entrambi i box, esso sara' caricato due volte
      if (inleaf1) {
        leaf1Obj.add(object);
      }
      if (inleaf2) {
        leaf2Obj.add(object);
      }
    }

    //una volta conosciuta la grandezza degli array dei
    //box figli possiamo crearli della giusta
    //dimensione abbandonando l'arrayList:questo
    //permette di effettuare una piu' efficente
    //gestione della memoria
    leaf1.objects = new ArrayList<>();
    //per ogni oggetto, salvo l'oggetto dell'arrayList
    //dentro a leaf1.objects
    leaf1.objects.addAll(leaf1Obj);

    //faccio lo stesso per leaf2
    leaf2.objects = new ArrayList<>();
    leaf2.objects.addAll(leaf2Obj);
  }
}
```

Classe Octree:

```
package partition;

import primitive.Obj;
import primitive.Point3D;
import primitive.Ray;
import primitive.Triangle;

import java.util.ArrayList;

/* Questa classe potrebbe essere utilizzata in sostituzione di Box,
 *  in base a se si vuole utilizzare una suddivisione binaria
```

```
 *  oppure una suddivisione con octree (suddivisione in 8 parti
 *   del cubo)
 */

public class Octree {

   //vertici che definiscono il box dell'octree:
    Point3D[] V=new Point3D[2];

    //centro pesato in base alla posizione
    // degli oggetti presenti nel box dell'octree:
    Point3D center=new Point3D();

    //oggetti che il box dell'octree contiene:
    int nObj=0;
    Obj[] objects;

    // puntatori a partition.Box
    //questi corrispondono agli 8 figli del box dell'octree
    Octree[] leaf= null;

   public Octree(Point3D min, Point3D max) {
      V[0]=min;
        V[1]=max;
   }

   void setObjects(Obj[] o, int nO){
        objects=o;
        nObj=nO;

        //calcolo del punto centrale del box
        for(int i=0; i<nObj; i++){

            if(objects[i].t != null){
                for(int j=0; j<3; j++){
                  center = center
                      .add(objects[i].t.vertices[j]
                              .divideScalar(3*nObj));
                }
            }
            if(objects[i].s != null){
                center = center
                    .add(objects[i].s.p.multiplyScalar(1.0f)
                          .divideScalar(nObj));
            }
        }
    }

   boolean intersect(Ray r) {
```

```
 //si inizializza il parametro di intersezione del raggio

//parametro di entrata dal box
double Tnear=Float.NEGATIVE_INFINITY;
//parametro di uscita dal box
double Tfar=Float.POSITIVE_INFINITY;

//si carica il raggio r dentro due array di 3 dimensioni
double[] d={r.d.x,r.d.y,r.d.z};
double[] o={r.o.x,r.o.y,r.o.z};

//si carica il minimo e massimo del box dentro un array di 3 dimensioni
double[] min={V[0].x,V[0].y,V[0].z};
double[] max={V[1].x,V[1].y,V[1].z};

//corrispondenze
//i=0 asse x
//i=1 asse y
//i=2 asse z

//per ogni coppia di piani
for(int i=0;i<3;i++){

  //si controlla se il raggio e' ortogonale all'asse i
  if(d[i]==0){
    //se e' ortogonale mi basta controllare
    // che l'origine del raggio non sia all'interno del box
    if((o[i]<min[i])||(o[i]>max[i])){
      return false;
    }
  }else{

    //altrimenti mi ricavo le intersezioni del raggio
    // con i piani i nelle posizioni min[i] e max[i]
    double T1= (min[i] -o[i])/d[i];
    double T2= (max[i] -o[i])/d[i];

    //si ordinano dal piu' piccolo al piu' grande le intersezioni
    if(T1>T2) {//swap(T1,T2);
      double app=T2;
      T2=T1;
      T1=app;
    }
    //si salva il t piu' vicino piu' grande
    if(T1>Tnear) Tnear=T1;
    //si salva il t piu' lontano piu' piccolo
    if(T2<Tfar) Tfar=T2;

    //se non c'e' intersezione tra i due segmenti
```

```
    // allora non c'e' intersezione col box
    if(Tnear>Tfar){return false;}

    // se il raggio interseca nella direzione sbagliata
    // allo stesso modo non c'e' intersezione col box
    if(Tfar<0){return false;}

  }
 }

 //se si superano tutti i controlli vuol dire che il box e' stato intersecato
 return true;
}

// divide gli oggetti tra i leaf del box padre basandosi
// sugli oggetti che questo contiene
 void setLeafObj() {

  // creo 8 std::vector che conterranno gli array di oggetti appartenenti
  // ai leaf del box
  ArrayList<Obj> leaf0_Objects = new ArrayList<>();
  ArrayList<Obj> leaf1_Objects = new ArrayList<>();
  ArrayList<Obj> leaf2_Objects = new ArrayList<>();
  ArrayList<Obj> leaf3_Objects = new ArrayList<>();
  ArrayList<Obj> leaf4_Objects = new ArrayList<>();
  ArrayList<Obj> leaf5_Objects = new ArrayList<>();
  ArrayList<Obj> leaf6_Objects = new ArrayList<>();
  ArrayList<Obj> leaf7_Objects = new ArrayList<>();

  ArrayList<ArrayList<Obj>> leaf_Objects = new ArrayList<>();
  leaf_Objects.add(0, leaf0_Objects);
  leaf_Objects.add(1, leaf1_Objects);
  leaf_Objects.add(2, leaf2_Objects);
  leaf_Objects.add(3, leaf3_Objects);
  leaf_Objects.add(4, leaf4_Objects);
  leaf_Objects.add(5, leaf5_Objects);
  leaf_Objects.add(6, leaf6_Objects);
  leaf_Objects.add(7, leaf7_Objects);

  for (int i = 0; i < nObj; i++) {

    //vogliamo scoprire in quale leaf e' l'oggetto
    boolean inleaf[] = {false, false, false, false, false, false, false, false};

    //se l'oggetto e' un triangolo
    if (objects[i].t != null) {

      //carico il triangolo
      Triangle t = objects[i].t;
```

```
//per ogni vertice
for (int j = 0; j < 3; j++) {
  //asse X
  if (t.vertices[j].getX() < center.getX()) {

    //asse Y
    if (t.vertices[j].getY() < center.getY()) {

      //asse Z
      if (t.vertices[j].getZ() < center.getZ()) {
        inleaf[0] = true;
      } else {
        inleaf[1] = true;
      }

      //asseY
    } else {
      //asse Z
      if (t.vertices[j].getZ() < center.getZ()) {
        inleaf[2] = true;
      } else {
        inleaf[3] = true;
      }
    }

    //asse X
  } else {
    //asse Y
    if (t.vertices[j].getY() < center.getY()) {

      //asse Z
      if (t.vertices[j].getZ() < center.getZ()) {
        inleaf[4] = true;
      } else {
        inleaf[5] = true;
      }

      //asseY
    } else {
      //asse Z
      if (t.vertices[j].getZ() < center.getZ()) {
        inleaf[6] = true;
      } else {
        inleaf[7] = true;
      }
    }
  }
}
```

```
}

// altrimenti per qualsiasi altro tipo di oggetto utilizziamo
// il bounding box dell'oggetto per verificare in quale box
// l'oggetto e' presente
else {
  Point3D minObj = objects[i].min;
  Point3D maxObj = objects[i].max;
  if (minObj.getX() < center.getX()) {
    if (minObj.getY() < center.getY()) {
      if (minObj.getZ() < center.getZ()) {
        inleaf[0] = true;
      }
      if (maxObj.getZ() > center.getZ()) {
        inleaf[1] = true;
      }
    }
    if (maxObj.getY() > center.getY()) {
      if (minObj.z < center.z) {
        inleaf[2] = true;
      }
      if (maxObj.z > center.z) {
        inleaf[3] = true;
      }
    }
  }
  if (maxObj.x > center.x) {
    if (minObj.y < center.y) {
      if (minObj.z < center.z) {
        inleaf[4] = true;
      }
      if (maxObj.z > center.z) {
        inleaf[5] = true;
      }
    }
    if (maxObj.y > center.y) {
      if (minObj.z < center.z) {
        inleaf[6] = true;
      }
      if (maxObj.z > center.z) {
        inleaf[7] = true;
      }
    }
  }
}

//carico l'oggetto nel vettore relativo
for (int k = 0; k < 8; k++) {
  if (inleaf[k]) {
```

```
            leaf_Objects.get(k).add(leaf_Objects.get(k).size(), objects[i]);
          }
        }

      }


      //ora che so' la grandezza degli array posso crearli nella giusta dimensione
      for (int k = 0; k < 8; k++) {
        int nO = (int) leaf_Objects.get(k).size();
        leaf[k].nObj = nO;
        leaf[k].objects = new Obj[nO];

        for (int i = 0; i < nO; i++) {
          leaf[k].objects[i] = leaf_Objects.get(k).get(i);

          //calcolo del centro dell
          for (int i1 = 0; i1 < nObj; i1++) {
            if (objects[i1].t != null) {
              for (int j = 0; j < 3; j++) {
                center = center.add(objects[i1].t.vertices[j]
                     .multiplyScalar(1.0f).divideScalar(3 * nObj));
              }
            }
            if (objects[i1].s != null) {
              center = center.add(objects[i1].s.p
                  .multiplyScalar(1.0f).divideScalar(nObj));
            }
          }
        }
      }
    }
}
```

Classe PhotonBox:

```
package partition;

import primitive.Photon;
import primitive.Point3D;

import java.util.ArrayList;

/* La partition.PhotonBox, come per la partition.Box per
 *  gli oggetti di scena, suddivide lo spazio per raggruppare i fotoni che
 *  colpiscono la porzione di spazio
 */
```

```
public class PhotonBox {
     //vertici del box
    public Point3D[] V=new Point3D[2];
    //array di fotoni
    public ArrayList<Photon> ph;
    public int nph;

    //dimensione corrispondente alla normale del piano con cui viene suddiviso il
        box
    public int dim;
    //posizione del piano lungo la dimensione dim
    public double planePos;

    public PhotonBox(Point3D v1, Point3D v2, ArrayList<Photon> p) {
        V[0]=v1;
        V[1]=v2;

        ph=p;
        this.nph = p.size();
        if(nph > 0){
        Point3D max=new Point3D(Float.NEGATIVE_INFINITY);
        Point3D min=new Point3D(Float.POSITIVE_INFINITY);

        Point3D median=new Point3D();

        //assegnazione del parametro dim:
        //il parametro viene assegnato alla dimensione in cui i fotoni sono piu'
            distanti tra loro

        //calcolo il bounding box dei fotoni
        for(int i = 0; i< this.nph; i++){

            Point3D pp= ph.get(i).position;

            //vedo se il vertice e' il massimo (se lo e' lo imposto come massimo)
            if(pp.x>max.x)
               max.x= pp.x;
            if(pp.y>max.y)
               max.y= pp.y;
            if(pp.z>max.z)
               max.z= pp.z;
            //vedo se il vertice e' il minimo (se lo e' lo imposto come minimo)
            if(pp.x<min.x)
               min.x= pp.x;
            if(pp.y<min.y)
               min.y= pp.y;
            if(pp.z<min.z)
               min.z= pp.z;
```

```
}

//ora viene scelto il parametro dim in base al lato del bounding box piu'
   lungo
Point3D d= max.subtract(min);
d.abs();
double[] dist={d.x,d.y,d.z};

//piano yz
dim=0;
if(d.y>dist[dim]){dim=1;}
if(d.z>dist[dim]){dim=2;}

//calcolo della mediana

//ordinamento

//si scorrono tutti gli elementi dell'array dei fotoni saltando il primo
for(int i = 1; i< this.nph; i++){
    //si salva il fotone in esame da una parte
    Photon pSaved = ph.get(i);

    //la posizione viene inserita in un array di 3 elementi
    double[] pos_i= {pSaved.position.x,pSaved.position.y,
                        pSaved.position.z};

    //si controlla l'elemento precedente
    int j = i-1;

    //viene caricato l'elemento precedente in un array di 3 elementi
    double[] pos_j={ph.get(j).position.x,ph.get(j)
                .position.y,ph.get(j).position.z};

    //se non e' stata controllata tutta la lista ordinata e l'elemento in
       posizione j
    //e' piu' grande di quello in i allora i due fotoni vengono scambiati
    while ((j >= 0) && (pos_j[dim]>pos_i[dim])){

        //scambio
        ph.set(j + 1, ph.get(j));

        //si scorre j
        j = j-1;

        //queste operazioni vengono effettuate solamente se la lista ordinata
        //non e' stata scandita completamente
        if(j>=0){
        pos_j[0]=ph.get(j).position.x;
        pos_j[1]=ph.get(j).position.y;
```

```
                pos_j[2]=ph.get(j).position.z;}
                //l'elemento che prima era in posizione j ora e' p
                ph.set(j+1, pSaved);
                }
            }

            //il fotone che ci interessa e' quello a meta' di questo array
            int mpos= (int) Math.floor(this.nph / (float) 2);

            //la mediana e' costituita proprio dalla posizione di questo fotone
            median.copy(ph.get(mpos).position);
            //viene quindi assegnata la posizione del piano in base al parametro dim
            double[] m={median.x,median.y,median.z};
            planePos=m[dim];
        }
    }
}
```

Classe DirectIlluminationClass:

```
package renderer;

import primitive.Obj;
import primitive.Point3D;
import primitive.Ray;
import ui.RenderAction;

public class DirectIlluminationClass {
  //Metodo per il calcolo dell'illuminazione diretta, cioe'
  //il contributo che arriva all'oggetto direttamente dalla
  //fonte di luce
  //I parametri in input sono:
  //r: raggio di entrata
  //o: oggetto dal quale partira' il nuovo raggio
  //x e y indici del seme iniziale per la generazione
  //di numeri randomici
  public static Point3D directIllumination(Ray r, Obj o, int x, int y) {
    Utilities utilities = new Utilities();

    //inizializzo le variabili aleatorie comprese tra 0 e
    //1 che utilizzeremo per campionare un punto all'
    //interno dell'oggetto o in maniera equidistribuita
    float rnd1 = 0;
    float rnd2 = 0;
    float rnd3 = 0;

    //si inizializza a 0 il valore in uscita dal processo
```

```
Point3D radianceOutput = new Point3D();

//si carica la normale all'oggetto nel punto r.o
Point3D n1 = o.normal(r.o);

//si carica l'identificativo del materiale
int mId = o.matId;

//definizione e inizializzoazione a null dell'oggetto
//che si andra' ad intersecare
Obj objX;

//per ogni luce
for (int i = 0; i < RenderAction.lights.size(); i++) {
  //carico l'area della luce in esame
  double area = RenderAction.lights.get(i).areaObj;

  //per ogni s campione della luce tra i dirsamps
  //campioni totali per l'illuminazione diretta
  for (int s = 0; s < RenderAction.dirSamps; s++) {
    Point3D B;
    //Caso BSSRDF
    if(RenderAction.material[mId].translucent) {
      /* zv e zr sono le distanze dalla superficie interna ed
       *  esterna per applicare il modello di Jensen
       *  per l'illuminazione empirica di un materiale traslucente
       */
      float zv=0.005f;
      float dv;
      float zr=0.0025f;
      float dr;
      dv=(float) (Math.random()*(60) + 90);
      //lo divido per non avere un numero troppo
      //grande
      dv /= 10000f;
      //rispetto la triangolarita'
      float l2=dv*dv-zv*zv;
      float l=(float) Math.sqrt(l2);
      dr=(float) Math.sqrt(l2+zr*zr);
      //r.o e' il punto in cui arriva il raggio:
      //per rispettare il modello di Jensen e'
      //necessario spostarlo di l per un
      //angolo casuale

      //utilizzo questa variabile tt perche' non
      //posso usare il valore x+y*width nell'array
      //dirSamples1[], altrimenti l'ultimo indice
      //sarebbe fuori dal range (ricordo che la
      //misura e' width*height ma gli indici vanno da 0 a
```

```
//width*height-1)
int tt = x+y* RenderAction.width;
//allora faccio l'if per tt<width*height cosi' da
//accertarmi che non sia considerato l'indice
//width*height-esimo
if(tt < RenderAction.width * RenderAction.height) {
  rnd1 =
     Utilities.generateRandom(RenderAction.dirSamples1[tt]);
  rnd2 =
     Utilities.generateRandom(RenderAction.dirSamples2[tt]);
  rnd3 =
     Utilities.generateRandom(RenderAction.dirSamples3[tt]);
}

//genero due angoli casuali
float rndPhi = 2 * Utilities.MATH_PI *(rnd1);
float rndTeta = (float)Math.acos((float)Math.
    sqrt(rnd2));
float cosP=(float)Math.cos(rndPhi);
float cosT=(float)Math.cos(rndTeta);
float sinT=(float)Math.sin(rndTeta);
double px=r.o.x+l*cosP*cosT;
double py=r.o.y+l*cosP*sinT;
double pz=r.o.z+l*sinT;
Point3D newPoint=new Point3D(px,py,pz);

//si carica il punto campionato sulla luce
Point3D p = RenderAction.lights.get(i).randomPoint(rnd1,rnd2,rnd3);

//la direzione e' quella che congiunge il punto
//r.o al punto campionato
Point3D dir = (p.subtract(newPoint));
//salviamo la distanza tra i due punti
double norma = dir.normalize();
dir = dir.getNormalizedPoint();
//creazione del raggio d'ombra diretto verso
//la luce
double cosTheta=r.d.dotProduct(n1);
Point3D Ftheta=
  RenderAction.material[mId].getFresnelCoefficient(cosTheta);

dir.x *= Ftheta.x;
dir.y *= Ftheta.y;
dir.z *= Ftheta.z;

Ray directRay= new Ray(newPoint, dir);

//viene inizializzato l'oggetto che il raggio
//intersechera' con l'oggetto che il raggio
```

```
//punta
objX = RenderAction.lights.get(i);
//si inizializza la massima distanza a cui il
//raggio puo' arrivare

//verifica del fattore di visibilita'
if (utilities.intersect(directRay, objX)) {
  utilities.inters = Utilities.inf;
  utilities.intersObj =null;
  //vengono caricati i dati della luce:
  //normale nel punto p
  Point3D n2 = RenderAction.lights.get(i).normal(p);
  //identificativo del materiale della luce
  int lid = RenderAction.lights.get(i).matId;
  //calcoliamo la BSSRDF
  double cosPsi=directRay.d.dotProduct(n1);
  Point3D Fpsi= RenderAction.material[mId].getFresnelCoefficient(
      cosPsi);
  Point3D one=new Point3D(1.0f);


  Point3D pi4=new Point3D(4* Utilities.MATH_PI);
  //i valori si sigmas e sigmaa sono specifici per la giada
  Point3D sigmas=new Point3D(0.657f,0.786f,0.9f);
  Point3D sigmaa=new Point3D(0.2679f,0.3244f,0.1744f);

  Point3D sigmat=sigmaa.add(sigmas);
  Point3D sigmatr
     =(sigmaa.multiplyComponents(sigmat))
.multiplyScalar(3.0f);
  sigmatr= Point3D.getSquareCompPoint(sigmatr);
  Point3D alpha=sigmas.divideComponents(sigmat);

  Point3D expdr=sigmatr.multiplyScalar(dr*-1.0f);
  float dr3=dr*dr*dr;
  Point3D edivdr=(Point3D.exponent(expdr)).
      divideScalar(dr3);

  Point3D expdv=sigmatr.multiplyScalar(dv*-1.0f);
  float dv3=dv*dv*dv;
  Point3D edivdv=(Point3D.exponent(expdv)).divideScalar(dv3);
  Point3D rPart=
     (((sigmatr.multiplyScalar(dr)).add(one))
.multiplyComponents(edivdr))
      .multiplyScalar(zr);
  Point3D vPart=
     (((sigmatr.multiplyScalar(dv)).add(one))
.multiplyComponents(edivdv))
      .multiplyScalar(zv);
```

```
    Point3D Rd=(alpha.divideComponents(pi4))
        .multiplyComponents(rPart.add(vPart));

    B=(Rd.multiplyComponents(Fpsi).multiplyComponents(Ftheta)).
        divideScalar(Utilities.MATH_PI);

    //enfatizzo il colore verde
    B.x=B.x*0.31f;
    B.y=B.y*0.65f;
    B.z=B.z*0.246f;

    //calcolo dell'illuminazione diretta
    //vengono definiti i seguenti float3 per
    //leggibilita' del risultato
    double dirN1N2=(-dir.dotProduct(n1))*(dir.dotProduct(n2));
    float norma2=(float) Math.pow(norma, 2);
    radianceOutput = radianceOutput.
        add(RenderAction.material[lid].emittedLight.
            multiplyComponents(B).multiplyScalar(area).
            multiplyScalar(dirN1N2).multiplyScalar(5)).
        divideScalar(norma2);
  }
}

//Caso BRDF
int tt =x+y* RenderAction.width;
if(tt < RenderAction.width * RenderAction.height) {
  rnd1 =
    Utilities.generateRandom(RenderAction.dirSamples1[tt]);
  rnd2 =
    Utilities.generateRandom(RenderAction.dirSamples2[tt]);
  rnd3 =
    Utilities.generateRandom(RenderAction.dirSamples3[tt]);
}

//si carica il punto campionato sulla luce
Point3D p =
 RenderAction.lights.get(i).randomPoint(rnd1,rnd2,rnd3);

//la direzione e' quella che congiunge il punto
//r.o al punto campionato
Point3D dir = (p.subtract(r.o));
//salviamo la distanza tra i due punti
double norma = dir.normalize();
dir = dir.getNormalizedPoint();

//creazione del raggio d'ombra diretto verso
//la luce
```

```
        Ray directRay= new Ray(r.o, dir);

        //viene inizializzato l'oggetto che il raggio
        //intersechera' con l'oggetto che il raggio
        //punta
        objX = RenderAction.lights.get(i);
        //si inizializza la massima distanza a cui il
        //raggio puo' arrivare

        //verifica del fattore di visibilita'
        if (utilities.intersect(directRay, objX)) {
          utilities.inters = Utilities.inf;
          utilities.intersObj = null;
          //vengono caricati i dati della luce:
          //normale nel punto p
          Point3D n2 = RenderAction.lights.get(i).normal(p);
          //identificativo del materiale della luce
          int lid = RenderAction.lights.get(i).matId;
          //calcoliamo la BRDF
          B = RenderAction.material[mId].C_T_BRDF(directRay, r, n1);

          //calcolo dell'illuminazione diretta
          //vengono definiti i seguenti float3 per
          //leggibilita' del risultato
          double dirN1N2=(-dir.dotProduct(n1))*(dir.dotProduct(n2));
          float norma2=(float) Math.pow(norma, 2);
          radianceOutput = radianceOutput
              .add(RenderAction.material[lid].emittedLight
                  .multiplyComponents(B)
                  .multiplyScalar(area)
                  .multiplyScalar(dirN1N2)
                     .multiplyScalar(RenderAction
            .material[mId].translucent ? 0.35 : 1))
              .divideScalar(norma2);
        }
      }
    }

    //dividiamo per il numero di campioni utilizzati
    //nell'estimatore di Monte Carlo
    radianceOutput
      = radianceOutput.divideScalar(RenderAction.dirSamps *
          RenderAction.lights.size());
    return radianceOutput;
  }
}
```

Classe EmittedObjRadianceClass:

```
package renderer;

import primitive.Obj;
import primitive.Point3D;
import ui.RenderAction;

public class EmittedObjRadianceClass {
  // Metodo che restituisce la radianza emessa dall'oggetto o in direzione r:
  static Point3D emittedObjRadiance(Obj o) {
    //carico l'indice del material
    int mId = o.matId;

    //dichiaro e inizializzo a 0 il valore in uscita
    Point3D radianceOutput = new Point3D();

    //con il seguente if si controlla se il materiale
    //emette effettivamente luce
    if (RenderAction.material[mId].emittedLight.max() > 0) {
      double Ler = RenderAction.material[mId].emittedLight.max();
      Point3D Ler3 = new Point3D(Ler);
      //con il metodo clamp3 si evita che la radianza in
      //uscita superi il valore massimo di radianza: 1
      Point3D.clamp3(Ler3);
      radianceOutput = Ler3;
    }
    return radianceOutput;
  }
}
```

Classe FinalGatheringClass:

```
package renderer;

import primitive.Obj;
import primitive.Point3D;
import primitive.Ray;

import static renderer.DirectIlluminationClass.directIllumination;
import static renderer.EmittedObjRadianceClass.emittedObjRadiance;
import static renderer.FinalIndirectClass.finalIndirect;

public class FinalGatheringClass {
  // metodo per il Final Gathering, che si serve di una sola
  // iterazione del raytracing stocastico, nella quale si
  // raccolgono (gathering) le informazioni ottenute dalla
```

```
  // soluzione precalcolata di radiosita', che viene considerata
  // come se fosse la luce emessa da un emettitore diffusivo al punto osservato,
  // tipo Warn, e quindi dĂ  una illuminazione proporzionale al coseno dell'angolo
  // di deviazione della normale a quel punto rispetto alla direzione
      dell'osservatore.
  // Viene poi emesso da tale punto un certo numero di raggi e si sommano i
      contributi
  // che questi raggi vedono colpendo il resto della scena.
  // Questo aggiunge alla soluzione diffusiva della radiositĂ 
  // precalcolata un effetto di illuminazione riflessa
  // Possiamo calcolare questi contributi come illuminazione diretta e indiretta

  static Point3D finalGathering(Ray viewRay, int x, int y, Obj o) {
    // Variabile per la radianza
    Point3D radianceOutput = new Point3D();

    // Illuminazione emessa
    Point3D le= emittedObjRadiance(o);
    radianceOutput = radianceOutput.add(le);

    // Illuminazione diretta
    Point3D di= directIllumination(viewRay, o, x, y);
    radianceOutput = radianceOutput.add(di);

    // Illuminazione indiretta
    // Nota: l'illuminazione non Ăš ricorsiva quindi solo un raggio con piĂč sample
        saranno inviati
    Point3D fi= finalIndirect(viewRay, o, x, y);
    radianceOutput = radianceOutput.add(fi);

    return radianceOutput;
  }
}
```

Classe FinalIndirectClass:

```
package renderer;

import primitive.Obj;
import primitive.Point3D;
import primitive.Ray;
import ui.RenderAction;

public class FinalIndirectClass {
  //metodo per l'illuminazione indiretta
  //I parametri in input sono:
  //r: raggio di entrata
```

```
//o: oggetto dal quale partira' il nuovo raggio
//x e y indici del seme iniziale per la generazione
//di numeri randomici
static Point3D finalIndirect(Ray r, Obj o, int x, int y) {
  Utilities utilities = new Utilities();

  //inizializzo a 0 il valore che resitituiro' alla fine
  //del processo
  Point3D radianceOutput = new Point3D();

  //normale dell'oggetto in esame
  Point3D n1 = o.normal(r.o);

  int mId = o.matId;

  //per ogni s campione della luce tra gli aosamps
  //campioni totali per l'illuminazione indiretta
  for (int s = 0; s < RenderAction.aoSamps; s++) {
    //quindi distribuito uniformemente sull'emisfero
    Point3D dir;
    float rndX = 0.0f;
    float rndY = 0.0f;

    //utilizzo questa variabile tt perche' non
    //posso usare il valore x+y*width nell'array
    //dirSamples1[], altrimenti l'ultimo indice
    //sarebbe fuori dal range (ricordo che la
    //misura e' width*height ma gli indici vanno da 0 a
    //width*height-1)
    int tt =x+y* RenderAction.width;
    //allora faccio l'if per tt<width*height cosi' da
    //accertarmi che non sia considerato l'indice
    //width*height-esimo
    if(tt< RenderAction.width * RenderAction.height) {
      rndX =
            Utilities.generateRandom(RenderAction.aoSamplesX[tt]);
      rndY =
            Utilities.generateRandom(RenderAction.aoSamplesY[tt]);
    }

    //distribuisco i numeri random sull'emisfero
    float rndPhi = 2 * Utilities.MATH_PI *(rndX);
    float rndTeta = (float)Math.acos((float)Math.
        sqrt(rndY));

    // Create onb (ortho normal basis) on iP punto di
    //intersezione
    Point3D u, v, w;
    w = n1;
```

```
      //vettore up (simile a (0,1,0))
      Point3D up=new Point3D(0.0015f, 1.0f, 0.021f);
      v = w.crossProduct(up);
      v = v.getNormalizedPoint();
      u = v.crossProduct(w);

      float cosPhi=(float) Math.cos(rndPhi);
      float sinTeta=(float) Math.sin(rndTeta);
      float sinPhi=(float) Math.sin(rndPhi);
      float cosTeta=(float) Math.cos(rndTeta);
      dir = (u.multiplyScalar(cosPhi*sinTeta))
          .add(v.multiplyScalar(sinPhi*sinTeta))
          .add(w.multiplyScalar(cosTeta));
      dir = dir.getNormalizedPoint();

      //creo il raggio dal punto di intersezione ad un
      //punto a caso sull'emisfero
      Ray reflRay = new Ray(r.o, dir);
      // il metodo va' avanti solo se interseca un
      //oggetto e se questo oggetto non e' una luce
      //(poiche' la luminosita' diretta l'abbiamo gia'
      //considerata)

      if (utilities.intersect(reflRay, null)) {
        utilities.inters = Utilities.inf;
        Obj objX = utilities.intersObj;
        utilities.intersObj = null;
        if(RenderAction.material[objX.matId].emittedLight.max() == 0) {
          double _area = 1 / (objX).area();
          radianceOutput
              = radianceOutput.add(((objX).P)
               .multiplyComponents(RenderAction.material[mId].diffusionColor)
         .multiplyScalar(_area));
        }
      }
    }

    radianceOutput = radianceOutput.divideScalar(RenderAction.aoSamps);
    return radianceOutput;
  }
}
```

Classe JacobiStocClass:

```
package renderer;

import primitive.Obj;
```

```
import primitive.Point3D;
import primitive.Ray;
import ui.InterfaceInitialiser;
import ui.RenderAction;

import java.text.DecimalFormat;

public class JacobiStocClass {
  //Metodo per il calcolo della radiosita' della scena:
  //in questa funzione si utilizza il metodo di Jacobi
  //stocastico per calcolare il valore di potenza di ogni
  //patch della scena
  //si usa la variabile globale GlobalObject in modo da
  //conservare i valori aggiornati delle potenze dei vari
  //oggetti

  /* Ci sono diversi problemi presenti nel metodo:
   *  -> la scelta del numero di sample da inviare per
   *      Jacobi stocastico (da variare in base alla potenza
   *      della scena)
   *  -> la suddivisione in varie patches della scena
   *      (al momento formata da sole semplici patches, rendendo
   *       l'intero render troppo approssimativo)
   */

  public static void jacobiStoc(int nObj) {
    Utilities utilities = new Utilities();

    //definisco la potenza residua totale delle patch:
    Point3D Prtot=new Point3D();
    //definisco la potenza di ogni patch della scena:
    Point3D[] P=new Point3D[nObj];

    //definisco la potenza residua di ogni patch della
    //scena:
    Point3D[] Pr=new Point3D[nObj];

    //inizializzo a (0,0,0) power e Pr
    for(int i=0; i<nObj;i++) {
      P[i]=new Point3D();
      Pr[i]=new Point3D();
    }

    //inizializzo l'array objX in cui inizialmente gli
    //oggetti sono nulli, ma poi vengono settati con
    //l'oggetto intersecato dal raggio che parte da una
    //patch i-esima presa in consideraazione:l'oggetto
    //intersecato e' proprio la patch j su cui sara'
    //rilasciata la potenza totale delle 3 componenti rgb
```

```
Obj[] objX=new Obj[nObj];

//Vengono caricati i valori iniziali di Luminosita'
//Emessa per ogni patch della scena (Pe)
for(int i=0;i<nObj;i++) {
  //viene caricata l'area dell'oggetto i-esimo
  double area= RenderAction.globalObjects.get(i).areaObj;
  //se l'area e' piu' piccola della precisione di
  //calcolo allora impostiamo l'area a 0
  if(area < Utilities.EPSILON) {
    area=0;
  }

  //potenza della luce
  //Potenza emessa dalla patch i:
  //(potenza emessa)*(pi greco)*(area)
  Point3D LP
     = RenderAction.material[RenderAction.globalObjects.get(i)
  .matId].emittedLight
          .multiplyScalar(Utilities.MATH_PI)
             .multiplyScalar(area);

  //viene calcolata la potenza totale iniziale:
  //(potenza residua della patch)+
  //+(potenza emessa dalla patch)
  Prtot=Prtot.add(LP);

  //viene salvata nell'array power la potenza dell'
  //elemento i
  P[i].copy(LP);

  //viene salvata nell'array Pr la potenza residua
  //dell'elemento i
  Pr[i].copy(LP);

  //viene calcolato l'errore di approssimazione
  //(con cui e' possibile fermare il metodo)
  RenderAction.err += Math.pow(LP.average(),2);
}//fine for

//dopo aver aggiunto dei quadrati dobbiamo fare la
//radice del risultato finale

//iterazioni del metodo di Jacobi:
//Si continuano le iterazioni finche' l'energia
//rilasciata non diventa minore di un certo valore
///(verificato dalla variabile err)
//o gli steps superano gli steps massimi (verificato
//dalla variabile maxSteps).
```

```
//Devono essere vere entrambe

RenderAction.steps = 0;

/* jacobiSamps possono essere impostati secondo la geometria della scena,
 *  cosĂŹ da generare i campioni
 */
while ((RenderAction.err > RenderAction.maxErr)
    && (RenderAction.steps < RenderAction.maxSteps)) {
  //percentuale di completamento
  InterfaceInitialiser.label.setText("Completamento Jacobi "
      + new DecimalFormat("###.##")
      .format((RenderAction.steps/(float) RenderAction.maxSteps)*100));
  //viene inizializzato un seme iniziale casuale
  //da 0 a 30000
  int s =(int) Math.floor(Math.random()*(30001));
  //inizializzo per le tre componenti una variabile
  //per il numero di sample utilizzati per la patch
  //(N*) e una per il numero di sample utilizzati
  //finora (Nprev*)
  int NprevX=0;
  int NprevY=0;
  int NprevZ=0;
  int NX, NY, NZ;

  //potenza residua totale nelle tre componenti
  double Prt= Prtot.x+Prtot.y+Prtot.z;
  //campioni distribuiti in base alla potenza
  //totale di ciascuna componente RGB che chiamiamo
  //qui (x,y,z)

  Point3D samps = new Point3D(
      RenderAction.jacobiSamps*Prtot.x,
      RenderAction.jacobiSamps*Prtot.y,
      RenderAction.jacobiSamps*Prtot.z);
  samps=samps.divideScalar(Prt);
  //parametro che ci permette di contare, quindi
  //utilizzare tutti i campioni che erano stati
  //previsti

  Point3D q=new Point3D();
  //per ogni patch della scena

  for(int i=0;i<nObj;i++) {
    //salviamo nella variabile locale o l'oggetto
    //i-esimo dell'array globalObjects
    Obj o= RenderAction.globalObjects.get(i);

    //probabilita' con cui viene scelta la patch i
```

```
Point3D pi=(Pr[i].divideComponents(Prtot));

q=q.add(pi);

//componente R:

//calcoliamo il numero di campioni utilizzati
//per la patch i in base alla sua potenza
//e anche in base a quanti campioni sono gia'
//stati usati
NX= (int) Math.round((q.x*samps.x)+NprevX*(-1));

//per ogni campione dell'elemento i
for (int j=0; j<NX; j++) {
  //creo una direzione randomica uniformemente
  //distribuita sull'emisfero frontale alla
  //patch
  //definisco la variabile che rappresentera'
  //la direzione
  Point3D dir;
  //creo 3 variabili randomiche che si basano
  //sul seme iniziale casuale s definito
  //randomicamente
  float rndX = Utilities.generateRandom(s);
  float rndY = Utilities.generateRandom(s);
  float rndZ = Utilities.generateRandom(s);

  //Inverse Cumulative Distribuction Function
  //creiamo la base ortonormale per generare
  //la direzione del raggio ffRay
  float rndPhi=2* Utilities.MATH_PI *(rndX);
  float rndTeta=(float) Math.acos(Math.sqrt(rndY));

  //dichiaro e inizializzo un punto scelto
  //uniformemente sulla patch i
  Point3D rndPoint = o.randomPoint(rndX, rndY, rndZ);

  //Si crea ora la base ortonormale
  Point3D u;
  Point3D v;
  //settiamo width come la normale all'oggetto nel
  //punto rndPoint
  Point3D w = o.normal(rndPoint);

  //vettore up (simile a (0,1,0))
  Point3D up=new Point3D(0.0015f,1.0f,0.021f);
  //il prodotto vettoriale tra width e up mi genera
  //il vettore v normale a entambi, che
  //normalizzo
```

```
v=w.crossProduct(up);
v=v.getNormalizedPoint();
//il prodotto vettoriale tra v e width mi genera
//il vettore u normale a entambi
//dal momento che i vettori v e width sono gia'
//normali, non c'e' bisogno di noemalizzare
//il vettore u)
u=v.crossProduct(w);

//ora che abbiamo la base ortonormale,
//possiamo calcolare la direzione dir.
//salvo in delle cariabili i calori di seno
//e coseno necessari per il calcolo di dir
float cosRndPhi=(float) Math.cos(rndPhi);
float sinRndTeta=(float) Math.sin(rndTeta);
float sinRndPhi=(float) Math.sin(rndPhi);
float cosRndTeta=(float) Math.cos(rndTeta);
//poi normalizzato
dir = u.multiplyScalar(cosRndPhi*sinRndTeta)
    .add(v.multiplyScalar(sinRndPhi*sinRndTeta))
    .add(w.multiplyScalar(cosRndTeta));
dir = dir.getNormalizedPoint();

//creo il raggio per scegliere la patch j
//con probabilita' uguale al fattore di forma
//tra la patch i e quella j
Ray ffRay=new Ray(rndPoint,dir);

//inizializzo a null l'oggetto intersecato
objX[i]=null;

//l'oggetto intersecato e' proprio la patch j
//su cui sara' rilasciata la potenza totale
//della componente rossa

if(utilities.intersect(ffRay, objX[i])) {
  //resetto inters uguale a inf in modo da
  //avere il giusto valore di partenza la
  //prossima volta che si utilizzera' il
  //metodo intersect()
  utilities.inters = Utilities.inf;
  //salvo nell'elemento i-esimo dell'array
  //objX l'elemento intersecato dal raggio
  //ffRay
  objX[i]= utilities.intersObj;
  //resetto intersObj=null in modo da avere
  //il giusto valore di partenza la prossima
  //volta che si utilizzera' il
  //metodo intersect()
```

```
    utilities.intersObj = null;
    //salviamo la potenza rilasciata all'interno
    //della struttura dell'oggetto: essa si
    //sommera' con la potenza residua parziale
    //che l'oggetto ha raggiunto finora; solo
    //alla fine del processo infatti avremo
    //la potenza residua totale della patch
    objX[i].P.x +=
        RenderAction.material[objX[i].matId]
             .diffusionColor.x*(Prtot.x)/(samps.x);
  }
}
//fine for per la componente rossa
//aggiorniamo il numero di campioni usati per
//questa componente
NprevX += NX;

//componente G:

//campioni per la patch i
NY= (int) Math.round((q.y*samps.y)+NprevY*(-1));

//per ogni campione sull'elemento i
for (int j=0; j<NY; j++) {
  Point3D dir;

  float rndX = Utilities.generateRandom(s);
  float rndY = Utilities.generateRandom(s);
  float rndZ = Utilities.generateRandom(s);

  float rndPhi=2* Utilities.MATH_PI *(rndX);
  float rndTeta=(float) Math.acos(Math.sqrt(
      rndY));

  //punto scelto uniformemente nella patch i:
  Point3D rndPoint = o.randomPoint(rndX, rndY, rndZ);

  //base ortonormale
  Point3D u,v;
  Point3D w = o.normal(rndPoint);

  //vettore up (simile a (0,1,0))
  Point3D up=new Point3D(0.0015f,1.0f,0.021f);
  v=w.crossProduct(up);
  v=v.getNormalizedPoint();
  u=v.crossProduct(w);

  float cosRndPhi=(float) Math.cos(rndPhi);
  float sinRndTeta=(float) Math.sin(rndTeta);
```

```
  float sinRndPhi=(float) Math.sin(rndPhi);
  float cosRndTeta=(float) Math.cos(rndTeta);
  dir=u.multiplyScalar(cosRndPhi*sinRndTeta).
      add(v.multiplyScalar(
          sinRndPhi*sinRndTeta)).add(
      w.multiplyScalar(cosRndTeta));
  dir=dir.getNormalizedPoint();

  //creo il raggio per scegliere la patch j
  //con probabilita' uguale al fattore di forma
  //tra la patch i e quella j
  Ray ffRay=new Ray(rndPoint,dir);
  objX[i]=null;

  if(utilities.intersect(ffRay, objX[i])) {
    utilities.inters = Utilities.inf;
    objX[i]= utilities.intersObj;
    utilities.intersObj =null;
    objX[i].P.y +=
        RenderAction.material[objX[i].matId].diffusionColor.y*(Prtot.y)/(samps.y);
  }
}//fine for per la componente verde
NprevY += NY;

//componente B:

//campioni per la patch i
NZ= (int) Math.round((q.z*samps.z)+NprevZ*(-1));

//per ogni campione sull'elemento i
for (int j=0; j<NZ; j++){
  Point3D dir;
  float rndX;
  float rndY;
  float rndZ;

  rndX= Utilities.generateRandom(s);
  rndY= Utilities.generateRandom(s);
  rndZ= Utilities.generateRandom(s);

  //direzione casuale
  float rndPhi=2* Utilities.MATH_PI *(rndX);
  float rndTeta=(float) Math.acos(Math.sqrt(rndY));

  //punto scelto uniformemente nella patch i
  Point3D rndPoint=null;
  rndPoint= o.randomPoint(rndX, rndY,rndZ);

  //base ortonormale
```

```
        Point3D u,v;
        Point3D w = o.normal(rndPoint);

        //vettore up (simile a (0,1,0))
        Point3D up=new Point3D(0.0015f,1.0f,0.021f);
        v=w.crossProduct(up);
        v=v.getNormalizedPoint();
        u=v.crossProduct(w);

        float cosRndPhi=(float) Math.cos(rndPhi);
        float sinRndTeta=(float) Math.sin(rndTeta);
        float sinRndPhi=(float) Math.sin(rndPhi);
        float cosRndTeta=(float) Math.cos(rndTeta);
        dir = u.multiplyScalar(cosRndPhi*sinRndTeta)
            .add(v.multiplyScalar(sinRndPhi*sinRndTeta))
            .add(w.multiplyScalar(cosRndTeta));

        dir=dir.getNormalizedPoint();

        //creo il raggio d'ombra dal punto di
        //intersezione ad un punto a caso sull'
        //emisfero
        Ray ffRay=new Ray(rndPoint,dir);
        objX[i] = null;

        if(utilities.intersect(ffRay,objX[i])) {
          utilities.inters = Utilities.inf;
          objX[i]= utilities.intersObj;
          utilities.intersObj = null;
          objX[i].P.z +=
              RenderAction.material[objX[i].matId]
                   .diffusionColor.z*(Prtot.z)/(samps.z);
        }
      }//fine for per la componente blu

      NprevZ += NZ;
    }//fine for per le patch della scena SAS1

    //una volta terminata la fase di shooting si
    //riaggiornano le componenti di Potenza residua,
    //Pr, Potenza power, e si azzerano le potenze
    //residue parziali salvate negli oggetti
    //azzeramento della potenza totale:

    Prtot=new Point3D(0);

    for(int i=0; i < nObj; i++) {
      //aggiornamento delle Potenze (vengono aggiunte le potenze residue totali
      // immagazzinate dalle patch durante il processo,
```

```
        // secondo la radiositĂ  basata su potenza incrementale)
        P[i] = P[i].add(RenderAction.globalObjects.get(i).P);
        //aggiornamento delle potenze residue totali
        Pr[i].copy(RenderAction.globalObjects.get(i).P);
        //calcolo dell'errore
        RenderAction.err += Math.pow(Pr[i].average(),2);
        //calcolo dell'energia residua totale
        Prtot = Prtot.add(Pr[i]);
        //azzeramento della potenza residua parziale contenuta nella patch
        RenderAction.globalObjects.get(i).P.copy(new Point3D());
      }

      RenderAction.steps++;
    }

    //I valori ottenuti vengono salvati su ciascuna
    //patch nella variabile power (in cui durante il processo
    //veniva salvata la potenza residua parziale)
    for(int i=0; i<nObj; i++) {
      (RenderAction.globalObjects.get(i).P).copy(P[i]);
    }
  }
}
```

Classe ParallelProcessRadiance:

```
package renderer;

import primitive.Camera;
import primitive.Obj;
import primitive.Point3D;
import primitive.Ray;
import renderer.Renderer;
import ui.InterfaceInitialiser;
import ui.RenderAction;

import java.text.DecimalFormat;

import static renderer.RadianceClass.radiance;

/* La classe renderer.ParallelProcessRadiance presenta
 *  il blocco di codice che si occupa di reindirizzare il rendering del
 *  pixel nei metodi interessati (scelti nella classe Render)
 * Tutte queste azioni avvengono in un thread apposito,
 *  creato nella clase renderer.Runner
 */
```

```
public class ParallelProcessRadiance implements Runnable {
  private int y;
  private Camera cam;

  ParallelProcessRadiance(int y, Camera cam) {
    this.y = y;
    this.cam = cam;
  }

  @Override
  public void run() {
    Utilities utilities = new Utilities();

    double percentY = (y*100 / (float) RenderAction.height);
    InterfaceInitialiser.label.setText("Percentuale di completamento radianza: "
        + new DecimalFormat("###.##").format(percentY));

    //per tutte le colonne
    for(int x = 0; x <= RenderAction.width; x++) {
      // Ora siamo nel pixel
      // r e' la radianza: in questo caso e' tutto nero
      // Radianza della scena
      Point3D sceneRadiance = new Point3D(0.0f);

      // Loop per ogni campione
      for (int s = 0; s < RenderAction.samps; s++) {
        //inizializiamo un raggio per la camera
        Ray cameraRay;

        //transformazione delle variabili x e y in
        //float corrispondono alla posizione che
        //cameraRay deve raggiungere
        float raster_x = (float)x;
        float raster_y = (float)y;

        //origine del raggio della fotocamera
        Point3D origin = cam.eye;

        //se ho piu' di un campione allora
        //distribuisco gli altri campioni in modo
        //casuale
        if (s > 0) {
          float rndX=0;
          float rndY=0;

          //utilizzo questa variabile tt perche' non
          //posso usare il valore x+y*width nell'array
          //samplesX[], altrimenti l'ultimo indice
          //sarebbe fuori dal range (ricordo che la
```

```
  //misura e' width*height ma gli indici vanno da 0 a
  //width*height-1)
  int tt = x+y* RenderAction.width;
  //allora faccio l'if per tt<width*height cosi' da
  //accertarmi che non sia considerato l'indice
  //width*height-esimo
  if(tt < RenderAction.width * RenderAction.height) {
    // gli passo il numero random da cui
    //siamo partiti all'interno del pixel
    rndX = Utilities.generateRandom(RenderAction.samplesX[tt]);
    rndY = Utilities.generateRandom(RenderAction.samplesY[tt]);
  }

  // Gestisco la messa a fuoco dell'immagine finale,
  // valutando il diaframma e il punto di messa a fuoco
  raster_x +=
      Math.cos(2 * Utilities.MATH_PI * rndX)*cam.aperture *rndY;
  raster_y +=
      Math.sin(2 * Utilities.MATH_PI * rndX)*cam.aperture *rndY;
  Point3D camUFuoco =
      cam.U.multiplyScalar(cam.fuoco*(x - raster_x));
  Point3D camVFuoco =
      cam.V.multiplyScalar(cam.fuoco*(y - raster_y));

  origin = origin.add(camUFuoco).add(camVFuoco);
}

// prediamo la direzione della fotocamera
//ray_direction e' calcolato con l'ONB(base
//ortonormale) della fotocamera
//il raggio dalla fotocamera al campione sara'
//data dalla combinazione lineare dell'ONB
//della fotocamera
//centro il piano rispetto alla fotocamera
//sottraendo width/2 alla componente in x e height/2
//alla componente in y infine la distanza z
//tra la fotocamera e il piano e' cam.d
Point3D ray_direction =
    (cam.U.multiplyScalar(raster_x - 0.5f * RenderAction.width))
    .add(cam.V.multiplyScalar(raster_y - 0.5f * RenderAction.height))
    .add(cam.W.multiplyScalar(-cam.d));
ray_direction=ray_direction.getNormalizedPoint();

//Ora si crea il raggio della fotocamera
cameraRay = new Ray(origin, ray_direction);

//dichiaro e inizializzo la variabile t in cui
//salveremo il punto di intersezione fra
//l'oggetto considerato e cameraRay
```

```
    double t;
    //inizializzo a null l'oggetto intersecato
    //dal raggio
    Obj o = null;
    //intersezione del raggio con gli elementi
    //della scena:
    if(utilities.intersect(cameraRay, o)) {
      //pongo t uguale al valore di intersezione
      //memorizzato nella variabile globale inters
      t = utilities.inters;
      //resetto inters uguale a inf in modo da
      //avere il giusto valore di partenza la
      //prossima volta che si utilizzera'
      //il metodo intersect()
      utilities.inters = Utilities.inf;
      //salvo nella variabile o objX l'elemento
      //intersecato dal raggio cameraRay
      o = utilities.intersObj;
      //resetto intersObj=null in modo da avere
      //il giusto valore di partenza la prossima
      //volta che si utilizzera' il metodo
      //intersect()
      utilities.intersObj=null;
      //si calcola il punto di intersezione
      Point3D iP = (cameraRay.o)
          .add(cameraRay.d.multiplyScalar(t));
      //viene creato il primo raggio per il
      //calcolo della radianza
      //questo raggio parte dal punto ed e'
      //diretto verso l'osservatore
      Ray first = new Ray(iP, (cameraRay.d).multiplyScalar(-1));
      //si aggiunge alla variabile r il contributo
      //di radianza del punto considerato
      sceneRadiance =
          sceneRadiance.add(radiance(first, o, x, y));
    } else {
      //se non si interseca nessun oggetto si
      //aggiunge alla variabile r il colore di
      //background (nero)

      sceneRadiance = sceneRadiance.add(RenderAction.background);
    }
  }

  //divido per il numero di campioni del pixel
  sceneRadiance = sceneRadiance.divideScalar(RenderAction.samps);
  sceneRadiance.multiplyScalar(0.3f);
  // A questo punto si crea un'immagine basata sui
  //valori di radianza r
```

```
      //le componenti RGB del vettore r vengono tagliate
      //se non comprese in [0,1] dopodiche' vengono
      //caricate nel vettore image
      //nota: per ogni y che aumenta abbiamo gia'
      //caricato width pixel

      //utilizzo questa variabile tt perche' non posso
      //usare il valore x+y*width nell'array image[width*y],
      //altrimenti l'ultimo indice sarebbe fuori dal
      //range (ricordo che la misura e' width*height ma gli
      //indici vanno da 0 a width*height-1)
      int tt =x+y* RenderAction.width;
      //allora faccio l'if per tt<width*height cosi' da
      //accertarmi che non sia considerato l'indice
      //width*height-esimo
      if(tt< RenderAction.width * RenderAction.height) {
        RenderAction.image[x+y* RenderAction.width].x =
            Point3D.clamp(sceneRadiance.x);
        RenderAction.image[x+y* RenderAction.width].y =
            Point3D.clamp(sceneRadiance.y);
        RenderAction.image[x+y* RenderAction.width].z =
            Point3D.clamp(sceneRadiance.z);
      }
    }
  }
}
```

Classe PhotonMappingClass:

```
package renderer;

import partition.PhotonBox;
import primitive.Obj;
import primitive.Photon;
import primitive.Point3D;
import primitive.Ray;
import ui.InterfaceInitialiser;
import ui.RenderAction;

import java.util.ArrayList;

import static renderer.PhotonScatterClass.causticScatter;
import static renderer.PhotonScatterClass.photonScatter;

public class PhotonMappingClass {
```

```
/* Metodo con lo scopo di inizializzare le variabili di interesse, ovvero i
    partition.PhotonBox per creare la mappa
 *  e chiamare i metodi per sparare i fotoni
 */
public static void calculatePhotonMapping() {
  int liv = 0;

  // Calcolo del numero di partition.PhotonBox da creare
  for(int i = 1; i < RenderAction.kDepth; i++){
    RenderAction.power += Math.pow(2, i);
  }

  RenderAction.kdTree = new PhotonBox[RenderAction.power +1];
  RenderAction.causticTree = new PhotonBox[RenderAction.power +1];

  //vengono emessi i fotoni dalle luci e fatti rimbalzare all'interno della scena
  emitPhotons();

  if(RenderAction.causticPhoton > 0 && Utilities.checkRefractionObjects()) {
    caustic();
  }

  RenderAction.kdTree[0]
    = new PhotonBox(RenderAction.min, RenderAction.max, RenderAction.photons);
  RenderAction.causticTree[0]
    = new PhotonBox(RenderAction.min, RenderAction.max, RenderAction.caustics);

  balance(RenderAction.kdTree,1,liv);
  balance(RenderAction.causticTree,1,liv);
}

// Metodo che effette fotoni in direzioni casuali, campionando uniformemente un
    emisfero
static void emitPhotons() {
  Utilities utilities = new Utilities();

  float random1;
  float random2;
  float random3;
  Obj objX = null;

  for (int i = 0; i < RenderAction.lights.size(); i++) {
    //carichiamo l'area della luce
    double area = RenderAction.lights.get(i).areaObj;
    //carichiamo i dati relativi alla luce
    int lid = RenderAction.lights.get(i).matId;
    //calcoliamo la potenza trasportata dal singolo fotone (condivisa con gli
        altri nPhoton)
    Point3D P = RenderAction.material[lid].emittedLight
```

```
      .multiplyScalar(Utilities.
      MATH_PI*area/(RenderAction.nPhoton));

//per ogni fotone
for(int s = 0; s < RenderAction.nPhoton; s++) {
  //campionamo la luce uniformemente
  random1 = Utilities.generateRandom(s);
  random2 = Utilities.generateRandom(s);
  random3 = Utilities.generateRandom(s);

  // punto scelto uniformemente nella patch i;
  Point3D p = RenderAction.lights.get(i)
      .randomPoint(random1,random2,random3);

  //dichiariamo i parametri per distribuire uniformemente i campioni
     sull'emisfero:
  float rndPhi = 2* Utilities.MATH_PI*(random1);
  double rndTeta = Math.acos(Math.sqrt(random2));

  //creazione della base ortonormale
  // Create onb (ortho normal basis) on iP punto di intersezione
  Point3D u,v,w;
  w = RenderAction.lights.get(i).normal(p);
  //vettore up (simile a (0,1,0))
  Point3D up = new Point3D(0.0015f,1.0f,0.021f);
  v = w.crossProduct(up);
  v = v.getNormalizedPoint();
  u = v.crossProduct(w);

  Point3D dir;
  float cosPhi=(float) Math.cos(rndPhi);
  float sinTeta=(float) Math.sin(rndTeta);
  float sinPhi=(float) Math.sin(rndPhi);
  float cosTeta=(float) Math.cos(rndTeta);
  dir = (u.multiplyScalar(cosPhi*sinTeta))
      .add(v.multiplyScalar(sinPhi*sinTeta))
      .add(w.multiplyScalar(cosTeta));
  dir = dir.getNormalizedPoint();

  //creiamo il raggio dal punto di intersezione r.o al punto aleatorio scelto
     a caso sull'emisfero:
  Ray photonRay = new Ray(p,dir);
  double t= Utilities.inf;

  if(utilities.intersect(photonRay, objX)) {
    t = utilities.inters;
    utilities.inters = Utilities.inf;

    //punto di intersezione:
```

```
        Point3D iP = photonRay.o.add(photonRay.d.multiplyScalar(t));

        //creiamo il nuovo fotone e inserito nella photonMap
        Photon p2 = new Photon(iP, dir.multiplyScalar(-1), P);

        RenderAction.photons.add(p2);

        objX = utilities.intersObj;
        utilities.intersObj = null;

        photonScatter(objX, 0, p2);
      }
    }
  }
}


// Metodo per creare i fotoni indirizzati agli oggetti traslucenti/trasparenti
    chiamando il metodo causticScatter()
static void caustic() {
  Utilities utilities = new Utilities();
  float rnd1;
  float rnd2;
  float rnd3;
  Obj objX;

  int nP=0;
  while(nP < RenderAction.causticPhoton) { //per ogni campione
    InterfaceInitialiser.label.setText("Calcolo campioni caustiche: " + nP);

    rnd1
       = Utilities.generateRandom(RenderAction.loadedBoxes)
       *(RenderAction.lights.size());

    int l = (int) Math.floor(rnd1);
    l = l >= RenderAction.lights.size() ? 1 : l;

    double area= RenderAction.lights.get(l).areaObj;
    int lid= RenderAction.lights.get(l).matId;

    //mappa di proiezione per la luce
    ArrayList<Point3D> ProjectionMap;

    //carichiamo la potenza di ciascun fotone
    Point3D P
       = RenderAction.material[lid].emittedLight
    .multiplyScalar(Utilities.MATH_PI* RenderAction
       .scaleCausticPower*area/(RenderAction.causticPhoton
          * RenderAction.aoCausticPhoton));
```

```
//se la luce Ăš un triangolo
if(RenderAction.lights.get(l).t != null) {
  //campiona uniformemente
  rnd1= Utilities.generateRandom(RenderAction.loadedBoxes);
  rnd2= Utilities.generateRandom(RenderAction.loadedBoxes);
  rnd3= Utilities.generateRandom(RenderAction.loadedBoxes);

  //un punto del triangolo in coordinate convesse
  Point3D point = RenderAction.lights.get(l).randomPoint(rnd1,rnd2,rnd3);
  Point3D normal = RenderAction.lights.get(l).normal(new Point3D());

  //calcoliamo la mappa di proiezione e teniamo conto delle ricorrenze di
      oggetti trasparenti
  ProjectionMap = utilities.Projection(point,normal);

  //incremento
  float dTheta= Utilities.MATH_PI/(2* RenderAction.projectionResolution);
  float dPhi=(2* Utilities.MATH_PI)/ RenderAction.projectionResolution;

  //angolo solido di ciascuna patch dell'emisfero
  float SolidAngle=dTheta*dPhi;
  //numero di patch in cui Ăš presente un oggetto trasparente
  int nAngle = ProjectionMap.size();

  if(nAngle > 0) {
    //se Ăš stato trovato almeno un oggetto trasparente

    //creo la base ortonormale
    Point3D u,v,w;
    w = RenderAction.lights.get(l).normal(new Point3D());
    Point3D up = new Point3D(0.0015f,1.0f,0.021f);
    v = w.crossProduct(up);
    v = v.getNormalizedPoint();
    u = v.crossProduct(w);

    //scalo la potenza del fotone in base all'angolo solido
    float totSolidAngle=nAngle*SolidAngle;
    float scale= totSolidAngle/2* Utilities.MATH_PI;

    //potenza scalata
    Point3D P2 = P.multiplyScalar(scale);

    nP++;

    int nS=0;

    while(nS < RenderAction.aoCausticPhoton) {
      //campiono uniformemente la mappa di proiezione
```

```
    rnd1 = Utilities.generateRandom(RenderAction.loadedBoxes)*(nAngle);

    int floor = (int) Math.floor(rnd1);

    //prendo un angolo a caso dalla mappa di proiezione
    Point3D angle = ProjectionMap.get(floor >= 1 ? 0 : floor);

    //creo un raggio distribuito uniformemente all'interno della patch
    double rndPhi;
    double rndTheta;

    //campiona uniformemente
    rndPhi = Utilities.generateRandom(RenderAction.projectionResolution
       * RenderAction.projectionResolution)*dPhi;
    rndTheta = Utilities.generateRandom(RenderAction.projectionResolution
       * RenderAction.projectionResolution)
          *((angle.z+dTheta)*Math.sin(angle.z+dTheta)-angle.y);

    //angoli finali per la creazione del raggio
    double Phi= angle.x+rndPhi;
    double Theta=angle.y+rndTheta;
    //raggio
    Point3D dir = u.multiplyScalar(Math.cos(Phi)*Math.sin(Theta))
       .add(v.multiplyScalar(Math.sin(Phi)*Math.sin(Theta)))
.add(w.multiplyScalar(Math.cos(Theta)));
    Ray pRay= new Ray(point,dir);

    double t;
    objX = null;

    if(utilities.intersect(pRay, objX)) {
      objX = utilities.intersObj;
      t = utilities.inters;
      //si controlla se l'oggetto colpito Ăš trasparente
      if(RenderAction.material[objX.matId].refractionColor.max() > 0) {
        //creiamo il nuovo fotone e ne calcoliamo i rimbalzi nella scena
        Point3D iP=pRay.o.add(pRay.d.multiplyScalar(utilities.inters));
        Photon p2 = new Photon(iP,dir.multiplyScalar(-1),P2);

        utilities.intersObj = null;
        utilities.inters = Utilities.inf;

        causticScatter(objX, 0, p2);
      }
    }

    //aumento il numero di fotoni sparati per l'emisfero
    nS++;
  }
```

```
      }
    }
  }
}


// Metodo per raffinare la suddivisione dei partition.PhotonBox e ridividerli
    nell'albero tree
static void balance(PhotonBox[] tree, int index, int liv) {
  liv++;

  if(liv < RenderAction.kDepth) {
    int dim = tree[index-1].dim;
    double median= tree[index-1].planePos;
    double n = tree[index-1].ph.size();
    ArrayList<Photon> ph= tree[index-1].ph;

    Point3D min= tree[index-1].V[0];
    Point3D max= tree[index-1].V[1];

    ArrayList<Photon> ph1 = new ArrayList<>();
    ArrayList<Photon> ph2 = new ArrayList<>();

    switch (dim) {
      case 0:
        //taglio con il piano x=median a metÃ  del Bound

        for(int i=0; i<n; i++){
          if(ph.get(i).position.x < median) {
            ph1.add(ph.get(i));
          } else {
            ph2.add(ph.get(i));
          }
        }

        tree[(2*index)-1] = new PhotonBox(min, new
            Point3D(median,max.y,max.z),ph1);
        tree[(2*index)] = new PhotonBox(new Point3D(median,min.y,min.z), max,
            ph2);

        break;
      case 1:
        //taglio con il piano y=median a metÃ  del Bound

        for(int i=0; i<n; i++){
          if(ph.get(i).position.y < median){
            ph1.add(ph.get(i));
          } else {
            ph2.add(ph.get(i));
```

```
          }
        }

        tree[(2*index)-1] = new PhotonBox(min, new Point3D(max.x,median,max.z),
            ph1);
        tree[2*index] = new PhotonBox(new Point3D(min.x,median,min.z), max, ph2);

        break;
      case 2:
        //taglio con il piano z=median a metÃ  del bound

        for(int i=0; i<n; i++){
          if(ph.get(i).position.z < median){
            ph1.add(ph.get(i));
          } else {
            ph2.add(ph.get(i));
          }
        }

        tree[(2*index)-1] = new PhotonBox(min, new Point3D(max.x,max.y,median),
            ph1);
        tree[2*index] = new PhotonBox(new Point3D(min.x,min.y,median), max, ph2);
    }

    balance(tree,2*index,liv);
    balance(tree,(2*index)+1,liv);
  }
 }
}
```

Classe PhotonRadianceClass:

```
package renderer;

import partition.PhotonBox;
import primitive.Obj;
import primitive.Photon;
import primitive.Point3D;
import primitive.Ray;
import ui.RenderAction;

import java.util.Hashtable;
import java.util.Map;

public class PhotonRadianceClass {
  /* Metodo che gestisce l'energia uscente dalla superficie tramite il Photon
     Mapping
```

```
   */
  static Point3D photonRadiance(Ray r, Obj objX, PhotonBox[] Tree, double
      photonSearchDisc, int nph) {
    Utilities utilities = new Utilities();

    Point3D radianceOutput = new Point3D();

    //carico l'ID del materiale
    int matId= objX.matId;

    //carico la normale dell'oggetto
    Point3D n1= objX.normal(r.o);

    //fotoni trovati nelle vicinanze del punto in esame

    Hashtable<Double, Photon> nearPh = new Hashtable<>();

    utilities.locatePhotons(nearPh,r.o,1,objX,Tree, photonSearchDisc,nph);

    //per ogni fotone trovato
    for (Map.Entry<Double, Photon> entry : nearPh.entrySet()) {
      //raggio di entrata del fotone
      Ray psi= new Ray(r.o, entry.getValue().direction);

      //calcolo della BRDF
      Point3D BRDF = RenderAction.material[matId].C_T_BRDF(psi,r,n1);

      //distanza del fotone dal punto
      double dist = entry.getKey();

      double W=1-(Math.sqrt(dist)/((1.1)*Math.sqrt(photonSearchDisc)));
      //stima della radianza nel punto
      radianceOutput = radianceOutput.add(BRDF
         .multiplyComponents(entry.getValue().power)
         .multiplyScalar(Utilities.MATH_PI)
         .multiplyScalar(1/ photonSearchDisc)
            .multiplyScalar(W));
    }

    return radianceOutput;
  }
}
```

Classe PhotonScatterClass:

```
package renderer;
```

```
import primitive.Obj;
import primitive.Photon;
import primitive.Point3D;
import primitive.Ray;
import ui.RenderAction;

public class PhotonScatterClass {
  /* Metodo per il rimbalzo delle caustiche, controllando l'intersezione con la
     superficie e quindi col suo materiale
   *  per definirne le proprietĂ  della riflessione/rifrazione e con quale potenza
   */
  static void photonScatter(Obj obj, int n_, Photon p) {
    Utilities utilities = new Utilities();

    //carichiamo il numero massimo di rimbalzi previsti per un fotone
    float MAX = Utilities.MAX_DEPTH_PHOTON;
    //ci ricaviamo quindi il peso da utilizzare nella roulette russa
    double peso=(MAX-n_)/MAX;

    //carichiamo l'ID del materiale dell'oggetto colpito
    int mId = obj.matId;

    //dai coefficenti del materiale otteniamo le probabilitĂ  di riflessione ,
        diffusione , rifrazione del fotone
    double P_refl = RenderAction.material[mId].reflectionColor.average();
    double P_diff = RenderAction.material[mId].diffusionColor.average();
    double P_glass = RenderAction.material[mId].refractionColor.average();

    //ci assicuriamo che la somma di questi valori corrispondino ad una probabilitĂ 
    double Ptot = P_refl + P_diff + P_glass;

    if(Ptot > 1) {
      P_refl /= Ptot;
      P_diff /= Ptot;
      P_glass /= Ptot;
    }

    //aggiungiamo la probabilita' di assorbimento del fotone data dal parametro peso
    double[] P = new double[3];
    P[0] = P_refl*peso;
    P[1] = P[0] + P_diff*peso;
    P[2] = P[1] + P_glass*peso;

    //carichiamo la normale dell'oggetto in esame
    Point3D n = obj.normal(p.position);
    Obj objY = null;

    //raggio di entrata del fotone:
    Ray entryRay = new Ray(p.position, p.direction);
```

```
double rnd = Math.random();


//metodo della Roulette russa
//probabilitĂ  materiali riflettenti: il fotone viene riflesso perfettamente
if(rnd < P[0]) {
  Point3D refl = utilities.reflect(p.direction, n);
  Ray reflRay = new Ray(p.position, refl);

  double t2 = Utilities.inf;

  if(utilities.intersect(reflRay, objY)){
    n_++;
    t2 = utilities.inters;
    utilities.inters = Utilities.inf;

    Point3D iP = reflRay.o.add(reflRay.d.multiplyScalar(t2));
    Point3D oldP = p.power;
    double cos_i = p.direction.dotProduct(n);
    Point3D Fresn = RenderAction.material[mId].getFresnelCoefficient(cos_i);
    Point3D BRDF = RenderAction.material[mId].S_BRDF(Fresn);
    Point3D newP = oldP.multiplyComponents(BRDF)
        .multiplyScalar(1/(P_refl*peso));
    Photon p2 = new Photon(iP,refl.multiplyScalar(-1), newP);

    objY = utilities.intersObj;
    utilities.intersObj = null;

    photonScatter(objY, n_, p2);
  }
}

//probabilitĂ  materiali diffusivi: il fotone viene riflesso con probabilitĂ 
    uniforme sull'emisfero.
if((rnd<P[1])&&(rnd>P[0])) {
  double rndls;
  double rndlt;

  //s parte da 0 quindi per il numero del sample metto s+1 (parto stavolta dal
      numero random generato per ogni pixel aoSampleX e aoSamplesY)
  rndls = Math.random();
  rndlt = Math.random();

  //distribuiamo i numeri sull'emisfero
  double rndPhi = 2* Utilities.MATH_PI*(rndls);
  double rndTeta = Math.acos(Math.sqrt(rndlt));

  // Create onb (ortho normal basis) on iP punto di intersezione
```

```
    Point3D u,v,w;
    w = obj.normal(p.position);
    //vettore up (simile a (0,1,0))
    Point3D up = new Point3D(0.0015f,1.0f,0.021f);
    v = w.crossProduct(up);
    v = v.getNormalizedPoint();
    u = v.crossProduct(w);

    Point3D dir = u.multiplyScalar((Math.cos(rndPhi)*Math.sin(rndTeta)))
       .add(v.multiplyScalar(Math.sin(rndPhi)*Math.sin(rndTeta)))
       .add(w.multiplyScalar(Math.cos(rndTeta)));
    dir = dir.getNormalizedPoint();

    //creazione del raggio
    Ray reflRay = new Ray(p.position,dir);

    double t2 = Utilities.inf;

    if(utilities.intersect(reflRay, objY)){
      n_++;
      t2 = utilities.inters;
      utilities.inters = Utilities.inf;

      Point3D iP = reflRay.o.add(reflRay.d.multiplyScalar(t2));
      Point3D oldP = p.power;
      Point3D BRDF = RenderAction.material[mId].diffusionColor
          .add(RenderAction.material[mId].reflectionColor);
      Point3D newP= oldP.multiplyComponents((BRDF)
          .multiplyScalar(1/(P_diff*peso)));
      Photon p2 = new Photon(iP,dir.multiplyScalar(-1),newP);

      RenderAction.photons.add(p2);

      objY = utilities.intersObj;
      utilities.intersObj = null;

      photonScatter(objY,n_,p2);
    }
  }


  //probabilitĂ  materiali trasparenti: il fotone viene rifratto.
  if((rnd<P[2])&&(rnd>P[1])) {
    // primitive.Ray refraction based on normal
    //carico un array di 3 raggi corrispondenti alle 3 lunghezza d'onda di base
        RGB
    Ray[] refrRay = new Ray[3];

    for (int i = 0; i < 3; i++) {
```

```
  refrRay[i] = new Ray();
}

refrRay[0].o = p.position;
refrRay[1].o = p.position;
refrRay[2].o = p.position;
//i raggi vengono rifratti in base all'indice ior ovvero l'indice di
    rifrazione del materiale esterno diviso l'indice di rifrazione del
    materiale interno

refrRay = utilities.refract(refrRay, p.direction, n,
    RenderAction.material[mId].refractionIndexRGB);

//velocizzazione del calcolo
//se IOR Ăš uguale per tutte e tre le componenti uso un solo raggio rifratto
if((RenderAction.material[mId].refractionIndexRGB.x ==
    RenderAction.material[mId].refractionIndexRGB.y)
    &&(RenderAction.material[mId].refractionIndexRGB.x ==
        RenderAction.material[mId].refractionIndexRGB.z)){

  if(refrRay[0].depth!=0){
    double t2 = Utilities.inf;

    if(utilities.intersect(refrRay[0], objY)){
      n_++;
      t2 = utilities.inters;
      utilities.inters = Utilities.inf;

      Point3D iP = refrRay[0].o.add(refrRay[0].d.multiplyScalar(t2));
      Point3D oldP = p.power;
      double cos_i = p.direction.dotProduct(n);
      Point3D Fresn= RenderAction.material[mId].getFresnelCoefficient(cos_i);
      Point3D BRDF = RenderAction.material[mId].T_BRDF(Fresn);
      Point3D newP= oldP.multiplyComponents(BRDF)
         .multiplyScalar(1/(P_glass*peso));
      Photon p2= new Photon(iP,refrRay[0].d.multiplyScalar(-1),newP);

      objY = utilities.intersObj;
      utilities.intersObj = null;

      photonScatter(objY, n_, p2);
    }
  }
} else{ //altrimenti si deve utilizzare un raggio per ogni componente
  double t2 = Utilities.inf;
  Point3D oldP = p.power;
  n_++;

  if(refrRay[0].depth!=0){
```

```
      objY = null;

      if(utilities.intersect(refrRay[0], objY)){
        t2 = utilities.inters;
        utilities.inters = Utilities.inf;

        Point3D iP=refrRay[0].o.add(refrRay[0].d.multiplyScalar(t2));
        Point3D BRDF =
            RenderAction.material[mId].C_T_BRDF(entryRay,refrRay[0],n)
           .add(RenderAction.material[mId].refractionColor);
        Point3D newP = new Point3D(oldP.x*BRDF.x*(1/(3*P_glass*peso)),0,0);
        Photon p2 = new Photon(iP,refrRay[0].d.multiplyScalar(-1),newP);

        objY = utilities.intersObj;
        utilities.intersObj = null;

        photonScatter(objY,n_,p2);
      }
    }

    if(refrRay[1].depth!=0){

      t2 = Utilities.inf;
      objY = null;

      if(utilities.intersect(refrRay[1], objY)){
        t2 = utilities.inters;
        utilities.inters = Utilities.inf;

        Point3D iP=refrRay[1].o.add(refrRay[2].d.multiplyScalar(t2));
        Point3D BRDF =
            RenderAction.material[mId].C_T_BRDF(entryRay,refrRay[1],n)
           .add(RenderAction.material[mId].refractionColor);
        Point3D newP = new Point3D(0,oldP.y*BRDF.y*(1/(3*P_glass*peso)),0);
        Photon p2 = new Photon(iP,refrRay[1].d.multiplyScalar(-1),newP);

        objY = utilities.intersObj;
        utilities.intersObj = null;

        photonScatter(objY, n_, p2);
      }
    }

    if(refrRay[2].depth!=0) {
      t2 = Utilities.inf;
      objY = null;

      if(utilities.intersect(refrRay[2], objY)){
        t2 = utilities.inters;
```

```
          utilities.inters = Utilities.inf;

          Point3D iP=refrRay[2].o.add(refrRay[2].d.multiplyScalar(t2));
          Point3D BRDF =
              RenderAction.material[mId].C_T_BRDF(entryRay,refrRay[2],n)
             .add(RenderAction.material[mId].refractionColor);
          Point3D newP = new Point3D(0,0,oldP.z*BRDF.z*(1/(3*P_glass*peso)));
          Photon p2 = new Photon(iP,refrRay[2].d.multiplyScalar(-1),newP);

          objY = utilities.intersObj;
          utilities.intersObj = null;

          photonScatter(objY, n_, p2);
        }
      }
    }
  }
}

/* Metodo per il rimbalzo delle caustiche, controllando l'intersezione con la
    superficie e quindi col suo materiale
 *  per definirne le proprietÃ  della riflessione/rifrazione e con quale potenza
 */
static void causticScatter(Obj obj, int n_, Photon p) {
  Utilities utilities = new Utilities();

  int mId = obj.matId;

  if(RenderAction.material[mId].diffusionColor.max()>0){
    RenderAction.caustics.add(p);
  }

  n_++;
  if(n_< Utilities.MAX_DEPTH_CAUSTIC){
    Obj objY = null;

    if((RenderAction.material[mId].refractionColor.max()>0)
          &&(RenderAction.material[mId].absorptionCoefficient.max()==0)) {
      Point3D n = obj.normal(p.position);

      Ray entryRay = new Ray(p.position, p.direction);
      double cos_i= n.dotProduct(p.direction);
      Point3D Fresn= RenderAction.material[mId].getFresnelCoefficient(cos_i);

      //velocizzazione del calcolo
      //se IOR ĂŠ uguale per tutte e tre le componenti uso un solo raggio rifratto
      if((RenderAction.material[mId].refractionIndexRGB.x ==
          RenderAction.material[mId].refractionIndexRGB.y)
```

```
      &&(RenderAction.material[mId].refractionIndexRGB.x ==
          RenderAction.material[mId].refractionIndexRGB.z)){
    Point3D dir = utilities.refract(entryRay.d,n,
        RenderAction.material[mId].refractionIndexRGB.x);

    if(dir != null) {
      Ray refrRay = new Ray(entryRay.o, dir);

      double t2 = Utilities.inf;
      if(utilities.intersect(refrRay, objY)){
        objY = utilities.intersObj;
        t2 = utilities.inters;

        utilities.inters = Utilities.inf;
        utilities.intersObj = null;

        n_++;
        Point3D iP=refrRay.o.add(refrRay.d.multiplyScalar(t2));
        Point3D oldP = p.power;
        Point3D BRDF = RenderAction.material[mId].T_BRDF(Fresn);
        Point3D newP = oldP.multiplyComponents(BRDF);
        Photon p2= new Photon(iP,refrRay.d.multiplyScalar(-1),newP);

        causticScatter(objY, n_, p2);
      }
    }
  } else {
    //altrimenti si deve utilizzare un raggio per ogni componente
    // primitive.Ray refraction based on normal
    //carico un array di 3 raggi corrispondenti alle 3 lunghezza d'onda di
        base RGB

    Ray[] refrRay= new Ray[3];
    refrRay[0].o=p.position;
    refrRay[1].o=p.position;
    refrRay[2].o=p.position;

    //i raggi vengono rifratti in base all'indice ior ovvero l'indice di
        rifrazione del materiale esterno diviso l'indice di rifrazione del
        materiale interno
    refrRay = utilities.refract(refrRay, p.direction, n,
        RenderAction.material[mId].refractionIndexRGB);
    double t2 = Utilities.inf;
    Point3D oldP = p.power;
    n_++;
    Point3D BRDF = RenderAction.material[mId].T_BRDF(Fresn);

    if(refrRay[0].depth!=0){
      if(utilities.intersect(refrRay[0], objY)){
```

```
    objY = utilities.intersObj;
    t2 = utilities.inters;

    Point3D iP=refrRay[0].o.add(refrRay[0].d.multiplyScalar(t2));
    Point3D newP= new Point3D(oldP.x*BRDF.x,0,0);
    Photon p2 = new Photon(iP,refrRay[0].d.multiplyScalar(-1),newP);

    utilities.intersObj = null;
    utilities.inters = Utilities.inf;

    causticScatter(objY, n_, p2);
  }
}

if(refrRay[1].depth!=0){
  t2= Utilities.inf;
  objY = null;

  if(utilities.intersect(refrRay[1], objY)){
    objY = utilities.intersObj;
    t2 = utilities.inters;

    Point3D iP = refrRay[1].o.add(refrRay[2].d.multiplyScalar(t2));
    Point3D newP = new Point3D(0,oldP.y*BRDF.y,0);
    Photon p2=new Photon(iP,refrRay[1].d.multiplyScalar(-1),newP);

    utilities.intersObj = null;
    utilities.inters = Utilities.inf;

    causticScatter(objY, n_, p2);
  }
}

if(refrRay[2].depth!=0) {
  t2= Utilities.inf;
  objY = null;

  if(utilities.intersect(refrRay[2], objY)) {
    objY = utilities.intersObj;
    t2 = utilities.inters;

    Point3D iP = refrRay[2].o.add(refrRay[2].d.multiplyScalar(t2));
    Point3D newP = new Point3D(0,0,oldP.z*BRDF.z);
    Photon p2 = new Photon(iP,refrRay[2].d.multiplyScalar(-1),newP);


    utilities.intersObj = null;
    utilities.inters = Utilities.inf;
```

```
            causticScatter(objY, n_, p2);
          }
        }
      }
    }
  }
}
```

Classe RadianceClass:

```
package renderer;

import primitive.Material;
import primitive.Obj;
import primitive.Point3D;
import primitive.Ray;
import ui.RenderAction;

import static renderer.EmittedObjRadianceClass.emittedObjRadiance;
import static renderer.FinalGatheringClass.finalGathering;
import static renderer.PhotonRadianceClass.photonRadiance;

public class RadianceClass {
  //Metodo che serve a calcolare la radianza nel punto
  //dell'oggetto che stiamo considerando
  //r e' il raggio che parte dal punto dell'oggetto
  //intersecato (dal raggio della fotocamera considerato)
  //e arriva all'osservatore
  //o e' l'oggetto in cui ci troviamo (quello intersecato
  //dal raggio della fotocamera che stavamo considerando)
  //x indica la colonna in cui ci troviamo, y indica la
  //riga in cui ci troviamo
  // Questo metodo tiene conto del supersampling che applichiamo (ovvero un numero
     di sample per pixel > 1)
  static Point3D radiance(Ray r, Obj o, int x, int y) {
    Utilities utilities = new Utilities();
    //definisco e inizializzo a (0,0,0) il float3 per la
    //radianza riflessa
    Point3D radianceRefl=new Point3D();
    //definisco e inizializzo a (0,0,0) il float3 per la
    //radianza rifratta
    Point3D radianceRefr=new Point3D();

    //salvo in una variabile l'indice del materiale dell'
    //oggetto considerato
    int mId=o.matId;
```

```
//normale dell'oggetto nel punto osservato
Point3D n1;
n1= o.normal(r.o);

//coseno tra il raggio entrante e la normale
double cos_i= r.d.dotProduct(n1);

//fattore di Fresnel
Point3D Fresn = RenderAction.material[mId].getFresnelCoefficient(cos_i);

//si verifica se il materiale ha una componente speculare
//e che non sia stato superato il numero massimo di
//riflessioni
if((RenderAction.material[mId].reflectionColor.max() > 0) &&
    (RenderAction.nRay < RenderAction.maxDepth +1)) {
  //con questo controllo si evitano le riflessioni interne
  //al materiale
  if (cos_i>0) {
    // riflessione del raggio in entrata rispetto alla
    //normale n1
    Point3D refl= Point3D.reflect(r.d,n1);
    //si verifica che il materiale sia perfettamente
    //speculare o che non sia la prima riflessione (si evita
    //in questo modo l'aumento esponenziale dei raggi nel
    //caso in cui ci siano riflessioni multiple tra specchi
    //imperfetti)
    if((RenderAction.material[mId].refImperfection == 0) || (RenderAction.nRay
        > 0)) {
      //definisco e inizializzo a (0,0,0) il raggio riflesso
      Ray reflRay=new Ray();
      //l'origine e' la stessa del raggio passato come
      //parametro
      reflRay.o=r.o;
      //la direzione e' la riflessa di quella del raggio
      //passato come parametro
      reflRay.d=refl;
      //dichiaro e inizializzo la variabile t in cui
      //salveremo il punto di intersezione fra l'oggetto
      //considerato e reflRay
      double t= Utilities.inf;
      //definizione e inizializzoazione a null dell'oggetto
      //che si andra' ad intersecare
      Obj objX;
      objX=null;
      //intersezione del raggio con gli elementi della scena
      if(utilities.intersect(reflRay,objX)){
        //pongo t uguale al valore di intersezione
        //memorizzato nella variabile globale inters
        t= utilities.inters;
```

```
    //resetto inters uguale a inf in modo da avere
    //il giusto valore di partenza la prossima volta
    //che si utilizzera' il metodo intersect()
    utilities.inters = Utilities.inf;
    //salvo nell'elemento i-esimo dell'array objX
    //l'elemento intersecato dal raggio ffRay
    objX= utilities.intersObj;
    //resetto intersObj=null in modo da avere il
    //giusto valore di partenza la prossima volta che
    //si utilizzera' il metodo intersect()
    utilities.intersObj =null;
    //si calcola il punto di intersezione
    Point3D iP=(reflRay.o).add((reflRay.d).
        multiplyScalar(t));
    //si crea il nuovo raggio
    Ray r2=new Ray(iP,refl.multiplyScalar(-1));
    //si aumentano il numero di riflessioni per il
    //raytracing
    RenderAction.nRay++;
    //si calcola la radianza riflessa utilizzando
    //ricorsivamente la funzione radiance
    radianceRefl
        = radiance(r2, objX, x, y)
          .multiplyComponents(RenderAction.material[mId]
 .S_BRDF(Fresn));
    //si diminuisce il numero di riflessioni
    RenderAction.nRay--;
  }
}//fine if per specchi imperfetti
else {
  //si aumenta il numero di riflessioni una volta per
  //tutti i campioni
  RenderAction.nRay++;
  //per ogni campione
  for(int s = 0; s < RenderAction.refSamps; s++)
  {
    //dichiaro due variabili aleatorie
    float random1;
    float random2;

    //flag per la verifica dell'orientazione del
    //raggio
    boolean okdir=true;
    //direzione del nuovo raggio
    Point3D dir=new Point3D();
    //finche' non abbiamo un raggio con giusta
    //orientazione
    while(okdir){
      //creazione delle variabili aleatorie
```

```
        //uniformi usando come semi gli elementi dell'
        //array refSamples1
        random1= Utilities.generateRandom(RenderAction.refSamples1[x+y*
            RenderAction.width]);
        random2= Utilities.generateRandom(RenderAction.refSamples2[x+y*
            RenderAction.width]);

        //Inverse Cumulative Distribution Function
        //del coseno modificata
        //creiamo la base ortonormale per generare
        //la direzione del raggio riflesso reflRay
        float rndPhi=2* Utilities.MATH_PI *(random1);
        float rndTeta=(float) Math.acos((float) Math.pow(random2,
            RenderAction.material[mId].refImperfection));
        // creazione della base ortonormale rispetto
        //alla direzione di riflessione
        Point3D u,v,w;
        //inizializzo width uguale alla riflessione del
        //raggio in entrata
        w=refl;
        //vettore up (simile a (0,1,0))
        Point3D up=new Point3D(0.0015f,1.0f,0.021f);
        //il prodotto vettoriale tra width e up mi
        //genera il vettore v normale a entambi, che
        //normalizzo
        v=w.crossProduct(up);
        v=v.getNormalizedPoint();
        //il prodotto vettoriale tra v e width mi genera
        //il vettore u normale a entambi dal momento
        //che i vettori v e width sono gia' normali, non
        //c'e' bisogno di noemalizzare il vettoe u)
        u=v.crossProduct(w);
        //ora che abbiamo la base ortonormale,
        //possiamo calcolare la direzione dir
        //salvo in delle cariabili i calori di seno
        //e coseno necessari per il calcolo di dir
        float cosPhi=(float) Math.cos(rndPhi);
        float sinTeta=(float) Math.sin(rndTeta);
        float sinPhi=(float) Math.sin(rndPhi);
        float cosTeta=(float) Math.cos(rndTeta);
        //dir=(u*(cosPhi*sinTeta))+(v*(sinPhi*sinTeta)
        //)+(width*(cosTeta)) poi normalizzato
        dir=(u.multiplyScalar(cosPhi*sinTeta))
         .add(v.multiplyScalar(sinPhi*sinTeta))
    .add(w.multiplyScalar(cosTeta));

        //si verifica che la direzione formi un angolo
        //di massimo 90 gradi con la normale
        if(dir.dotProduct(n1)>0)
```

```
            okdir=false;
          //altrimenti si cerca una nuova direzione
        }//fine while(okdir)

        //creazione del raggio riflesso
        Ray reflRay=new Ray();
        reflRay.o=r.o;
        reflRay.d=dir;
        //massima distanza del raggio
        double t;
        Obj objX = null;
        //intersezione con gli oggetti della scena
        if(utilities.intersect(reflRay, objX)){
          t= utilities.inters;
          utilities.inters = Utilities.inf;
          objX= utilities.intersObj;
          utilities.intersObj =null;
          //punto di intersezione del raggio con
          //l'oggetto objX:
          Point3D iP=(reflRay.o).add((reflRay.d).multiplyScalar(t));
          //nuovo raggio:
          Ray r2=new Ray(iP,refl.multiplyScalar(-1));

          //calcolo della radianza riflessa
          radianceRefl
              =radianceRefl.add(radiance(r2, objX, x, y)
                 .multiplyComponents(RenderAction.material[mId]
        .S_BRDF(Fresn)));
        }
      }
      RenderAction.nRay--;
      //divido per il numero di raggi che sono stati creati
      radianceRefl=radianceRefl.divideScalar((float) RenderAction.refSamps);
    }
  }
}

//Rifrazione
//Si verifica che l'oggetto abbia un coefficiente di rifrazione >0, non sia un
    metallo
//  e non si siano superato il numero massimo di riflessioni del ray tracer
if((RenderAction.material[mId].refractionColor.max()>0)
    &&(RenderAction.nRay < Utilities.MAX_DEPTH +1)
    &&(RenderAction.material[mId].absorptionCoefficient.max()==0)) {
  //si verifica che l'indice di rifrazione sia uguale per
  //tutte le componenti RGB
  //in questo caso il calcolo per la rifrazione sara'
  //semplificato
```

```
if((RenderAction.material[mId].refractionIndexRGB.x ==
    RenderAction.material[mId].refractionIndexRGB.y)
    && (RenderAction.material[mId].refractionIndexRGB.x ==
        RenderAction.material[mId].refractionIndexRGB.z))
{
  //direzione del raggio rifratto:
  //calcolo della direzione per il raggio rifratto:
  Point3D dir = Point3D.getRefraction(r.d, n1,
      RenderAction.material[mId].refractionIndexRGB.x);
  //se c'e' effettivamente una rifrazione non interna
  if(!dir.equals(new Point3D(-1.0f))) {

    //viene creato il raggio rigratto
    Ray refrRay=new Ray(r.o,dir);

    //si verifica il parametro di imperfezione del
    //materiale
    if((RenderAction.material[mId].refImperfection==0)
      ||(RenderAction.nRay>0))
    {
      double t= Utilities.inf;
      Obj objX;
      objX=null;

      if(utilities.intersect(refrRay, objX))
      {
        t= utilities.inters;
        utilities.inters = Utilities.inf;
        objX= utilities.intersObj;
        utilities.intersObj =null;
        Point3D iP=(refrRay.o).add(
            refrRay.d.multiplyScalar(t));
        Ray r2=new Ray(iP,dir.multiplyScalar(-1));

        RenderAction.nRay++;
        //calcolo della radianza rifratta
        radianceRefr = radiance(r2,objX,x,y)
            .multiplyComponents(RenderAction.material[mId]
      .T_BRDF(Fresn));
        RenderAction.nRay--;
      }
    } else {
      RenderAction.nRay++;
      //per ogni sample
      for(int s = 0; s < RenderAction.refSamps; s++){
        float random1;
        float random2;
        //flag di controllo sulla direzione creata
        boolean okdir=true;
```

```
//float3 dir;

while(okdir){
  //variabili aleatorie uniformi in
  //[0,1]
  random1= Utilities.generateRandom(RenderAction.refSamples1[x+y*
      RenderAction.width]);
  random2= Utilities.generateRandom(RenderAction.refSamples1[x+y*
      RenderAction.width]);

  //distribuisco i numeri random sull'
  //emisfero
  float rndPhi=2* Utilities.MATH_PI *(random1);
  float rndTeta=(float) Math.acos((float) Math.pow(random2,
      RenderAction.material[mId].refImperfection));

  // creazione della base ortonormale
  //creata a partire dal raggio rifratto
  Point3D u,v,w;
  w=refrRay.d;
  //vettore up (simile a (0,1,0))
  Point3D up=new Point3D(0.0015f,1.0f,0.021f);
  v=w.crossProduct(up);
  v=v.getNormalizedPoint();
  u=v.crossProduct(w);
  //si calcola la direzione del raggio

  float cosPhi=(float) Math.cos(rndPhi);
  float sinTeta=(float) Math.sin(rndTeta);
  float sinPhi=(float) Math.sin(rndPhi);
  float cosTeta=(float) Math.cos(rndTeta);
  dir=(u.multiplyScalar(cosPhi*sinTeta))
      .add(v.multiplyScalar(sinPhi*sinTeta))
      .add(w.multiplyScalar(cosTeta));
  //si verifica che l'angolo tra la
  //normale e la nuova direzione sia
  //maggiore di 90 gradi
  if(dir.dotProduct(n1.multiplyScalar(-1))>0)
    okdir=false;
}//fine while(okdir)

//nuovo raggio:
Ray rRay=new Ray();
rRay.o=r.o;
rRay.d=dir;

double t;
Obj objX = null;
```

```
        if(utilities.intersect(refrRay, objX)){
          t= utilities.inters;
          utilities.inters = Utilities.inf;
          objX= utilities.intersObj;
          utilities.intersObj =null;
          //punto di intersezione
          Point3D iP=(refrRay.o).add((refrRay.d).multiplyScalar(t));
          Ray r2=new Ray(iP,dir.multiplyScalar(-1));

          //calcolo della radianca rifratta
          radianceRefr=radianceRefr.add(radiance(r2,objX,x,y)
               .multiplyComponents(RenderAction.material[mId]
      .T_BRDF(Fresn)));
          }
        }
        RenderAction.nRay--;
        //si mediano i contributi di tutti i raggi
        //usati
        radianceRefr=radianceRefr.divideScalar(RenderAction.refSamps);
      }
    }
  } else {
    //se l'indice di rifrazione e' diverso nelle 3
    //componeneti RGB allora si devono calcolare 3 raggi
    //uno per ogni componente. In questo caso pero' per
    //facilitare il calcolo non viene considerato l'indice
    //di imperfezione del materiale

    // rifrazone del raggio basata sulla normale
    // Raggio utilizzato per la rifrazione
    Ray[] refrRay= new Ray[3];
    refrRay[0]=new Ray();
    refrRay[1]=new Ray();
    refrRay[2]=new Ray();
    refrRay[0].o=r.o;
    refrRay[1].o=r.o;
    refrRay[2].o=r.o;
    float[] K = {0, 0, 0};

    //calcolo dei 3 raggi rifratti
    refrRay = Point3D.getRefraction(refrRay,r.d,n1, RenderAction.material[mId].
        refractionIndexRGB);

    //carichiamo la brdf sul vettore g[] di 3 elementi
    //cosi da accedervi piu' facilmente
    Point3D brdf= RenderAction.material[mId].T_BRDF(Fresn);
    double[] g = {brdf.x, brdf.y, brdf.z};
    RenderAction.nRay++;
    //per ogni componente RGB
```

```
      for(int i=0;i<3;i++) {
        //si verifica che non ci sia stata riflessione
        //totale
        if(refrRay[i].depth!=0){
          double t= Utilities.inf;
          Obj objX;
          objX=null;

          if(utilities.intersect(refrRay[i], objX)){
            t= utilities.inters;
            utilities.inters = Utilities.inf;
            objX= utilities.intersObj;
            utilities.intersObj =null;
            RenderAction.nRay++;
            Point3D iP=(r.o).add((refrRay[i].d).multiplyScalar(t));
            Ray r2=new Ray(iP,refrRay[i].d.multiplyScalar(-1));

            //viene calcolato il valore di radianza
            //del raggio
            Point3D pr= radiance(r2,objX,x,y);
            //si deve ora estrapolare la componente
            //i di tale radianza
            double[] rg=new double[3];
            rg[0]=pr.x;
            rg[1]=pr.y;
            rg[2]=pr.z;
            //si moltiplica infine per la componente
            //della brdf corrispondente
            K[i]+=rg[i]*g[i];
          }
        }
      }

      radianceRefr=new Point3D(K[0],K[1],K[2]);
      RenderAction.nRay--;
    }
  }

  //algoritmi per il calcolo dell'illuminazione globale:
  //si controlla che il materiale abbia componente Kd>0 in
  //almeno una delle 3 componenti o che il coefficiente
  //slope per la rugosita' della superficie sia >0 o che
  //il materiale emetta luce
  if((RenderAction.material[mId].diffusionColor.max()>0)
       ||(Material.slope >0)||
      (RenderAction.material[mId].emittedLight.max()>0)){
    //photon mapping:
    //if we render with photon mapping
    if(RenderAction.doPhotonMapping) {
```

```
        return photonRadiance(r, o, RenderAction.kdTree,
            RenderAction.photonSearchDisc, RenderAction.nPhotonSearch)
            .add(radianceRefr).add(radianceRefl)
            .add(photonRadiance(r, o, RenderAction.causticTree,
               RenderAction.causticSearchDisc, RenderAction.nCausticSearch))
            .add(emittedObjRadiance(o));
      }

      //metodo di Final gathering:
      if (RenderAction.doFinalGathering) {
        Point3D f= finalGathering(r, x, y, o);
        return f.add(radianceRefr).add(radianceRefl).add(emittedObjRadiance(o));
      }

      //metodo di Jacobi stocastico:
      if(RenderAction.doJacobi){
        double areaInverse= (1.0f)/((o).areaObj);
        Point3D L=((o).P).multiplyScalar(areaInverse);

        //si somma alla radianza emessa dalla patch
        //(pesata per l'area), la radianza data dalla
        //riflessione e dalla rifrazione, quindi si
        //restituisce il valore di radianza calcolato in
        //questo modo

        return L.add(radianceRefr).add(radianceRefl);
      }

      //se non e' stato impostato nessuno di tali metodi
      //allora viene restituito il colore nero
      return new Point3D();

    } else {
      //se il materiale e' riflettente o trasparente
      return radianceRefl.add(radianceRefr);
    }
  }
}
```

Classe Renderer:

```
package renderer;

import partition.PhotonBox;
import primitive.*;
import ui.InterfaceInitialiser;
import ui.RenderAction;
```

```
import java.util.ArrayList;

/* La classe Render contiene i metodi di rendering adottati nel programma
 *  e tutti quelli alle loro dipendenze.
 * La classe ĂŠ gestita con variabili non statiche e come oggetti
 *  nel programma per far sĂŹ di non far interferire i calcoli per un pixel
 *  con altri nel multithreading
 */

public class Renderer {
  /* Metodo per l'impostazione della multithreader radiance (qualsiasi metodo esso
      sia poichĂŠ il calcolo
   *  ĂŠ basato sulla selezione dei pixel, quindi parallelizzabile
   */
  public static void calculateThreadedRadiance(Camera cam) {
    new Runner(cam);
  }
}
```

Classe Runner:

```
package renderer;

import primitive.Camera;
import ui.RenderAction;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/* La classe renderer.Runner ĂŠ adibita alla gestione
 *  del multithreading del rendering
 * Il rendering viene suddiviso per le righe dell'immagine,
 *  creando un Runnable per ogni riga, nel quale per ogn
 *  pixel della riga avviene il rendering.
 * Il tutto ĂŠ gestito tramite le pools, dalle librerie di Java
 * Creando i threads, si fa avviare ogni thread con .execute()
 *  e si attende il termine del rendering con
 *  .awaitTermination().
 * .newWorkStealingPool() crea i thread in base ai threads
 *  disponibili determinati dal sistema operativo
 */

class Runner {
  private Camera cam;
```

```
  Runner(Camera cam) {
    this.cam = cam;

    execute();
  }

  private void execute() {
    ExecutorService pool = Executors.newWorkStealingPool();

    for (int y = 0; y < RenderAction.height; y++) {
      pool.execute(
          new ParallelProcessRadiance(y, cam));
    }

    pool.shutdown();

    try {
      pool.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Classe Utilities:

```
package renderer;

import partition.Box;
import partition.PhotonBox;
import primitive.Obj;
import primitive.Photon;
import primitive.Point3D;
import primitive.Ray;
import ui.RenderAction;

import java.util.ArrayList;
import java.util.Map;

/* La classe renderer.Utilities contiene le variabili giÃ
 *  precalcolate e i metodi necessari per le intersezioni
 * La classe Ãš gestita come un oggetto e con variabili non
 *  statiche poichÃš Ãš necessario gestire tutte le intersezioni
 *  in modo interno all'oggetto durante il multithreading
 *  per evitare accessi non autorizzati ad altre variabili.
 *
 * Il principale problema presente Ãš l'utilizzo di questa
```

```
 *  classe come oggetto obbligatorio per la classe Renderer
 *  per far si che, durante il multithreading, non si generino
 *  errori nel richiamo delle variabili inters e intersObj.
 * Sarebbe possibile riscrivere la classe in modo tale che
 *  restituisca ogni volta che necessario un vettore con la
 *  distanza e l'oggetto, in modo tale da non salvare variabili
 *  in questa classe e utilizzarla solo per gli strumenti utili
 *  al rendering
 */


public class Utilities {
   //DEFINIZIONE DI VARIABILI

   public static final float EPSILON = 1.e-2f;
   public static final float MATH_PI = 3.141592653589793238 46f;
   //1 su pi greco
   public static final float MATH_1_DIV_PI = 0.318309886183790671538f;
   static final float MATH_1_DIV_180 = 0.005555555690079927f;

   //massima ricorsivita' del ray tracing
   static int MAX_DEPTH=100;

   //massima ricorsivita' del photon mapping
   static int MAX_DEPTH_PHOTON=100000;

   //massima ricorsivita' del ray tracing
   static int MAX_DEPTH_CAUSTIC=100;


  //variabile globale, utilizzata nel metodo intersect(),
  //in cui viene salvato l'oggetto intersecato da un
  //raggio considerato
  Obj intersObj = null;
  //variabile globale, utilizzata nel metodo intersect(),
  //in cui viene salvato punto di intersezione tra l'oggetto
  //e il raggio considerato
  static double inf = 1e20; //distanza massima dal raggio
  double inters = inf;//sarebbe t del metodo intersect;

  //Intersezione con i box della ripartizione spaziale.
  //Controllando in modo ricorsivo, cerso l'intersezione
  //piu' vicina nei 2 figli di ogni box che stiamo
  //considerando, fino a trovare il punto di intersezione
  //b: box da intersecare con il raggio,
  //r: raggio considerato
  private boolean intersectBSP(Box b, Ray r){
    if(b.intersect(r)) {
      //controllo che i figli non siano nulli (ovvero che abbia figli)
```

```
    if((b.leaf1 != null) && (b.leaf2 != null)) {
      //se non e' nullo ricomincia con i figli
      intersectBSP(b.leaf1,r);
      intersectBSP(b.leaf2,r);
    } else { //altrimenti se non ha figli:
      //vengono salvati nell'array objects tutti
      //gli oggetti contenuti nel box che stiamo
      //considerando
      ArrayList<Obj> objects= b.objects;

      //inizializziamo il punto di intersezione
      //con il raggio
      double d=0;

      for (Obj object : objects) { //per ogni oggetto
        //nell'if si richiama il metodo intersect()
        //dell'oggetto in questione: ricordiamo che
        //questo metodo restituisce il valore della
        //distanza o -1.0f se non c'e' intersezione,
        //quindi nella prima condizione dell'if
        //controlliamo che effetticamente ci sia un'
        //intersezione e assegnamo questo valore a d.
        //nella seconda condizione controlliamo pero'
        //che questo valore di d appena calcolato
        //sia < del valore gia' memorizzato nella
        //variabile inters (dal momento che dobbiamo
        //salvare l'intersezione piu' vicina)
        if ((d = object.intersect(r)) >= 0.0f && d < inters) {
          //salviamo quindi il valore di d nella
          //variabile globale inters
          inters = d;
          //e salviamo l'oggetto intersecato nella
          //variabile globale intersObj
          intersObj = object;
        }
      }
    }
  } else { //altrimenti se non si interseca il box
    return false;
  }

    //dobbiamo anche controllare che il valore inters sia
    //rimasto minore della distanza massima del raggio inf
  //altrimenti l'intersezione e' troppo lontana quindi
  //non la consideriamo
  return inters < inf;
}

//metodo che serve a calcolare l'intersezione tra un
```

```
//oggetto e un raggio, che richiama la funzione
//inersectBSP
boolean intersect(Ray r, Obj oi) {
  if((oi)==null) {
      //se l'oggetto e' nullo si richiama semplicemente
      //il metodo intersectBSP
      return intersectBSP(RenderAction.bound,r);
  } else {
    //se l'oggetto non e' nullo, si salva in un'altra
    //variabile o1 e poi si rende nullo, per poter
    //richiamare il metodo intersectBSP
    Obj o1=oi;
    oi=null;

    if(intersectBSP(RenderAction.bound,r)) {
      oi = intersObj;

      return (o1) == (oi);
    } else {
      return false;
    }
  }
}

// riflessione di un vettore i rispetto alla normale n
Point3D reflect(Point3D i, Point3D n) {
  return (n.multiplyScalar(2))
      .multiplyScalar(n.dotProduct(i)).subtract(i);
}

// rifrazione di un vettore i rispetto ad una normale n
// in questa funzione la rifrazione non varia con la lunghezza d'onda
Point3D refract(Point3D i, Point3D n, double ior) {

  //ci si accerta che il vettore in entrata sia normalizzato
  i = i.getNormalizedPoint();

  //si calcola il coseno tra la normale e il vettore entrante i
  // <n,i>
  double cos_theta_i = n.dotProduct(i);

  //Si prende in esame l'indice di rifrazione ior del materiale;
  //per semplificare il calcolo viene considerato solamente
  //il passaggio dal vuoto ad un materiale.
  //Non Ăš quindi possibile con questa funzione modellare il passaggio di luce
  //tra due materiali con indici di rifrazione differenti.
  //Indice di rifrazione nel vuoto / indice di rifrazione del materiale:
  double eta = 1/ior;
```

```
  //se il coseno dell'angolo tra il vettore i e n Ăš minore di 0 allora
  //dobbiamo invertire la normale e l'indice di rifrazione
  //ovvero il passaggio avverrĂ  dal mezzo denso fino al vuoto
  // (di conseguenza il coseno diventerĂ  positivo)
  if(cos_theta_i<0){
    cos_theta_i=-cos_theta_i;
    eta = 1.0/eta;
    n = n.multiplyScalar(-1);
  }

  //calcolo dei fattori necessari
  double sin2_theta_i=1-(cos_theta_i*cos_theta_i);
  double sin2_theta_t= eta*eta*sin2_theta_i;

  // se questo coefficiente Ăš minore di 0 allora avviene una riflessione totale
      e il resto del calcolo non viene effettuato
  double K= 1-sin2_theta_t;
  if(K<0){
    //riflessione totale
    return null;
  } else {
    //altrimenti si procede con il calcolo del vettore rifratto
    double cos_theta_t = Math.sqrt(K);
    return n.multiplyScalar(eta*cos_theta_i-cos_theta_t)
          .subtract(i.multiplyScalar(eta));
  }
}

/// Refract a vector i on normal n based on external ior
Ray[] refract(Ray[] rays, Point3D i, Point3D n, Point3D ior) {
  Ray[] t = rays;

  //utilizziamo la proprietĂ  maxDepth del raggio per verificare se c'Ăš
      riflessione totale
  t[0].depth=1;
  t[1].depth=1;
  t[2].depth=1;

  i = i.getNormalizedPoint();
  // <n, i>
  double cos_theta_i=n.dotProduct(i);
  Point3D eta = new Point3D(1).divideComponents(ior);

  //se l'angolo di incidenza Ăš superiore a pi/2
  // allora devo cambiare il verso della normale e cambiare indice di rifrazione
      (prendendo il suo reciproco)
  if(cos_theta_i<0.0f){
    cos_theta_i=-cos_theta_i;
    n = n.multiplyScalar(-1.0f);
```

```
      eta = new Point3D(1).divideComponents(eta);
    }

    double sin2_theta_i=1-(cos_theta_i*cos_theta_i);

    Point3D sin2_theta_t= eta.multiplyComponents(eta).multiplyScalar(sin2_theta_i);

    Point3D K= new Point3D(1).subtract(sin2_theta_t);

    //componente rossa
    if (K.x<0.0f) {
      // TIR
      //radice negativa niente rifrazione
      t[0].depth=0;
    } else {
      double cos_theta_t= Math.sqrt(K.x);
      t[0].d = n.multiplyScalar(eta.x*cos_theta_i-cos_theta_t)
            .subtract(i.multiplyScalar(eta.x));
    }

    //componente verde
    if (K.y<0.0f) {
      // TIR
      //radice negativa niente rifrazione
      t[1].depth=0;
    } else {
      double cos_theta_t= Math.sqrt(K.y);
      t[1].d= n.multiplyScalar(eta.y*cos_theta_i-cos_theta_t)
          .subtract(i.multiplyScalar(eta.y));
    }

    //componente blu
    if (K.z<0.0f){
      // TIR
      //radice negativa niente rifrazione
      t[2].depth=0;
    } else {
      double cos_theta_t= Math.sqrt(K.z);
      t[2].d= n.multiplyScalar(eta.z*cos_theta_i-cos_theta_t)
          .subtract(i.multiplyScalar(eta.z));
    }

    return t;
  }

  ArrayList<Point3D> Projection(Point3D p,Point3D n){
    ArrayList<Point3D> ProjectionMap = new ArrayList<>();

    Point3D u,v,w;
```

```
w = n;
Point3D up = new Point3D(0.0015f,1.0f,0.021f);
v = w.crossProduct(up);
v = v.getNormalizedPoint();
u = v.crossProduct(w);

//incremento
float dTheta = (MATH_PI/2)/(RenderAction.projectionResolution);
float dPhi=(2*MATH_PI)/ RenderAction.projectionResolution;

//latitudine
double Theta=0;
double T=0;

Obj objY = null;

for(int i = 0; i < RenderAction.projectionResolution; i++) {
  //longitudine
  float Phi=0;

  for(int j = 0; j < RenderAction.projectionResolution; j++) {

    //direzione corrispondente all'angolo in esame
    Point3D dir = u.multiplyScalar((Math.cos(Phi)*Math.sin(Theta)))
        .add(v.multiplyScalar(Math.sin(Phi)*Math.sin(Theta)))
        .add(w.multiplyScalar((Math.cos(Theta))));

    Ray pRay= new Ray(p,dir);

    if(intersect(pRay, objY)){
      objY = intersObj;

      intersObj = null;
      inters = inf;

      //verifico la presenza di un materiale trasparente
      if(RenderAction.material[objY.matId].refractionColor.max() > 0) {
        Point3D angle = new Point3D(Phi,Theta,T);
        ProjectionMap.add(angle);
      }
    }

    Phi+=dPhi;
  }

  T+=dTheta;
  Theta=T*Math.sin(T);
}
```

```
    return ProjectionMap;
  }

  void locatePhotons(Map<Double, Photon> nearPh, Point3D iP, int index,
                     Obj objX, PhotonBox[] Tree, double d_2, int nph) {
//si verifica che il box index non sia vuoto
    if(Tree[index-1].nph != 0){

      //si verifica che il nuovo indice non abbia ha superato
      // la lunghezza dell'albero (power),
      // in tal caso l'indice corrisponde ad un box all'estremitÃ  dell'albero
      if((2*index)+1 < RenderAction.power){

        //viene caricato il punto di intersezione in un array di 3 elementi
        double[] pos = {iP.x, iP.y, iP.z};

        //si calcola la distanza del punto dal piano del nodo in esame
        double delta = pos[Tree[index-1].dim]-Tree[index-1].planePos;

        //a seconda della risposta si continua la ricerca
        // nella foglia sinistra o nella foglia destra
        if(delta < 0) {
          //foglia sinistra:
          //si continua la ricerca solo se il box ha dei fotoni all'interno
          locatePhotons(nearPh,iP,2*index,objX,Tree,d_2,nph);

          //viene verificato se la distanza dal piano Ãš piÃč piccola della
              distanza di ricerca
          if(Math.pow(delta,2)<d_2){

            //se lo Ãš si deve cercare anche nella foglia destra
            locatePhotons(nearPh,iP,2*index+1,objX,Tree,d_2,nph);
          }
        }else{

          //foglia destra (stesso procedimento invertito della foglia sinistra)
          locatePhotons(nearPh,iP,2*index+1,objX,Tree,d_2,nph);

          if(Math.pow(delta,2)<d_2) {
            locatePhotons(nearPh,iP,2*index,objX,Tree,d_2,nph);
          }
        }
      } else {
        //se ci si trova all'estremitÃ  dell'albero
        //si caricano tutti i fotoni del box
        int n = Tree[index-1].nph;
        ArrayList<Photon> p = Tree[index-1].ph;

        //per ogni fotone
```

```
for(int i=0;i<n;i++){

  //si verifica la distanza del fotone dal punto di intersezione
  Point3D dist = p.get(i).position.subtract(iP);


  if(objX.t != null){
    //se l'oggetto Ăš un triangolo la ricerca sarĂ  planare
    //in realtĂ  per evitare fenomeni di aliasing su
    // le facce piccole si Ăš costretti a considerare un parametro che
    // ci permetta di effettuare la ricerca sul disco o su una sfera
    //questo parametro Ăš definito come sphericalSearch
    //piĂč questo parametro Ăš grande piĂč

    //proiezione del vettore dist sul piano tangente ad objX

    Point3D d= dist.multiplyScalar(1/dist.normalize());
    double projN = d.dotProduct(objX.normal(iP));
    double val;

    if(objX.area() < 1){
      val= 1/(RenderAction.sphericalSearch*objX.area());
    }else{
      val= 1/(RenderAction.sphericalSearch);
    }

    if(val < Utilities.EPSILON)val=0;


    if((projN<=val) && (projN >= -val)) {

      double d2p = dist.squareNorm();
      //verifico la distanza
      if(d2p<d_2){

        nearPh.put(d2p, p.get(i));

        if(nearPh.size() > nph){
          Double value = (double) 0;
          for (Map.Entry<Double, Photon> entry : nearPh.entrySet()) {
            value = entry.getKey();
          }

          d_2 = value;
        }
      }
    }
  } else if (objX.s != null){
```

```
          //se l'oggetto Ăš una sfera la ricerca sarĂ  sferica
          //verifico la distanza
          double d2p=dist.squareNorm();
          //verifico la distanza
          if(d2p<d_2){
            nearPh.put(d2p, p.get(i));

            if(nearPh.size() > nph){
              Double value = (double) 0;
              for (Map.Entry<Double, Photon> entry : nearPh.entrySet()) {
                value = entry.getKey();
              }

              d_2 = value;
            }
          }
        }
      }
    }
  }

  static boolean checkRefractionObjects() {
    boolean isThereRefraction = false;

    for (Obj o : RenderAction.globalObjects) {
      if (RenderAction.material[o.matId].refractionColor.max() > 0) {
        isThereRefraction = true;
        break;
      }
    }

    return isThereRefraction;
  }

  //metodo che genera randomicamente un valore in [0,1]
  //utilizzando x come seme
  static float generateRandom(int x) {
    double a = Math.floor(Math.random()*(x+1));
    return (float)(a/x); //normalizzazione per riportare il valore in [0,1]
  }

  // TRASFORMAZIONI MATEMATICHE

   //trasformazione da gradi a radianti
   public float degreesToRadiants(float deg) {
      return deg* MATH_PI * MATH_1_DIV_180;
   }
```

```
   //trasformazione da radianti a gradi
   public float radiantsToDegrees(float rad) {
      return rad*180.0f* MATH_1_DIV_PI;
   }

   //troncamento all'intervallo [0,1]
   static double clamp(double x){
      return x<0 ? 0 : x>1 ? 1 : x;
   }

   //Transforma un float in [0,1], in un intero in [0,255]
   //Si utilizza con una gamma di 2.2, ovvero il numero
   //viene portato a [0,1] dopodiche' viene elevato alla
   //2.2: in questo modo il valore 0,218 si avvicina allo
   //0,5 ovvero viene portato a meta' della gamma dando
   //piu' spazio ai colori scuri, i quali creano sempre
   //piu' problemmi. Moltiplicando infine per 255 si
   //estende per il range di bit preso in esame e la
   //funzione ritorna l'approssimazione ad intero
   public static int toInt(double value){
      return (int) (Math.pow(clamp(value), 1/2.2)*255);
   }
}
```

Classe EditPanel:

```
package ui;

import primitive.ModelerProperties;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/* Classe di inizializzazione della UI.
 * Ci sono diversi problemi durante l'utilizzo dei pannelli,
 *  i quali, con un certo numero di utilizzi, rendono il
 *  programma instabile.
 *
 */

public class EditPanel extends JPanel {
  private JButton renderButton;

  private JPanel centralPanel;
  private JPanel bottomPanel;
```

```
  private JTabbedPane tabs = null;
  private JComboBox<String> materials;
  private JComboBox<String> positions;

  // Sample per il rendering dell'immagine
  private JSpinner samples;

  JacobiPanel jacobiPanel;
  FinalGatheringPanel finalGatheringPanel;
  PhotonPanel photonPanel;

  /* ui.EditPanel ĂŠ un pannello che contiene
   *  un JTabbedPane in cui sono presenti le 3 tecniche di rendering
   *  implementate. Per ognuna ĂŠ presente il suo pannello
   *  con i suoi parametri. RenderAction, con accesso statico,
   *  controllerĂ  quale tab ĂŠ scelto e quali valori dai vari
   *  JSpinner (e successivamente JSlider) richiedere per impostare il rendering
      effettivo
   */

  public EditPanel() {
    super();
    setLayout(new BorderLayout());

    bottomPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    bottomPanel.add(new JLabel("Materiali: "));
    bottomPanel.add(new JSeparator(SwingConstants.VERTICAL));
    bottomPanel.add(materials = new JComboBox<>(Properties.MATERIALS));
    bottomPanel.add(new JSeparator(SwingConstants.VERTICAL));
    bottomPanel.add(new JLabel("Posizione sfere: "));
    bottomPanel.add(new JSeparator(SwingConstants.VERTICAL));
    bottomPanel.add(positions = new JComboBox<>(Properties.POSITIONS));
    bottomPanel.add(new JSeparator(SwingConstants.VERTICAL));
    bottomPanel.add(renderButton = new JButton("Render!"));

    initialise();

    renderButton.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        new Thread(new Runnable() {
          @Override
          public void run() {
            new RenderAction(ModelerProperties.ENABLE_MODELER);
          }
        }).start();
      }
    });
```

```
  add(centralPanel);
  add(bottomPanel, BorderLayout.PAGE_END);
}

private void initialise() {
  centralPanel = new JPanel(new BorderLayout());

  tabs = new JTabbedPane();

  tabs.add(jacobiPanel = new JacobiPanel(), "Jacobi");
  tabs.add(finalGatheringPanel = new FinalGatheringPanel(), "Final Gathering");
  tabs.add(photonPanel = new PhotonPanel(), "Photon Mapping");

  centralPanel.add(tabs);

  samples = new JSpinner(new SpinnerNumberModel(50, 1, 1000, 1));

  JPanel panel = new JPanel();
  panel.add(new JLabel("Campioni per pixel (raggi)"));
  panel.add(samples);
  centralPanel.add(panel, BorderLayout.PAGE_END);
}

int getMethod() {
  return tabs.getSelectedIndex();
}

String getMaterial() {
  return materials.getItemAt(materials.getSelectedIndex());
}

String getPosition() {
  return positions.getItemAt(positions.getSelectedIndex());
}

void setUI(boolean isEnabled) {
  this.setEnabled(isEnabled);

  samples.setEnabled(isEnabled);
  jacobiPanel.setEnabled(isEnabled);
  finalGatheringPanel.setEnabled(isEnabled);
  photonPanel.setEnabled(isEnabled);
}

@Override
public void setEnabled(boolean isEnabled) {
  super.setEnabled(isEnabled);

  materials.setEnabled(isEnabled);
```

```
    positions.setEnabled(isEnabled);
    tabs.setEnabled(isEnabled);
    renderButton.setEnabled(isEnabled);
  }

  int getSamps() {
    return (Integer) samples.getValue();
  }
}
```

Classe FinalGatheringPanel:

```
package ui;

import javax.swing.*;
import java.awt.*;

public class FinalGatheringPanel extends JPanel {
  //campioni (numero di raggi) per l'illuminazione indiretta
  //(ricorsivo in global illumination, non ricorsivo in
  //final gathering)
  private JSpinner aoSamples;

  //campioni (numero di raggi) illuminazione diretta (non ricorsivo)
  private JSpinner dirSamples;

  //campioni scelti per le riflessioni e le rifrazioni
  private JSpinner refSamples;

  private JCheckBox jacobiCheck;

  FinalGatheringPanel() {
    super(new GridLayout(0, 1));

    aoSamples = new JSpinner(new SpinnerNumberModel(1, 1, 100, 1));
    dirSamples = new JSpinner(new SpinnerNumberModel(1, 1, 100, 1));
    refSamples = new JSpinner(new SpinnerNumberModel(75, 1, 1000, 1));
    jacobiCheck = new JCheckBox("Sperimentale: aggiunta di Jacobi al processo");
    jacobiCheck.setSelected(false);

    JPanel panel = new JPanel();
    panel.add(new JLabel("Campioni per pixel, illuminazione diretta"));
    panel.add(dirSamples);
    add(panel);

    panel = new JPanel();
    panel.add(new JLabel("Campioni per pixel, illuminazione indiretta"));
```

```
    panel.add(aoSamples);
    add(panel);

    panel = new JPanel();
    panel.add(new JLabel("Campioni per pixel, riflessioni e rifrazioni"));
    panel.add(refSamples);
    add(panel);

    add(jacobiCheck);
  }

  int getAOSamples() {
    return (Integer) aoSamples.getValue();
  }

  int getDirSamples() {
    return (Integer) dirSamples.getValue();
  }

  int getRefSamples() {
    return (Integer) refSamples.getValue();
  }

  boolean getJacobiCheck() {
    return jacobiCheck.isSelected();
  }

  @Override
  public void setEnabled(boolean isEnabled) {
    super.setEnabled(isEnabled);

    jacobiCheck.setSelected(isEnabled);
    aoSamples.setEnabled(isEnabled);
    dirSamples.setEnabled(isEnabled);
    refSamples.setEnabled(isEnabled);
  }
}
```

Classe InterfaceInitialiser:

```
import javax.swing.*;
import java.awt.*;

/* L'accesso a questa classe viene effettuato tramite
 *  chiamate statiche. Per questioni di sicurezza e di accesso
 *  alle variabili, sarebbe ideale creare un contollore
```

```
 *  che modifica solo i parametri necessari delle variabili
 *  invece che renderle statiche.
 * (Ad esempio, cambiare testo al label potrebbe essere fatto
 *  con un metodo che chiede come argomento una stringa,
 *  invece di dare l'accesso all'intera variabile)
 */

public class InterfaceInitialiser {
  //la classe main costruisce una stanza rettangolare con
  //dentro 3 sfere e opportune luci, e un osservatore che
  //guarda in una direzione appropriata: il generico raggio
  //di visuale attraversa un appropriato pixel del
  //viewplane. Su quel pixel vengono calcolati, mediante
  //metodi di radiosita' stocastica, colore e luminosita'
  //di cio' che l'osservatore vede.

  static JFrame mainFrame;
  public static final JLabel label = new JLabel();
  static EditPanel editPanel;

  public InterfaceInitialiser() {
    mainFrame = new JFrame("renderer.Renderer");
    //inizialmente imposto la finestra con le varie
    //scelte per l'utente

    try {
      // Set System L&F
      UIManager.setLookAndFeel
            (UIManager.getSystemLookAndFeelClassName());
    } catch (UnsupportedLookAndFeelException | ClassNotFoundException |
        InstantiationException | IllegalAccessException e) {
      // handle exception
    }

    editPanel = new EditPanel();
    mainFrame.add(editPanel);
    mainFrame.add(label, BorderLayout.PAGE_END);

    mainFrame.pack();
    mainFrame.setResizable(false);
    mainFrame.setLocationRelativeTo(null);
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.setVisible(true);
  }
}
```

Classe JacobiPanel:

```
package ui;

import javax.swing.*;
import java.awt.*;

public class JacobiPanel extends JPanel {
  // Sample per lo Stochastic Jacobi, utilizzati per il calcolo con Monte Carlo
  private JSpinner samples;

  // Step massimi per le iterazioni di Jacobi Stocastico
  private JSpinner maxSteps;

  // Errore massimo nel processo di Jacobi Stocastico
  private JSpinner maxErr;

  JacobiPanel() {
    super(new GridLayout(0, 1));

    samples = new JSpinner(new SpinnerNumberModel(15000, 1, 1000000, 1));
    maxSteps = new JSpinner(new SpinnerNumberModel(15, 1, 50, 1));
    maxErr = new JSpinner(new SpinnerNumberModel(3, 0, 10, 1));

    JPanel panel = new JPanel();
    panel.add(new JLabel("Campioni per Jacobi stocastico (Montecarlo)"));
    panel.add(samples);
    add(panel);

    panel = new JPanel();
    panel.add(new JLabel("Numero di iterazioni"));
    panel.add(maxSteps);
    add(panel);

    panel = new JPanel();
    panel.add(new JLabel("Errore massimo (potenza negativa di 10)"));
    panel.add(maxErr);
    add(panel);
  }

  int getSamps() {
    return (Integer) samples.getValue();
  }

  int getMaxSteps() {
    return (Integer) maxSteps.getValue();
  }

  double getMaxErr() {
    return Math.pow(10, -(Integer) maxErr.getValue());
```

```
  }

  @Override
  public void setEnabled(boolean isEnabled) {
    super.setEnabled(isEnabled);

    samples.setEnabled(isEnabled);
    maxSteps.setEnabled(isEnabled);
    maxErr.setEnabled(isEnabled);
  }
}
```

Classe Modeler:

```
package ui;

import primitive.ModelerProperties;
import primitive.Point3D;
import primitive.Sphere;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.Ellipse2D;
import java.util.ArrayList;

/* Metodo che inizializza una finestra per un semplice modellatore
 * La finestra Ăš strutturata sulla falsa riga di una proiezione
 * ortogonale dall'alto, posizionando sfere nello spazio
 */

/* Il problema principale della classe consiste nella sua natura.
 * Per la preview, utilizza un sistema molto approssimativo di
 *  Jacobi. Sarebbe utile utilizzare uno zBuffer per avere anche
 *  anteprime in tempo reale delle traslazioni, rotazioni e
 *  creazioni di oggetti.
 * In aggiunta, la simulazione della proiezione ortogonale Ăš
 *  approssimativa e non in scala con la scena.
 */

class Modeler extends JDialog {
  private final double SIZE =
      Toolkit.getDefaultToolkit().getScreenSize().getHeight()/2;
  private int radius = 75;

  class ImagePanel extends JPanel {
    ArrayList<Ellipse2D> ellipse2DS = new ArrayList<>();
```

```
  ImagePanel() {
    super(null);

    repaint();
  }

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);

    Graphics2D g2 = (Graphics2D) g;
    g2.setStroke(new BasicStroke(1.0f));

    if (RenderAction.spheres != null) {
      for (Sphere s : RenderAction.spheres) {
        g2.draw(new Ellipse2D.Double(
            SIZE + (s.p.x*50) ,
            s.p.z*50 + radius,
            s.rad*radius,
            s.rad*radius));
      }
    }

    if (ellipse2DS != null) {
      for (Ellipse2D e : ellipse2DS) {
        g2.draw(e);
      }
    }
  }
}

private JPanel buttonsPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
private JButton deleteButton = new JButton("Elimina ultima sfera");
private JButton doneButton = new JButton("Fatto");

ImagePanel imagePanel = new ImagePanel();

Modeler(JDialog frame) {
  super(frame, "Modellatore", true);

  deleteButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      if (!RenderAction.additionalSpheres.isEmpty() &&
          !imagePanel.ellipse2DS.isEmpty()) {
        RenderAction.additionalSpheres.remove
            (RenderAction.additionalSpheres.size()-1);
        imagePanel.ellipse2DS.remove (imagePanel.ellipse2DS.size()-1);
```

```
        imagePanel.repaint();
      }
    }
  });

  doneButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      new RenderAction(ModelerProperties.PREVIEW_ONLY);

      frame.repaint();
      dispose();
    }
  });

  addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
      super.mouseClicked(e);

      if (e.getButton() == MouseEvent.BUTTON1) {
        Object choice = JOptionPane.showInputDialog(
            Modeler.this,
            "Scelta materiale",
            "Scelta materiale",
            JOptionPane.QUESTION_MESSAGE,
            null,
            RenderAction.material, RenderAction.material[0]);
        int matId = 0;

        for (int i = 0; i < RenderAction.material.length; i++) {
          if (choice.toString().equals (RenderAction.material[i].toString())) {
            matId = i;
            break;
          }
        }

        //TODO add slider to manage sphere radius
        //TODO fix -8 & -31 parameters (incorrect position from MouseEvent e)
        imagePanel.ellipse2DS.add(new Ellipse2D.Double(
            e.getX() - (radius/ (double) 2) -8,
            e.getY()- (radius/ (double) 2) -31,
            radius,
            radius));
        Point3D newPoint = new Point3D((
            (e.getX() - SIZE)/ (double) 40),
            0,
            e.getY()/ (double) 75);
```

```
        RenderAction.additionalSpheres.add(new Sphere(1, newPoint, matId));

        repaint();
      }
    }
  });

  buttonsPanel.add(deleteButton);
  buttonsPanel.add(doneButton);

  add(buttonsPanel, BorderLayout.PAGE_END);

  add(imagePanel);

  addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
      super.windowClosing(e);

      int choice = JOptionPane.showConfirmDialog(
          Modeler.this,
          "Vuoi salvare le sfere create?",
          "Conferma",
          JOptionPane.YES_NO_OPTION);

      switch (choice) {
        case JOptionPane.YES_OPTION:
          dispose();
          break;
        case JOptionPane.NO_OPTION:
          RenderAction.additionalSpheres.clear();
          dispose();
          break;
      }
    }
  });

  setMinimumSize(new Dimension((int) SIZE, (int) SIZE));
  setLocationRelativeTo(frame);
  setResizable(false);
  setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
  setVisible(true);
 }
}
```

Classe PhotonPanel:

```
package ui;

import javax.swing.*;
import java.awt.*;

public class PhotonPanel extends JPanel {
  // Sample per la massima ricorsivitĂ  del photon mapping
  private JSpinner photonNum;

  // Sample per la massima ricorsivitĂ  del photon mapping
  private JSpinner causticNum;

  // Sample per la massima ricorsivitĂ  del photon mapping
  private JSpinner aoCaustic;

  //TODO add here
  private JSpinner projectResolution;

  //distanza al quadrato disco di ricerca dei fotoni
  private JSpinner photonSearchDisc;

  //distanza al quadrato disco di ricerca dei fotoni nell caustiche
  private JSpinner causticSearchDisc;

  //numero di fotoni da ricercare
  private JSpinner photonSearchNum;
  private JSpinner causticSearchNum;

  PhotonPanel() {
    super(new GridLayout(0, 1));

    photonNum = new JSpinner(new SpinnerNumberModel(100, 1, 10000, 1));
    causticNum = new JSpinner(new SpinnerNumberModel(100, 1, 10000, 1));
    aoCaustic = new JSpinner(new SpinnerNumberModel(150, 1, 10000, 1));
    projectResolution = new JSpinner(new SpinnerNumberModel(300, 1, 10000, 1));
    photonSearchNum = new JSpinner(new SpinnerNumberModel(80, 1, 10000, 1));
    causticSearchNum = new JSpinner(new SpinnerNumberModel(1500, 1, 10000, 1));
    photonSearchDisc = new JSpinner(new SpinnerNumberModel(1000, 1, 10000, 1));
    causticSearchDisc = new JSpinner(new SpinnerNumberModel(100, 1, 10000, 1));

    //TODO add caustic process checkbox

    JPanel panel = new JPanel();
    panel.add(new JLabel("Numero di fotoni generici"));
    panel.add(photonNum);
    add(panel);

    panel = new JPanel();
    panel.add(new JLabel("Numero di fotoni per le caustiche"));
```

```
  panel.add(causticNum);
  add(panel);

  panel = new JPanel();
  //TODO: fix text here
  panel.add(new JLabel("Numero di fotoni per le caustiche indirette"));
  panel.add(aoCaustic);
  add(panel);

  panel = new JPanel();
  panel.add(new JLabel("Risoluzione per la mappa di proiezione fotoni"));
  panel.add(projectResolution);
  add(panel);

  panel = new JPanel();
  panel.add(new JLabel("Numero di fotoni da ricercare"));
  panel.add(photonSearchNum);
  add(panel);

  panel = new JPanel();
  panel.add(new JLabel("Numero di caustiche da ricercare"));
  panel.add(causticSearchNum);
  add(panel);

  panel = new JPanel();
  panel.add(new JLabel("Grandezza del disco di ricerca fotoni"));
  panel.add(photonSearchDisc);
  add(panel);

  panel = new JPanel();
  panel.add(new JLabel("Grandezza del disco di ricerca caustiche"));
  panel.add(causticSearchDisc);
  add(panel);
}

int getPhotonNum() {
  return (Integer) photonNum.getValue();
}

int getCausticNum() {
  return (Integer) causticNum.getValue();
}

int getAOCaustic() {
  return (Integer) aoCaustic.getValue();
}

int getProjectionResolution() {
  return (Integer) projectResolution.getValue();
```

```
  }

  int getPhotonSearchNum() {
    return (Integer) photonSearchNum.getValue();
  }

  int getCausticSearchNum() {
    return (Integer) causticSearchNum.getValue();
  }

  int getPhotonSearchDisc() {
    return (Integer) photonSearchDisc.getValue();
  }

  int getCausticSearchDisc() {
    return (Integer) causticSearchDisc.getValue();
  }

  @Override
  public void setEnabled(boolean isEnabled) {
    super.setEnabled(isEnabled);

    photonNum.setEnabled(isEnabled);
    causticNum.setEnabled(isEnabled);
    aoCaustic.setEnabled(isEnabled);
    projectResolution.setEnabled(isEnabled);
    photonSearchNum.setEnabled(isEnabled);
    causticSearchNum.setEnabled(isEnabled);
    photonSearchDisc.setEnabled(isEnabled);
    causticSearchDisc.setEnabled(isEnabled);
  }
}
```

Classe Preview:

```
package ui;

import primitive.*;
import renderer.JacobiStocClass;
import renderer.Utilities;
import renderer.Renderer;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Hashtable;
```

```
import static primitive.ModelerProperties.PREVIEW_ONLY;
import static primitive.ModelerProperties.START_RENDERING;
import static ui.RenderAction.*;

public class Preview {
  public Preview(boolean isModeler) {
    // Interfaccia per mostrare l'immagine effettiva su schermo,
    boolean[] isRunning = {false};

    JDialog frame = new JDialog(InterfaceInitialiser.mainFrame, "Anteprima", true);
    frame.setMinimumSize(new Dimension(width, height));
    frame.setLocationRelativeTo(InterfaceInitialiser.editPanel);
    frame.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
    JPanel imagePanel = new JPanel(null) {
      @Override
      public void paintComponent(Graphics g) {
        super.paintComponent(g);

        for (int i = 0; i < height; i++) {
          for (int j = 0; j < width; j++) {
            g.setColor(image[j + i* width].toColor());
            g.drawLine(j, i, j, i);
          }
        }
      }
    };

    if (isModeler) {
      JButton changePositionButton = new JButton("Cambia posizioni");
      changePositionButton.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
          JDialog dialog = new JDialog(frame, "Cambia posizione", true);
          dialog.setMinimumSize(new Dimension(400, 500));
          dialog.setLocationRelativeTo(frame);
          dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
          JList<Obj> list = new JList<>(getEditableObjects());
          list.setModel(new DefaultListModel<>());

          Obj[] o = getEditableObjects();
          for (int i = 0; i < o.length; i++) {
            ((DefaultListModel<Obj>) list.getModel()).add(i, o[i]);
          }

          list.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
              super.mouseClicked(e);
```

```
if (e.getClickCount() == 2) {
  JDialog properties = new JDialog(dialog,
      list.getSelectedValue().toString(), true);
  properties.setLayout(new GridLayout(0, 1));
  JPanel panel = new JPanel();
  JSlider xSlider = new JSlider(-100, 100);
  xSlider.createStandardLabels(1);
  JSlider ySlider = new JSlider(-100, 100);
  ySlider.createStandardLabels(1);
  JSlider zSlider = new JSlider(-100, 100);
  JSlider rotatePhiSlider = new JSlider(0, 100, 0);
  JCheckBox[] checkBoxes = new JCheckBox[3];
  JButton abortButton = new JButton("Annulla");
  JButton doneButton = new JButton("Accetta modifiche");

  doneButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      Point3D direction = new Point3D(
          xSlider.getValue()/ (double) 10,
          ySlider.getValue()/ (double) 10,
          zSlider.getValue()/ (double) 10);

      for (Obj globalObject : globalObjects) {
        if (globalObject.equals (list.getSelectedValue())) {
          globalObject.setNewPosition(direction);

          if (globalObject.t != null) {
            Point3D axis = (new Point3D(
                checkBoxes[0].isSelected() ? 1 : 0,
                checkBoxes[1].isSelected() ? 1 : 0,
                checkBoxes[2].isSelected() ? 1 : 0))
                .getNormalizedPoint();
            globalObject.rotateTriangleOnly(
                axis,
                (rotatePhiSlider.getValue()*(2* Utilities.MATH_PI))/
                    (double) 100);
          }
        }
      }

      RenderAction.initialiseMeshes(PREVIEW_ONLY);
      JacobiStocClass.jacobiStoc(objects.size());
      Renderer.calculateThreadedRadiance(cam);

      frame.repaint();
      properties.dispose();

      ((DefaultListModel<Obj>) list.getModel()).removeAllElements();
```

```
    Obj[] o = RenderAction.getEditableObjects();

    for (int i = 0; i < o.length; i++) {
      ((DefaultListModel<Obj>) list.getModel()).add(i, o[i]);
    }
  }
});

panel.add(new JLabel("Posizione su X: "));
panel.add(xSlider);
properties.add(panel);

panel = new JPanel();
panel.add(new JLabel("Posizione su Y"));
panel.add(ySlider);
properties.add(panel);

panel = new JPanel();
panel.add(new JLabel("Posizione su Z"));
panel.add(zSlider);
properties.add(panel);

if (list.getSelectedValue().t != null) {
  panel = new JPanel();

  for (int i = 0; i < checkBoxes.length; i++) {
    String axis = i == 0 ? "X" : i == 1 ? "Y" : "Z";
    checkBoxes[i] = new JCheckBox("Asse " + axis);
    panel.add(checkBoxes[i]);
  }
  properties.add(panel);

  panel = new JPanel();
  panel.add(new JLabel("Rotazione sulla latitudine"));

  Hashtable<Integer, JLabel> table = new Hashtable<>();
  table.put(0, new JLabel("0"));
  table.put(25, new JLabel("\u03C0/2"));
  table.put(50, new JLabel("\u03C0"));
  table.put(75, new JLabel("3\u03C0/2"));
  table.put(100, new JLabel("2\u03C0"));
  rotatePhiSlider.setLabelTable(table);
  rotatePhiSlider.setPaintLabels(true);

  panel.add(rotatePhiSlider);

  properties.add(panel);
}
```

```
            panel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
            panel.add(abortButton);
            panel.add(doneButton);
            properties.add(panel);

            properties.setMinimumSize(new Dimension(300, 400));
            properties.setLocationRelativeTo(dialog);
            properties.setDefaultCloseOperation (JDialog.DISPOSE_ON_CLOSE);
            properties.setVisible(true);
          }
        }
      });

      dialog.add(list);

      dialog.setVisible(true);
    }
  });

  JPanel bottomPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
  JButton insert = new JButton("Aggiungi sfere");
  insert.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      new Modeler(frame);
    }
  });

  JButton button = new JButton("Avvia render");
  button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      isRunning[0] = true;

      frame.dispose();

      new Thread(new Runnable() {
        @Override
        public void run() {
          new RenderAction(START_RENDERING);
        }
      }).start();
    }
  });

  bottomPanel.add(changePositionButton);
  bottomPanel.add(insert);
  bottomPanel.add(button);
```

```
      frame.add(bottomPanel, BorderLayout.PAGE_END);
    }

    frame.add(imagePanel);

    frame.setVisible(true);

    frame.addWindowListener(new WindowAdapter() {
      @Override
      public void windowClosed(WindowEvent e) {
        super.windowClosed(e);

        if (isRunning[0]) {
          frame.dispose();
        } else {
          System.exit(0);
        }
      }
    });
  }
}
```

Classe Properties:

```
package ui;

public interface Properties {
  /* Interfaccia che contiene tutte le opzioni scelte nel Main e in ui.EditPanel
   *  per un utilizzo piÄč pratico tra le classi
   */
  int JACOBI_PANEL = 0;
  int FINAL_GATHERING_PANEL = 1;
  int PHOTON_PANEL = 2;

  String TRANSLUCENT_JADE = "Giada Realistica (traslucente)";
  String DIFFUSIVE_JADE = "Giada Diffusiva";
  String GLASS = "Vetro";

  String[] MATERIALS = {
      TRANSLUCENT_JADE,
      DIFFUSIVE_JADE,
      GLASS
  };

  String ALIGNED = "Sfere allineate";
  String OVERLAPPED = "Sfere sovrapposte";
```

```
  String[] POSITIONS = {
      OVERLAPPED,
      ALIGNED
  };
}
```

Classe RenderAction:

```
package ui;

import partition.Box;
import partition.PhotonBox;
import primitive.*;
import renderer.JacobiStocClass;
import renderer.PhotonScatterClass;
import renderer.Utilities;
import renderer.Renderer;

import java.io.FileOutputStream;
import java.io.IOException;
import java.text.DecimalFormat;
import java.util.ArrayList;

import static renderer.PhotonMappingClass.calculatePhotonMapping;

/* Classe nata per gestire l'intero processo del rendering.
 * Inizializza le variabili necessarie e reindirizza ai vari
 *  metodi di rendering in base alle scelte dell'utente.
 *
 * Il principale problema consiste nell'accesso a questa
 *  classe, da parte delle altre classi, tramite accessi
 *  statici. In aggiunta, si inizializzano i materiali tramite
 *  un array material da questa classe, il che richiede ogni
 *  volta l'accesso a RenderAction.
 */

public class RenderAction implements Properties, ModelerProperties {
  // Vettore dei materiali, per i modelli predefiniti di scena
    public static Material[] material = {
            //luce
            StandardMaterial.MATERIAL_LIGHT_WHITE,

            //diffusivi:
            StandardMaterial.MATERIAL_DIFFUSIVE_RED,
            StandardMaterial.MATERIAL_DIFFUSIVE_GREEN,
            StandardMaterial.MATERIAL_DIFFUSIVE_BLUE,
            StandardMaterial.MATERIAL_DIFFUSIVE_GRAY,
```

```
            StandardMaterial.MATERIAL_DIFFUSIVE_BLACK,

            //riflettenti:
            StandardMaterial.MATERIAL_REFLECTIVE_GLASS,
            StandardMaterial.MATERIAL_REFLECTIVE_PERFECT_GLASS,

            //materiali particolari:
            StandardMaterial.MATERIAL_COOK_TORRANCE_VIOLET,
            StandardMaterial.MATERIAL_STEEL,
            StandardMaterial.MATERIAL_DEEP_RED,
            StandardMaterial.MATERIAL_IMPERFECT_STEEL,
            StandardMaterial.MATERIAL_TRANSLUCENT_JADE,
            StandardMaterial.MATERIAL_DIFFUSIVE_PINK,
            StandardMaterial.MATERIAL_DIFFUSIVE_DEEP_GRAY,
            StandardMaterial.MATERIAL_DIFFUSIVE_JADE
    };

    /* Controlli booleani per accedere ai vari metodi per la scelta del
     *  tipo di rendering da effettuare
     *  -> Jacobi stocastico
     *  -> Final Gathering
     *  -> Photon Mapping
     */
    public static boolean doJacobi = false;
    public static boolean doFinalGathering = false;
    public static boolean doPhotonMapping = false;

    /* Queste variabili assegnano solo agli oggetti predefiniti
     *  (e quelli aggiunti nel modellatore) lo stesso tipo di materiale
     *   senza poter scegliere il tipo di materiale
     * PuÃČ essere corretto creando una variabile di controllo
     * nell'oggetto (Material),
     * dato che il renderer.Renderer giÃ  Ãš capace di determinare
     * le proprietÃ  del materiale selettivamente
     * (per la presenza della variabile matId nell'oggetto)
     */
    // Giada traslucente
    private static boolean translucentJade = false;
    // Giada diffusiva
    private static boolean diffusiveJade = false;
    // Vetro
    private static boolean glass = false;

    // Variabile di controllo per il posizionamento delle sfere
    /* Questa variabile puÃČ essere eliminata nel momento in cui il modellatore puÃČ
     *  gestire autonomamente la presenza e la posizione degli oggetti creati
     *  dall'utente
     */
    public static boolean aligned = false;
```

```
private static int SPHERES = 3; // Numero delle sfere create in via predefinita

// Direzione dal punto di vista (eye) verso l'oggetto
// La variabile verrĂ  definita in corso d'opera
static Point3D lookAt = new Point3D();

// Posizione della camera rispetto al centro della scena (0, 0, 0)
static Point3D eye = new Point3D(0.0f, 2.0f, 12.0f);

/* Punto di messa a fuoco della camera (puĂČ essere modificato per definire la
 *  posizione della messa a fuoco
 */
static Point3D focusPoint = new Point3D(0.0f);

//TODO add variables in UI
public static int width = 1080; //larghezza dell'immagine
public static int height = 720; //altezza dell'immagine

//costruttore della fotocamera
//imposto la fotocamera che guarda il centro
//dell'oggetto ed e' posizionata davanti
static Camera cam;
static Point3D focalPoint;
static float filmDistance = 700; // Lunghezza focale
/* Per una resa visiva del motore,
 * la lunghezza focale impostata a 700 fa si che l'apertura
 * del diaframma sia molto piccola,
 * al fine di avere un piano di messa a fuoco piĂč esteso.
 */

/* DensitĂ  triangoli nella stanza
 * Variabile che gestisce il grado di tessellazione della scena
 * suddividendo le mesh con splitMeshes()
 */
static int scenePrecision = 0;

// Contatore di suddivisione della scena (che dovrĂ  essere <= maxDepth
public static int depthLevel = 0;
// ProfonditĂ  massima di suddivisione dell'OCTree
public static int maxDepth = 14; //TODO add this variable to a specific menu

// Punti per il calcolo delle dimensioni della scena
// (creata dinamicamente in base al numero di oggetti)
public static Point3D max = null;
public static Point3D min = null;

// Variabile per la suddivisione della scena tramite BSP
// BSP: suddivisione dello spazio binaria secondo una struttura ad albero
```

```
public static partition.Box bound;

static ArrayList<Sphere> spheres;

// Variabile globale in cui verranno salvati gli oggetti della scena
static ArrayList<Obj> objects;    // Come vettore di supporto
public static ArrayList<Obj> globalObjects; // Come variabile globale

// Variabile globale per le luci della scena
public static ArrayList<Obj> lights;

// Array che contiene tutti i pixel (RGB) dell'immagine
/* E' possibile modificare l'array come array bidimensionale per una
   implementazione
 *  piu semplice nei cicli
 */
public static Point3D[] image = new Point3D[width * height];

// Vettori per i campioni casuali per la fotocamera (x, y)
public static int[] samplesX = new int[width * height];
public static int[] samplesY = new int[width * height];

// Vettori per i campioni casuali per l'illuminazione diretta
public static int[] dirSamples1 = new int[width * height];
public static int[] dirSamples2 = new int[width * height];
public static int[] dirSamples3 = new int[width * height];

// Vettori per i campioni casuali per l'illuminazione indiretta
public static int[] aoSamplesX = new int[width * height];
public static int[] aoSamplesY = new int[width * height];

// Vettori per i campioni di riflessione e rifrazione
public static int[] refSamples1 = new int[width * height];
public static int[] refSamples2 = new int[width * height];

// Background: lo impostiamo come bianco
public static Point3D background = new Point3D(1.0f,1.0f,1.0f);

// Campioni del pixel: numero di campioni per ogni pixels
// Si tratta dei punti nei pixel attraverso cui faremo passare raggi
public static int samps;

public static int aoSamps; // Campioni per l'illuminazione indiretta
public static int dirSamps; // Campioni illuminazione diretta (non ricorsivo)
public static int refSamps; // Campioni scelti per le riflessioni e le rifrazioni

// Lista di fotoni "sparati"
public static ArrayList<Photon> photons = new ArrayList<>();
// Lista di fotoni per caustiche "sparati"
```

```
public static ArrayList<Photon> caustics = new ArrayList<>();
// Suddivisione dello spazio secondo KDTree per la mappa fotonica
public static PhotonBox[] kdTree;
// Suddivisione dello spazio secondo KDTree per la mappa di caustiche
public static PhotonBox[] causticTree;
// KDTree: suddivisione dello spazio in sezioni K-dimensionali in una struttura
    ad albero
//ProfonditÄ  della suddivisione dello spazio per la mappa fotonica
//TODO add this variable to a specific menu
public static int kDepth = 17;
// Numero di fotoni da inviare nello spazio
public static int nPhoton;
// Numero di fotoni specifici per le caustiche da inviare agli oggetti
    trasparenti o traslucidi
public static int causticPhoton;
// Numero di fotoni per le caustiche per l'illuminazione indiretta
public static int aoCausticPhoton;
// Risoluzione della proiezione della mappa fotonica su un emisfero
public static int projectionResolution;
// Scalamento di potenza del fotone
public static float scaleCausticPower = 1;
//TODO add this variable in ui.PhotonPanel

// Distanza al quadrato disco di ricerca dei fotoni
public static double photonSearchDisc;
public static double causticSearchDisc;

// Numero di fotoni da ricercare
public static int nPhotonSearch;
public static int nCausticSearch;

// Contatore di potenza del fotone in base alla suddivisione kDepth
public static int power = 0;

// Contatore di step raggiunti dal processo Jacobi Stocastico
public static int steps = 0;
// Stima dell'errore raggiunto dal processo Jacobi Stocastico
public static double err = 0;
// Step massimi per le iterazioni di Jacobi Stocastico
public static int maxSteps;
// Errore massimo nel processo di Jacobi Stocastico
public static double maxErr;
// Sample per Jacobi Stocastico, utilizzati per il calcolo con Monte Carlo
public static int jacobiSamps;

//soglia dei triangoli dentro ad un box se ce ne sono di
//meno si ferma la partizione; si puo' regolare in base
//al numero totale dei triangoli della scena
public static int boxThreshold = 4;
```

```
/* nRay indica il numero di rimbalzi all'interno della scena
 * E' una variabile globale in modo da potersi aggiornare
 *  all'interno del metodo radiance()
 * Sarebbe anche possibile dare ricorsivamente il numero di
 *  raggi raggiunto nel metodo stesso
 */
public static int nRay = 0;

// Altezza da aggiungere alla stanza in fase di rendering
public static float hroom = 1.2f;

// Indice del materiale della luce della stanza (secondo il vettore material)
public static int matIdL=0;

//se true la luce e' frontale. Se si cambia in true,
//deccommentare le parti in createScene()
public static boolean frontL = false;

// Contatore per le box caricate durante la partizione spaziale della scena
public static int loadedBoxes = 0;

// Parametro per la ricerca sferica su una faccia piana in [0,1]
public static double sphericalSearch = 1;

/* Array di oggetti da inserire su scelta dell'utente
 * Array riempito all'interno del modellatore
 * Al momento sono aggiungibili solo sfere,
 // ma se si estende il modellatore anche ad altri
 *  oggetti, e' possibile estendere questo array a piĂč oggetti
 */
static ArrayList<Sphere> additionalSpheres = new ArrayList<>();

RenderAction(int modelerProperties) {
  // Il valore booleano fa sapere al programma in che modo
  // deve partire il render (se semplice o quello finale)
  doRender(modelerProperties);
}

private void doRender(int modelerProperties) {
  InterfaceInitialiser.label.setText("Creazione immagine in corso");
  InterfaceInitialiser.editPanel.setUI(false);

  // Inizializzo l'array nei quali conservare gli oggetti di scena
  //TODO rendere passaggio valido per oggetti generici
  int mIS = setMatIdSphere();
  spheres = new ArrayList<>();

  // Oggetti predefiniti
```

```
for(int sph = 0; sph < SPHERES; sph++) {
  Point3D sPos = Sphere.setSpheresPosition(sph);

  //vettore costruttore delle sfere
  spheres.add(new Sphere(1, sPos, mIS));
}

// Aggiunta di oggetti dal modellatore
if (additionalSpheres != null && !additionalSpheres.isEmpty()) {
  spheres.addAll(additionalSpheres);
  SPHERES = spheres.size();
}

/* Reimposta le variabili utilizzate
 * Il metodo risulta utile quando si avvia il render
 *  dopo aver utilizzato il modellatore (ripristino una condizione originale)
 *  e in caso si volesse rieseguire un render dopo averne effettuato uno in
     precedenza
 */
resetVariables();

if (modelerProperties == ENABLE_MODELER || modelerProperties == PREVIEW_ONLY) {
  // Impostazioni per un render veloce
  // ma sarebbe opportuno aggiungere altri metodi
  // (ad esempio uno zBuffer)
  samps = 1;

  doJacobi = true;

  jacobiSamps = 175;
  maxSteps = 3;
  maxErr = 0.01f;
} else if (modelerProperties == START_RENDERING) {
  // Impostazioni per il rendering effettivo
  samps = InterfaceInitialiser.editPanel.getSamps();

  switch (InterfaceInitialiser.editPanel.getMethod()) {
    case JACOBI_PANEL:
      doJacobi = true;

      jacobiSamps = InterfaceInitialiser.editPanel.jacobiPanel .getSamps();
      maxSteps = InterfaceInitialiser.editPanel.jacobiPanel .getMaxSteps();
      maxErr = InterfaceInitialiser.editPanel.jacobiPanel .getMaxErr();
      break;
    case FINAL_GATHERING_PANEL:
      doFinalGathering = true;

      aoSamps = InterfaceInitialiser.editPanel.finalGatheringPanel
          .getAOSamples();
```

```
      dirSamps = InterfaceInitialiser.editPanel.finalGatheringPanel
          .getDirSamples();
      refSamps = InterfaceInitialiser.editPanel.finalGatheringPanel
          .getRefSamples();
      break;
    case PHOTON_PANEL:
      doPhotonMapping = true;

      nPhoton = InterfaceInitialiser.editPanel.photonPanel .getPhotonNum();
      causticPhoton = InterfaceInitialiser.editPanel.photonPanel
          .getCausticNum();
      aoCausticPhoton = InterfaceInitialiser.editPanel.photonPanel
          .getAOCaustic();
      projectionResolution = InterfaceInitialiser.editPanel.photonPanel
          .getProjectionResolution();
      photonSearchDisc = InterfaceInitialiser.editPanel.photonPanel
          .getPhotonSearchDisc();
      nPhotonSearch = InterfaceInitialiser.editPanel.photonPanel
          .getPhotonSearchNum();
      causticSearchDisc = InterfaceInitialiser.editPanel.photonPanel
          .getCausticSearchDisc();
      nCausticSearch = InterfaceInitialiser.editPanel.photonPanel
          .getCausticSearchNum();
      break;
  }
}

// Impostazione del materiale predefinito
switch (InterfaceInitialiser.editPanel.getMaterial()) {
  case TRANSLUCENT_JADE:
    translucentJade = true;
    break;
  case DIFFUSIVE_JADE:
    diffusiveJade = true;
    break;
  case GLASS:
    glass = true;
    break;
}

// Impostazione delle posizioni predefinite
switch (InterfaceInitialiser.editPanel.getPosition()) {
  case ALIGNED:
    aligned = true;
    break;
  case OVERLAPPED:
    aligned = false;
    break;
}
```

```
//TODO le opzioni predefinite possono essere modificate
//per raggiungere un livello di creazione dinamico
initialiseMeshes(modelerProperties);

//richiamo la funzione per il calcolo della radiosita'
//della scena attraverso il metodo di Jacobi
//stocastico

if(doJacobi || InterfaceInitialiser.editPanel.finalGatheringPanel
    .getJacobiCheck()) {
  // Avvio il calcolo dell'energia uscente dal metodo di Jacobi stocastico
  JacobiStocClass.jacobiStoc(objects.size());
}

if (doPhotonMapping) {
  // Avvio il calcolo dell'energia ottenuta dai fotoni
  calculatePhotonMapping();
}

// Calcolo delle direzioni dei sample in anticipo
for (int i = 0; i < (width * height); i++) {
  //creiamo i campioni necessari per:

  //la fotocamera
  samplesX[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);
  samplesY[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);

  //la luce indiretta
  aoSamplesX[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);
  aoSamplesY[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);

  //la luce diretta
  dirSamples1[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);
  dirSamples2[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);
  dirSamples3[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);

  //riflessioni/rifrazioni
  refSamples1[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);
  refSamples1[i] = (int) (Math.random()*(Integer.MAX_VALUE) +1);

  //inizializzo l'immagine nera
  image[i] = new Point3D();
}

/* Avvio il render effettivo (automaticamente effettuerÃ  un render completo
 *  o meno a seconda dei calcoli effettuati nelle righe precedenti
 */
Renderer.calculateThreadedRadiance(cam);
```

```
  //Ora viene creata l'immagine (per il modellatore viene solo mostrata,
  // per il render invece viene salvata)
  if (modelerProperties == ENABLE_MODELER) {
    showImage(true);
  } else if (modelerProperties == START_RENDERING) {
    createImage();
    //showImage(false);

    InterfaceInitialiser.editPanel.setUI(true);
  }
}

static ArrayList<Sphere> getSpheresFromGlobalObjects() {
  ArrayList<Sphere> spheres = new ArrayList<>();

  for (Obj o : globalObjects) {
    if (o.s != null) {
      spheres.add(o.s);
    }
  }

  return spheres;
}

static Obj[] getEditableObjects() {
  ArrayList<Obj> objs = new ArrayList<>();

  for (Obj o : globalObjects) {
    if (o.s != null) {
      objs.add(o);
    } else if (o.t != null) {
      if (!o.t.isBorderMeshScene) {
        objs.add(o);
      }
    }
  }

  return objs.toArray(new Obj[0]);
}

void showImage(boolean isModeler) {
  new Preview(isModeler);
}

private void createImage() {
  //stringa contenente le informazioni da scrivere nel file immagine image.ppm
  StringBuilder matrix = new StringBuilder();

  //iniziamo la stringa matrix con valori di settaggi richiesti
```

```
    matrix.append("P3\n").append(width).append("\n").append(height).append("\n255\n");

    //Ora si disegna l'immagine: si procede aggiungendo
    //alla stringa matrix le informazioni contenute
    //nell'array image in cui abbiamo precedentemente
    //salvato tutti i valori di radianza
    for(int i = 0; i < width * height; i++) {
      //stampiamo la percentuale di completamento per
      //monitorare l'avanzamento della creazione
      //dell'immagine
      double percent = (i / (float)(width * height)) * 100;
      double percentFloor=Math.floor(percent);
      double a=percent-percentFloor;
      if(a==0.0) {
        InterfaceInitialiser.label.setText("Percentuale di completamento immagine: "
            + new DecimalFormat("###.##").format(percent));
      }

      //i valori di radianza devono essere trasformati
      //nell'intervallo [0,255] per rappresentare la
      //gamma cromatica in valori RGB
      matrix.append(Utilities.toInt(image[i].x))
          .append(" ")
          .append(Utilities.toInt(image[i].y))
          .append(" ")
          .append(Utilities.toInt(image[i].z))
          .append(" ");
    }

    InterfaceInitialiser.label.setText("Immagine completata!");

    //nome del file in cui si andra' a salvare l'immagine
    //di output
    String filename = "image.ppm";

    //Per finire viene scritto il file con con permesso
    //di scrittura e chiuso.
    FileOutputStream fos;
    try {
      fos = new FileOutputStream(filename);
      fos.write(matrix.toString().getBytes());
      fos.close();
    } catch (IOException e1) {
      e1.printStackTrace();
    }
  }

  static void initialiseMeshes(int modelPreview) {
    //dovendo disegnare 3 sfere e la stanza definisco un
```

```
//array di 2 primitive.Mesh: nella prima delle due mesh (per
//meshes[0]) aggiungiamo le 3
//sfere richiamando il metodo caricaSphere

//array di mesh della scena
ArrayList<Mesh> meshes = new ArrayList<>(2);
meshes.add(new Mesh(
    modelPreview == PREVIEW_ONLY ? getSpheresFromGlobalObjects() : spheres));

//inizializzo dei fittizi massimo e minimo, che mi
//serviranno per definire i valori di max e min
Point3D oldMin = new Point3D(Float.POSITIVE_INFINITY);
Point3D oldMax = new Point3D(Float.NEGATIVE_INFINITY);

//trovo le dimensioni della scena (tralasciando la
//stanza in cui gli oggetti sono contenuti)
for (Mesh tmpObjects : meshes) {
  max = Obj.getBoundMax(tmpObjects.objects, oldMax);
  min = Obj.getBoundMin(tmpObjects.objects, oldMin);
}

//definisco e calcolo il punto in cui guarda
//l'osservatore: il centro della scena
Point3D center = (max.add(min)).multiplyScalar(0.5f)
    .subtract(new Point3D(0, 0.8, 0));

//parametri della fotocamera con sistema di riferimento centrato nella scena:
//absolutePos e' vero se stiamo guardando proprio al centro della scena
boolean absolutePos = false;

if (modelPreview == ENABLE_MODELER || modelPreview == PREVIEW_ONLY) {
  lookAt = new Point3D();
}

//TODO check variable usage
if(!absolutePos) {
  lookAt = lookAt.add(center);
}

//l'osservatore si trova nel punto camPosition
Point3D camPosition = center.add(eye);

// Inizializzo il punto di messa a fuoco
focalPoint = center.add(focusPoint);

cam = new Camera(camPosition, lookAt,
    new Point3D(0.00015f,1.00021f,0.0f),
    width, height, filmDistance);
```

```
//Abbiamo ora a disposizione tutti gli elementi
//necessari per costruire la stanza
//Creo la stanza in cui mettere l'oggetto (per
//visualizzare l'illuminazione globale)
//La carico come ultima mesh
meshes.add(new Mesh(max, min));

//nel nostro caso scenePrecision=0, quindi non si entra
//mai in questo ciclo
for(int q = 0; q < scenePrecision; q++) {
  meshes.get(meshes.size()-1).splitMeshes();
}

//A questo punto consideriamo l'intero array di mesh,
//ora composto da oggetti+stanza e aggiorno i valori
//della grandezza della stanza, usando di nuovo i
//metodi getBoundMin e getBoundMax.
oldMax = max;
oldMin = min;
max = Obj.getBoundMax(meshes.get(meshes.size()-1).objects, oldMax);
min = Obj.getBoundMin(meshes.get(meshes.size()-1).objects, oldMin);

//vettore che conterra' gli oggetti della scena
objects = new ArrayList<>();
//vettore che contiene solo le luci della scena
lights = new ArrayList<>();

for (Mesh tmpMesh : meshes) {
  //e carico tutto nella lista globale degli oggetti
  for (int j = 0; j < tmpMesh.objects.size(); j++) {
    objects.add(tmpMesh.objects.get(j));
    //se l'oggetto e' una luce la carico dentro
    //l'array delle luci
    if (material[tmpMesh.objects .get(j).matId].emittedLight.max() > 0) {
      lights.add(tmpMesh.objects.get(j));
    }
  }
}

// Aggiungo un oggetto (triangolo) predefinito nella scena, uno specchio
if (modelPreview == ENABLE_MODELER) {
  objects.add(new Obj(new Triangle(
      new Point3D(-5.5, 0, 1.5),
      new Point3D(-6.2, 0, 2),
      new Point3D(-5.75, 1, 1.75),
      7)));
} else {
  for (Obj o : globalObjects) {
    if (o.t != null && !o.t.isBorderMeshScene) {
```

```
        objects.add(o);
      }
    }
  }

  //depthLevel e' il livello di profondita' all'interno dell'albero
  depthLevel = 0;

  //creo il Bounding partition.Box
  //Bound e' il primo elemento dell'albero che contiene tutti gli oggetti della
      scena
  bound = new partition.Box(min, max, 0);
  bound.setObjects(objects);

  //crea il tree: si richiama il metodo setPartition()
  //per dividere gli oggetti del box padre nei box figli
  bound = Box.setPartition(bound);

  // Fattore di scala per la messa a fuoco in base al
  // punto focale e alla camera tramite trasformazione prospettica
  cam.fuoco = (focalPoint.z-cam.eye.z)/(cam.W.z*(-cam.d));

  //salviamo gli oggetti della scena nella variabile
  //globale globalObjects in modo da poterli aggiornare
  //in jacobiStoc()
  globalObjects = new ArrayList<>();
  globalObjects.addAll(objects);
}

private void resetVariables() {
  // Reimpostazione delle variabili allo stato originale
  doJacobi = false;
  doFinalGathering = false;
  doPhotonMapping = false;

  translucentJade=false;
  diffusiveJade=false;
  glass=false;
  aligned=false;

  lookAt =new Point3D(0.0f);

  focusPoint=new Point3D(0.0f);

  scenePrecision = 0;
  lights = new ArrayList<>();

  depthLevel = 0;
```

```
    max = null;
    min = null;

    globalObjects = new ArrayList<>();

    samplesX=new int[width * height];
    samplesY=new int[width * height];

    aoSamplesX=new int[width * height];
    aoSamplesY=new int[width * height];

    dirSamples1=new int[width * height];
    dirSamples2=new int[width * height];
    dirSamples3=new int[width * height];

    refSamples1=new int[width * height];
    refSamples2=new int[width * height];

    image = new Point3D[width * height];

    photons = new ArrayList<>();
    caustics = new ArrayList<>();

    steps = 0;
    nRay = 0;
    err = 0;

    boxThreshold =4;

    frontL=false;
    loadedBoxes = 0;
    sphericalSearch = 1;
  }

  //metodo che imposta, a seconda della scelta
  //effettuata dall'utente, il materiale
  //appropriato alle prime tre sfere
  private int setMatIdSphere() {
    int translucentJadeIndex=12;
    int diffusiveJadeIndex=15;
    int glassIndex=7;
    int one=1;

    if(translucentJade)
      return translucentJadeIndex;
    else if(diffusiveJade)
      return diffusiveJadeIndex;
    else if(glass)
      return glassIndex;
```

```
    return one;
  }
}
```

CAPITOLO 21

Appendice: modellazione di moti planetari nel linguaggio Java

Un codice Java ed uno C++ per la modellazione e rendering del sistema Terra-Luna sono stati presentati nelle Appendici 5.8 e 5.9 del Capitolo 5. Qui estendiamo questo codice (in Java) alla modellazione di un sistema con Terra e cinque asteroidi o pianeti. Oltre all'equazione della dinamica applicata ai baricentri dei pianeti, consideriamo anche l'equazione del moto dei corpi rigidi per illustrare come gli assi di rotazione vengano modificati nel corso del moto. La modifica consiste nella precessione degli assi di rotazione dei pianeti: si tratta dello stesso effetto di una trottola, ovvero la precessione degli equinozi (solo dei pianeti, perché si assume che la Terra abbia massa molto maggiore). Con le costanti scelte in questo codice, la precessione è molto esagerata, ed i pianeti sembrano ondeggiare. Il problema del moto gravitazionale di più di due corpi non ha una soluzione analitica, e l'esame delle traiettorie, reso più chiaro dal rendering tridimensionale (incluse le eclissi), è un opportuno esempio di come il rendering 3D possa servire ad illustrare fenomeni scientifici. Le equazioni del moto di un corpo rigido sottoposto ad una forza radiale non fanno precedere l'asse di rotazione se il corpo ha simmetria sferica: ad esempio, una sfera di densità uniforme non precede. Invece un anello sì. come si vede se si fa ruotare una moneta. Pertsanto occorre supporre che i pianeti non abbiano simmetria sferica, ad esempio perché sono schiacciati ad un polo. Allora il loro momento di inerzia rispetto al baricentro è non nullo e sotto l'azione di una forza centrale, come la forza di gravità della Terra, il loro asse di rotazione precede. Però anche la forza di gravità generata dagli altri pianeti induce una precessione, e pertanto, così come succede alle loor orbite, anche i loro assi di rotazione seguono un andamento molto caotico, che questa animazione calcola ed illustra.

Il codice è stato elaborato da Ilaria Teodorii [?Teodori-Terra_e_5_lune].

Main.java:

```
/*
  File di definizione della classe Main.
  Ha il compito di creare le costanti e le variabili globali e di creare i pianeti
      e far partire i threads del
  loro movimento, creando la finestra in cui lo si visualizza.
*/



public class Main {

   static final int numPlanets = 6;

   public static Sphere earth=new Sphere(), moon=new Sphere(), venus=new Sphere(),
       mercury=new Sphere(), jupiter=new Sphere(), mars=new Sphere();
```

```
public static Sphere earth_tmp=new Sphere(), moon_tmp=new Sphere(),
    venus_tmp=new Sphere(), mercury_tmp=new Sphere(), jupiter_tmp=new Sphere(),
    mars_tmp=new Sphere();
static Thread t, t_eu;
public static Sphere[] pianeti, pianeti_tmp;
public static double aPrecStart[] = new double[6];
public static int T=0;
public static int width = 850;
public static int height = 680;
public static long millisOLD;
public static long millisNEW;
public static PhysicsEngineRK rk;
public static PhysicsEngineEuler eu;

public static void main(String args[]) {

   System.out.println("start");
   millisOLD = System.currentTimeMillis();

   pianeti = new Sphere[] { earth, moon, mars, venus, mercury, jupiter };
   pianeti_tmp = new Sphere[] { earth_tmp, moon_tmp, mars_tmp, venus_tmp,
       mercury_tmp, jupiter_tmp };

   Sphere.createPlanets(pianeti, aPrecStart, pianeti_tmp);

   rk = new PhysicsEngineRK(pianeti, pianeti_tmp);
   eu = new PhysicsEngineEuler(pianeti, pianeti_tmp);

   t = new Thread(rk);

   MyFrame mainWindow = new MyFrame("Simulazione 5 Lune", width, height,pianeti);

    t.start();

   while (true) {

      millisNEW = System.currentTimeMillis();

      if (millisNEW - millisOLD > 50) {
         millisOLD = millisNEW;

          mainWindow.repaint();

         T+=1; //variabile temporale

      }
   }
}
```

MyFrame.java:

```
/*
  File di definizione della classe MyFrame, che crea e gestisce il Frame di
      interfaccia ed i suoi ascoltatori di eventi,
  ed anche i pannelli inseriti nel Frame (pannello di visualizzazione 3D, pannelli
      di proiezione ortografica sui piani coordinati,
  pulsanti di scelta delle tessiture, delle luci, della attivazione della
      precessione, del metodo di integrazione numerica.
 /*

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.KeyEventDispatcher;
import java.awt.KeyboardFocusManager;
import java.awt.Label;
import java.awt.TextArea;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.InputStream;

import javax.imageio.ImageIO;
import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class MyFrame extends JFrame {
   /**
    *
    */
   private static final long serialVersionUID = 1L;
```

```
static final int numPlanets = 6;

public static UpPanel titleContainer = new UpPanel();
public static TitlePanel title = new TitlePanel();
public static ButtonPanel buttons = new ButtonPanel();
public static HomePanel home = new HomePanel();
public static OrthoProjections o_pj;
public static HelpPanel help = new HelpPanel();
public static PlanetPanel planet = new PlanetPanel();
public static PlanetsContainerPanel planetH = new PlanetsContainerPanel();
public static HomeOptionsPanel home_opz = new HomeOptionsPanel();

public static Canvas canvas;
public static RenderingEngine _rnd;
public static Scene _scn;

public static Sphere[] _planets;

public MyFrame(String s, int width, int height, Sphere[] pianeti) {
   super(s);// costruttore della classe genitore
   setTitle("Render");
   setSize(1500, 900);
   setLayout(new BorderLayout());
   _planets = pianeti;


   o_pj = new OrthoProjections(width, height, _planets);

   add(title);
   add(titleContainer);
   add(buttons);
   add(home_opz);
   add(home);
   add(o_pj);
   o_pj.setVisible(false);
   add(help);
   help.setVisible(false);
   add(planet);
   planet.setVisible(false);
   add(planetH);
   planetH.setVisible(false);

   _scn = new Scene(0.05);
   // Creo e aggiungo le sorgenti di luce
   Light l1 = new Light(-366, -166, 800, 300);
   Light l2 = new Light(3000, 0, 0, 1);
   l2.l = 0.0;
   l2.c = 1.0;
   l1.l = 0.5;
```

```
l2.q = l1.q = 0;
_scn.lgt.add(l1);
_scn.lgt.add(l2);
for (int i = 0; i < numPlanets; i++) {
   _scn.addSphere(_planets[i]);
}

_rnd = new RenderingEngine(width, height, 850, new Point(0, 0, -1200), new
    Point(0, 0, 1), _scn);
_rnd.texture_mapping = false;

canvas = new Canvas(width, height, _rnd);
canvas.setPreferredSize(new Dimension(1500, 900));
canvas.setVisible(true);
this.add(canvas);

this.addMouseListener(new MouseListener() {

   public void mousePressed(MouseEvent e) {
      // System.out.println("Mouse pressed; # of clicks: ");
   }

   public void mouseReleased(MouseEvent e) {
      // System.out.println("Mouse released; # of clicks: ");
   }

   public void mouseEntered(MouseEvent e) {
      // System.out.println("Mouse entered");
   }

   public void mouseExited(MouseEvent e) {
      // System.out.println("Mouse exited");
   }

   public void mouseClicked(MouseEvent e) {
      Point p = _rnd.camera.sum(e.getX() - _rnd.w_2, e.getY() - _rnd.h_2, 0);
      p.z = 800;
      _scn.lgt.get(0).set(p);
   }
});

KeyboardFocusManager.getCurrentKeyboardFocusManager().addKeyEventDispatcher(new
    KeyEventDispatcher() {
   public double vx = 0.0, vy = 0.0, vz = 0.0;
   public int focal;

   @Override
   public boolean dispatchKeyEvent(KeyEvent e) {
      if (e.getKeyCode() == KeyEvent.VK_UP)
```

```
            vy = -10;
         if (e.getKeyCode() == KeyEvent.VK_DOWN)
            vy = 10;
         if (e.getKeyCode() == KeyEvent.VK_LEFT)
            vx = -10;
         if (e.getKeyCode() == KeyEvent.VK_RIGHT)
            vx = 10;
         if (e.getKeyCode() == KeyEvent.VK_F) {
            vz = 10;
         }
         if (e.getKeyCode() == KeyEvent.VK_B) {
            vz = -10;
         }

         _rnd.shiftCamera(new Point(vx, vy, vz));
         vx = 0;
         vy = 0;
         vz = 0;
         return false;
      }
   });

   this.setFocusable(true);
   this.setAutoRequestFocus(true);
   this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setVisible(true);

}

}

class UpPanel extends JPanel { // pannello titolo
   private static final long serialVersionUID = 1L;

   public UpPanel() {
      setBounds(0, 0, 850, 80);
   }
}

class TitlePanel extends JPanel { // pannello interno al pannello titolo

   private static final long serialVersionUID = 1L;
   JLabel etichetta = new JLabel("SCIENZE E TECNOLOGIE PER I MEDIA");
   JLabel etichetta2 = new JLabel("Metodi Matematici per Computer Graphics");

   public TitlePanel() {
      BufferedImage myPicture;
      JLabel picLabel = new JLabel();
      try {
```

```
            ClassLoader classLoader = this.getClass().getClassLoader();
            InputStream is = classLoader.getResourceAsStream("images/Uniroma22.png");
            myPicture = ImageIO.read(is);

            picLabel = new JLabel(new ImageIcon(myPicture));
         } catch (IOException e) {
            e.printStackTrace();
         }
         setBounds(0, 10, 800, 55);
         add(picLabel);
         add(etichetta);
         add(etichetta2);
         etichetta.setFont(new Font("SansSerif", Font.BOLD, 20));
         etichetta2.setFont(new Font("SansSerif", Font.BOLD, 15));
      }
   }

   class ButtonPanel extends JPanel {// pannello bottoni home - menu'

      private static final long serialVersionUID = 1L;
      JButton home = new JButton("Home");
      JButton ortho = new JButton("OrthoPanels");
      JButton pianeti = new JButton("Pianeti");
      JButton help = new JButton("Help");
      JButton exit = new JButton("Exit");

      public ButtonPanel() {
         setBounds(800, 10, 500, 65);

         add(home);
         add(ortho);
         add(pianeti);
         add(help);
         add(exit);

         MenuButtonAction homeAL = new MenuButtonAction();
         home.addActionListener(homeAL);

         MenuButtonAction orthoAL = new MenuButtonAction();
         ortho.addActionListener(orthoAL);

         MenuButtonAction pianetiAL = new MenuButtonAction();
         pianeti.addActionListener(pianetiAL);

         MenuButtonAction helpAL = new MenuButtonAction();
         help.addActionListener(helpAL);

         MenuButtonAction exitAL = new MenuButtonAction();
         exit.addActionListener(exitAL);
```

```
   }
}

class HomePanel extends JPanel {// pannello container home

   private static final long serialVersionUID = 1L;

   public HomePanel() {
      Label etichetta = new Label("HOME");
      etichetta.setFont(new Font("SansSerif", Font.BOLD, 17));

      setBorder(BorderFactory.createLineBorder(Color.black));
      setBounds(850, 80, 500, 600);

      add(etichetta);
   }
}

class HomeOptionsPanel extends JPanel { // pannello home con opzioni

   private static final long serialVersionUID = 1L;
   JPanel texture = new JPanel();
   JPanel shading = new JPanel();
   JPanel gouraud = new JPanel();
   JPanel integ = new JPanel();
   JPanel light = new JPanel();
   JPanel ifTexture = new JPanel();

   Label txtr = new Label("Show Texture: ");
   Label shd = new Label("Shading type: ");
   Label grd = new Label("Gouraud shading: ");
   Label alg = new Label("Algorithm: ");
   Label lgt = new Label("Luce 2: posizionare con click sinistro");
   Label ifTxtr = new Label("Solo con texture attiva: ");

   JButton texture_opz1 = new JButton("off");
   JButton texture_opz2 = new JButton("planets");
   JButton texture_opz3 = new JButton("chess");
   JButton texture_opz4 = new JButton("stripes");
   JButton shading_opz1 = new JButton("off");
   JButton shading_opz2 = new JButton("Phong");
   JButton shading_opz3 = new JButton("Blinn");
   JButton gouraud_opz1 = new JButton("off");
   JButton gouraud_opz2 = new JButton("on");
   JCheckBox integ_opz1 = new JCheckBox("Runge Kutta 4", true);
   JCheckBox integ_opz2 = new JCheckBox("Eulero", false);
   JButton light_opz1 = new JButton("off");
   JButton light_opz2 = new JButton("on");
   JCheckBox nuvole = new JCheckBox("Nuvole");
```

```
JCheckBox prec = new JCheckBox("Precessione Equinozi");

public double intensity;

public HomeOptionsPanel() {
   setBounds(880, 150, 450, 500);
   setFont(new Font("SansSerif", Font.BOLD, 10));

   texture.setPreferredSize(new Dimension(450, 70));
   txtr.setFont(new Font("SansSerif", Font.BOLD, 14));
   texture.add(txtr);
   texture.add(texture_opz1);
   texture.add(texture_opz2);
   texture.add(texture_opz3);
   texture.add(texture_opz4);
   add(texture);

   ifTexture.setPreferredSize(new Dimension(450, 70));
   ifTxtr.setFont(new Font("SansSerif", Font.BOLD, 14));
   ifTexture.add(ifTxtr);
   ifTexture.add(nuvole);
   nuvole.setEnabled(false);
   ifTexture.add(prec);
   prec.setEnabled(false);
   add(ifTexture);

   shading.setPreferredSize(new Dimension(450, 70));
   shd.setFont(new Font("SansSerif", Font.BOLD, 14));
   shading.add(shd);
   shading.add(shading_opz1);
   shading.add(shading_opz2);
   shading.add(shading_opz3);
   add(shading);

   gouraud.setPreferredSize(new Dimension(450, 70));
   grd.setFont(new Font("SansSerif", Font.BOLD, 14));
   gouraud.add(grd);
   gouraud.add(gouraud_opz1);
   gouraud.add(gouraud_opz2);
   add(gouraud);

   integ.setPreferredSize(new Dimension(450, 70));
   alg.setFont(new Font("SansSerif", Font.BOLD, 14));
   integ.add(alg);
   integ.add(integ_opz1);
   integ.add(integ_opz2);
   add(integ);
   integ_opz1.setEnabled(false);
   integ_opz1.setSelected(true);
```

```
light.setPreferredSize(new Dimension(450, 70));
lgt.setFont(new Font("SansSerif", Font.BOLD, 14));
light.add(lgt);
light.add(light_opz1);
light.add(light_opz2);
add(light);

texture_opz1.addActionListener(new ActionListener() { // Texture OFF button

   @Override
   public void actionPerformed(ActionEvent e) {
      MyFrame._rnd.texture_mapping = false;
      nuvole.setEnabled(false);
      nuvole.setSelected(false);
      for (int i = 0; i < MyFrame.numPlanets; i++) {
         MyFrame._planets[i].precessione = false;
      }
      prec.setEnabled(false);
      prec.setSelected(false);
      Sphere.c = false;

   }
});

texture_opz2.addActionListener(new ActionListener() { // Texture planets
    button

   @Override
   public void actionPerformed(ActionEvent e) {
      for (int i = 0; i < MyFrame.numPlanets; i++) {
         MyFrame._planets[i].setTexture(MyFrame._planets[i].texture_name);
      }
      MyFrame._rnd.texture_mapping = true;
      nuvole.setEnabled(true);
      prec.setEnabled(true);
      nuvole.setSelected(false);
      prec.setSelected(false);
      Sphere.c = false;
      for (int i = 0; i < MyFrame.numPlanets; i++) {
         MyFrame._planets[i].precessione = false;
      }

   }
});

texture_opz3.addActionListener(new ActionListener() { // Texture 'chess'
    button
```

```
   @Override
   public void actionPerformed(ActionEvent e) {
      for (int i = 0; i < MyFrame.numPlanets; i++) {
         MyFrame._planets[i].setTexture(MyFrame._planets[i].texture_chess);
      }
      MyFrame._rnd.texture_mapping = true;
      nuvole.setEnabled(true);
      prec.setEnabled(true);
      nuvole.setSelected(false);
      prec.setSelected(false);
      Sphere.c = false;
      for (int i = 0; i < MyFrame.numPlanets; i++) {
         MyFrame._planets[i].precessione = false;
      }

   }
});

texture_opz4.addActionListener(new ActionListener() { // Texture 'stripes'
    button

   @Override
   public void actionPerformed(ActionEvent e) {
      for (int i = 0; i < MyFrame.numPlanets; i++) {
         MyFrame._planets[i].setTexture(MyFrame._planets[i].texture_stripes);
      }
      MyFrame._rnd.texture_mapping = true;
      nuvole.setEnabled(true);
      prec.setEnabled(true);
      nuvole.setSelected(false);
      prec.setSelected(false);
      Sphere.c = false;
      for (int i = 0; i < MyFrame.numPlanets; i++) {
         MyFrame._planets[i].precessione = false;
      }

   }
});

Sphere.c = false;
nuvole.addActionListener(new ActionListener() { // nuvole ON-OFF

   @Override
   public void actionPerformed(ActionEvent event) {
      JCheckBox cb = (JCheckBox) event.getSource();
      if (cb.isSelected()) {
         Sphere.c = true;

      } else {
```

```
            Sphere.c = false;
         }
      }
   });

   prec.addActionListener(new ActionListener() { // precessione

      @Override
      public void actionPerformed(ActionEvent event) {
         JCheckBox cb = (JCheckBox) event.getSource();
         if (cb.isSelected()) {
            for (int i = 0; i < MyFrame.numPlanets; i++) {
               MyFrame._planets[i].precessione = true;
            }
         } else {
            for (int i = 0; i < MyFrame.numPlanets; i++) {
               MyFrame._planets[i].precessione = false;
            }
         }
      }
   });

   shading_opz1.addActionListener(new ActionListener() { // No z-shading

      @Override
      public void actionPerformed(ActionEvent event) {
         MyFrame._rnd.zShading = true;
      }
   });

   shading_opz2.addActionListener(new ActionListener() { // Phong Shading

      @Override
      public void actionPerformed(ActionEvent event) {
         MyFrame._rnd.zShading = false;
         for (int i = 0; i < MyFrame._scn.obj.size(); i++) {
            MyFrame._scn.obj.get(i).setBlinn(false);
         }
      }
   });
   shading_opz3.addActionListener(new ActionListener() { // Blinn Shading

      @Override
      public void actionPerformed(ActionEvent event) {
         MyFrame._rnd.zShading = false;
         for (int i = 0; i < MyFrame._scn.obj.size(); i++) {
            MyFrame._scn.obj.get(i).setBlinn(true);
         }
      }
```

```
});

gouraud_opz1.addActionListener(new ActionListener() { // Gouraud Shading OFF

   @Override
   public void actionPerformed(ActionEvent event) {
      MyFrame._rnd.gouraud = false;
   }
});

gouraud_opz2.addActionListener(new ActionListener() { // Gouraud Shading ON

   @Override
   public void actionPerformed(ActionEvent event) {
      MyFrame._rnd.gouraud = true;
   }
});

light_opz1.addActionListener(new ActionListener() { // 2nd Light OFF

   @Override
   public void actionPerformed(ActionEvent event) {
      intensity = 0;
      MyFrame._scn.lgt.get(0).setIntensity(intensity);
   }
});

light_opz2.addActionListener(new ActionListener() { // 2nd Light ON

   @Override
   public void actionPerformed(ActionEvent event) {
      intensity = 200;
      MyFrame._scn.lgt.get(0).setIntensity(intensity);
   }
});

integ_opz1.addActionListener(new ActionListener() { // Runge Kutta

   @Override
   public void actionPerformed(ActionEvent event) {

      Main.eu.kill();

      integ_opz1.setEnabled(false);
      integ_opz1.setSelected(true);

      Main.t = new Thread(Main.rk);
      Main.t.start();
```

```
            integ_opz2.setSelected(false);
            integ_opz2.setEnabled(true);
         }
      });

      integ_opz2.addActionListener(new ActionListener() { // Eulero

         @Override
         public void actionPerformed(ActionEvent event) {

            Main.rk.kill();

            integ_opz2.setEnabled(false);
            integ_opz2.setSelected(true);

            Main.t_eu = new Thread(Main.eu);
            Main.t_eu.start();

            integ_opz1.setSelected(false);
            integ_opz1.setEnabled(true);

         }
      });



   }

}

class OrthoProjections extends JPanel {// pannello che racchiude le 3 proiezioni
    ortogonali

   private static final long serialVersionUID = 1L;
   public OrthoPanels orthoX, orthoZ, orthoY;
   public Label pX, pY, pZ, blank1, blank2, blank3;
   public JPanel x = new JPanel(), y = new JPanel(), z = new JPanel();

   public OrthoProjections(int w, int h, Sphere[] pianeti) {

      setBorder(BorderFactory.createLineBorder(Color.black));
      setBounds(850, 80, 500, 600);
      int width = 250;
      int height = 300;
      blank1 = new Label(" ");
      blank2 = new Label(" ");
      blank3 = new Label(" ");

      this.setLayout(new GridLayout(3, 2));
```

```
        orthoX = new OrthoPanels(width, height, 'x', pianeti);
        orthoX.setBorder(BorderFactory.createRaisedBevelBorder());
        orthoX.setBorder(BorderFactory.createLoweredBevelBorder());
        x.setLayout(new BorderLayout());
        x.setBorder(BorderFactory.createLoweredBevelBorder());
        pX = new Label("Depth Axis: x");
        pX.setFont(new Font("SansSerif", Font.BOLD, 17));
        x.add(blank1, BorderLayout.WEST);
        x.add(pX, BorderLayout.CENTER);

        orthoZ = new OrthoPanels(width, height, 'z', pianeti);
        orthoZ.setBorder(BorderFactory.createRaisedBevelBorder());
        orthoZ.setBorder(BorderFactory.createLoweredBevelBorder());
        z.setLayout(new BorderLayout());
        z.setBorder(BorderFactory.createLoweredBevelBorder());
        pZ = new Label("Depth Axis: z");
        pZ.setFont(new Font("SansSerif", Font.BOLD, 17));
        z.add(blank2, BorderLayout.WEST);
        z.add(pZ, BorderLayout.CENTER);

        orthoY = new OrthoPanels(width, height, 'y', pianeti);
        orthoY.setBorder(BorderFactory.createRaisedBevelBorder());
        orthoY.setBorder(BorderFactory.createLoweredBevelBorder());
        y.setLayout(new BorderLayout());
        y.setBorder(BorderFactory.createLoweredBevelBorder());
        pY = new Label("Depth Axis: y");
        pY.setFont(new Font("SansSerif", Font.BOLD, 17));
        y.add(blank3, BorderLayout.WEST);
        y.add(pY, BorderLayout.CENTER);

        this.add(orthoX, 0, 0);
        this.add(x, 0, 1);
        this.add(orthoY, 1, 0);
        this.add(y, 1, 1);
        this.add(orthoZ, 2, 0);
        this.add(z, 2, 1);

    }

}

class HelpPanel extends JPanel { // pannello con instruzioni

    private static final long serialVersionUID = 1L;

    JLabel label1, label2, label3, label4, label5;

    public HelpPanel() {
```

```
      setBounds(850, 80, 500, 600);
      this.setLayout(new BorderLayout());
      setBorder(BorderFactory.createLineBorder(Color.black));

      Label etichetta = new Label("ISTRUZIONI");
      etichetta.setFont(new Font("SansSerif", Font.BOLD, 17));

      TextArea field = new TextArea();
      field.setText("\nPer cambiare le coordinate del centro e della velocita' dei
          vari pianeti, "
            + "\nscegliere il pianeta desiderato nel menu' a tendina e modificare
               le "
            + "\ncoordinate tramite gli sliders, " + " \n\nCon le freccette 'su',
               'giu', 'dx', 'sx' puoi"
            + " spostare la visuale e \ncon 'F' e 'B' puoi fare lo zoom \n\nIl
               sistema di riferimento preso in "
            + "considerazione e' quello centrato\nnella Terra, quindi in (0,0,1100)
               ");

      field.setFont(new Font("SansSerif", Font.PLAIN, 14));
      field.setEditable(false);

      add(etichetta, BorderLayout.NORTH);
      add(field, BorderLayout.CENTER);

   }

}

class PlanetsContainerPanel extends JPanel {// pannello container pianeti

   private static final long serialVersionUID = 1L;

   public PlanetsContainerPanel() {
      Label etichetta = new Label("PIANETI");
      etichetta.setFont(new Font("SansSerif", Font.BOLD, 17));

      setBorder(BorderFactory.createLineBorder(Color.black));
      setBounds(850, 80, 500, 600);

      add(etichetta);
   }
}

class PlanetPanel extends JPanel implements ChangeListener { // pannello per
    modificare i parametri dei pianeti

   private static final long serialVersionUID = 1L;
```

```
String[] planets = new String[] { "Luna", "Marte", "Venere", "Mercurio", "Giove"
    };
JComboBox<String> planetsCB = new JComboBox<>(planets);

int[] val = new int[] { 0, 0, 0, 0, 0, 0 };
int[] new_val = new int[] { 0, 0, 0, 0, 0, 0 };

JSlider disLx = new JSlider(-600, 600, 0);
JSlider disLy = new JSlider(-400, 400, 0);
JSlider disLz = new JSlider(-1000, 1000, 0);

JSlider dis2Lx = new JSlider(-9, 9, 0);
JSlider dis2Ly = new JSlider(-9, 9, 0);
JSlider dis2Lz = new JSlider(-9, 9, 0);

public PlanetPanel() {
   setBounds(880, 150, 450, 500);

   Label label = new Label("Scegli il pianeta da modificare");
   Label cX = new Label("centro X");
   Label cY = new Label("centro Y");
   Label cZ = new Label("centro Z");
   Label vX = new Label("velocita' X");
   Label vY = new Label("velocita' Y");
   Label vZ = new Label("velocita' Z");

   JPanel blank = new JPanel();
   blank.setPreferredSize(new Dimension(500, 50));
   JPanel blank1 = new JPanel();
   blank1.setPreferredSize(new Dimension(500, 50));
   JPanel blank2 = new JPanel();
   blank2.setPreferredSize(new Dimension(500, 50));
   JPanel blank3 = new JPanel();
   blank3.setPreferredSize(new Dimension(500, 50));
   JPanel blank4 = new JPanel();
   blank4.setPreferredSize(new Dimension(500, 50));
   JPanel blank5 = new JPanel();
   blank.setPreferredSize(new Dimension(500, 50));
   JPanel blank6 = new JPanel();
   blank.setPreferredSize(new Dimension(500, 50));

   blank.add(planetsCB);
   blank.add(label);
   add(blank);

   blank1.add(cX);
   blank1.add(disLx, "coordinata X del centro");
   add(blank1);
```

```
blank2.add(cY);
blank2.add(disLy, "coordinata Y del centro");
add(blank2);

blank3.add(cZ);
blank3.add(disLz, "coordinata Z del centro");
add(blank3);

blank4.add(vX);
blank4.add(dis2Lx, "coordinata X della velocita'");
add(blank4);

blank5.add(vY);
blank5.add(dis2Ly, "coordinata Y della velocita'");
add(blank5);

blank6.add(vZ);
blank6.add(dis2Lz, "coordinata Z della velocita'");
add(blank6);

label.setFont(new Font("Times New Roman", Font.BOLD, 13));
cX.setFont(new Font("Times New Roman", Font.BOLD, 13));
cY.setFont(new Font("Times New Roman", Font.BOLD, 13));
cZ.setFont(new Font("Times New Roman", Font.BOLD, 13));
vX.setFont(new Font("Times New Roman", Font.BOLD, 13));
vY.setFont(new Font("Times New Roman", Font.BOLD, 13));
vZ.setFont(new Font("Times New Roman", Font.BOLD, 13));

disLx.setMajorTickSpacing(100);// crea una tacca ogni 100 unita
disLx.setPaintTicks(true);// disegna tacche tra le varie unita
disLx.setPaintLabels(true);// scrive i valori sotto lo slider
disLx.setPreferredSize(new Dimension(300, 40));
disLx.addChangeListener(this);

disLy.setMajorTickSpacing(100);
disLy.setPaintTicks(true);
disLy.setPaintLabels(true);
disLy.setPreferredSize(new Dimension(300, 40));
disLy.addChangeListener(this);

disLz.setMajorTickSpacing(200);
disLz.setPaintTicks(true);
disLz.setPaintLabels(true);
disLz.setPreferredSize(new Dimension(300, 40));
disLz.addChangeListener(this);

dis2Lx.setMajorTickSpacing(1);
dis2Lx.setPaintTicks(true);
dis2Lx.setPaintLabels(true);
```

```
   dis2Lx.setPreferredSize(new Dimension(300, 40));
   dis2Lx.addChangeListener(this);

   dis2Ly.setMajorTickSpacing(1);
   dis2Ly.setPaintTicks(true);
   dis2Ly.setPaintLabels(true);
   dis2Ly.setPreferredSize(new Dimension(300, 40));
   dis2Ly.addChangeListener(this);

   dis2Lz.setMajorTickSpacing(1);
   dis2Lz.setPaintTicks(true);
   dis2Lz.setPaintLabels(true);
   dis2Lz.setPreferredSize(new Dimension(300, 40));
   dis2Lz.addChangeListener(this);

}

@Override
public void stateChanged(ChangeEvent arg0) {

   Point center_new=new Point();
   Point vel_new=new Point();

   String nome = (String) planetsCB.getSelectedItem();

   for (int i = 0; i < MyFrame.numPlanets; i++) {

      if (nome==(MyFrame._planets[i].nome)) {

         center_new.set(MyFrame._planets[i].center);
         vel_new.set(MyFrame._planets[i].v);

         if (disLx.getValue() != MyFrame._planets[i].center.x &&
             disLx.getValue() != val[0]) {
            center_new.x = disLx.getValue();
            val[0]=disLx.getValue();
         }
         if (disLy.getValue() != MyFrame._planets[i].center.y &&
             disLy.getValue() != val[1]) {
            center_new.y = disLy.getValue();
            val[1]=disLy.getValue();
         }
         if (disLz.getValue() != MyFrame._planets[i].center.z &&
             disLz.getValue() != val[2]) {
            center_new.z = disLz.getValue();
            val[2]=disLz.getValue();
         }
         if (dis2Lx.getValue() != MyFrame._planets[i].v.x && dis2Lx.getValue()
             != val[3]) {
```

```
            vel_new.x = dis2Lx.getValue();
            val[3]=dis2Lx.getValue();
          }
          if (dis2Ly.getValue() != MyFrame._planets[i].v.y && dis2Ly.getValue()
              != val[4]) {
            vel_new.y = dis2Ly.getValue();
            val[4]=dis2Ly.getValue();
          }
          if (dis2Lz.getValue() != MyFrame._planets[i].v.z && dis2Lz.getValue()
              != val[5]) {
            vel_new.z = dis2Lz.getValue();
            val[5]=dis2Lz.getValue();
          }

          MyFrame._planets[i].center.set(center_new);
          MyFrame._planets[i].v.set(vel_new);
        }
      }

    }
}

class MenuButtonAction implements ActionListener { // action listener dei bottoni
    del menu'

    public void actionPerformed(ActionEvent event) {

      if (event.getActionCommand().equals("Home")) {

        MyFrame.home_opz.setVisible(true);
        MyFrame.home.setVisible(true);
        MyFrame.o_pj.setVisible(false);
        MyFrame.help.setVisible(false);
        MyFrame.planet.setVisible(false);
        MyFrame.planetH.setVisible(false);

      } else if (event.getActionCommand().equals("OrthoPanels")) {

        MyFrame.home_opz.setVisible(false);
        MyFrame.home.setVisible(false);
        MyFrame.o_pj.setVisible(true);
        MyFrame.help.setVisible(false);
        MyFrame.planet.setVisible(false);
        MyFrame.planetH.setVisible(false);

      } else if (event.getActionCommand().equals("Help")) {

        MyFrame.home_opz.setVisible(false);
        MyFrame.home.setVisible(false);
```

```
         MyFrame.o_pj.setVisible(false);
         MyFrame.help.setVisible(true);
         MyFrame.planet.setVisible(false);
         MyFrame.planetH.setVisible(false);

      } else if (event.getActionCommand().equals("Pianeti")) {

         MyFrame.home_opz.setVisible(false);
         MyFrame.home.setVisible(false);
         MyFrame.o_pj.setVisible(false);
         MyFrame.help.setVisible(false);
         MyFrame.planet.setVisible(true);
         MyFrame.planetH.setVisible(true);

      } else if (event.getActionCommand().equals("Exit")) {

         if (!Main.rk.isDead)
            Main.rk.kill();
         if (!Main.eu.isDead)
            Main.eu.kill();
         System.out.println("end");
         System.exit(0);
      }
   }
}
```

Canvas.java:

```
/*
  La classe Canvas crea e gestisce il Canvas della finestra della vista
      prospettica 3D del movimento dei 5 pianeti
  intorno alla Terra, inclusi gli effetti di luce, le eclissi, la visualizzazione
      della rotazione delle tessiture.
 /*

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.image.BufferedImage;

import javax.swing.JPanel;

public class Canvas extends JPanel{

private static final long serialVersionUID = 1L;

/*
```

```
 Pannello della visione prospettica su cui viene calcolato lo Z-Buffer del
    RenderingEngine


*/
     public int width, height;
     public static int t;
     public BufferedImage image;
     public RenderingEngine rnd;

     public Canvas(int w, int h, RenderingEngine rnd) {
        super();
        width = w;
        height = h;
        this.setPreferredSize(new Dimension(width, height));
        this.setVisible(true);
        this.rnd=rnd;
     }

     @Override
     public void paintComponent(Graphics gr) {
        super.paintComponent(gr);

        image = rnd.render();

        gr.drawImage(image, 0, 0, null);
//      rnd.scn.lgt.get(0).rotateY(0);
//      rnd.scn.lgt.get(1).rotateY(0);
        if(rnd.texture_mapping)
        rnd.scn.lgt.get(1).rotateY(0.00595);
        rnd.scn.simulate();

        t += 1; // incremento la variabile temporale ad ogni frame
     }



}


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  \bigskip
\begin{lstlisting}
```

OrthoPanels.java:

```
/*
  La classe OrthoPanel crea e gestisce i JPanels delle viste del movimento dei
      pianeti, schematizzati come dischetti colorati,
  in prospettiva ortogonale sui tre piani coordinati.
 /*

  import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JPanel;

public class OrthoPanels extends JPanel {
      /*
        classe dei pannelli delle viste ortogonali. Simula una proiezione
            ortogonale, creando dei cerchi prendendo
        a due a due le coordinate dei pianeti, a seconda della vista scelta
        es: per la vista con asse di profondita' Z, prendo come coordinate del
            cerchio le coordinate x e y dei pianeti
       */
      private static final long serialVersionUID = 1L;
      public int width, height;
      public int[] x, y;
      public char a;
      Sphere[] o;

      public OrthoPanels(int w, int h, char axis, Sphere[] pianeti) {
         super();
         width = w;
         height = h;
         x = new int[6];
         y = new int[6];
         a = axis;
         o = pianeti;

         this.setPreferredSize(new Dimension(width, height));
         this.setVisible(true);

      }

      @Override
      public void paintComponent(Graphics g) {
         super.paintComponent(g);
         int offsetX = (int) this.getWidth() / 2;
         int offsetY = (int) this.getHeight() / 2;
         int zOffset = (int) o[0].center.z;
         switch (a) {
         case 'x':// zyx
            for (int i = 0; i < o.length; i++) {
```

```
            x[i] = widthProp((int) o[i].center.z) + offsetX - widthProp(zOffset);
            y[i] = heightProp((int) o[i].center.y) + offsetY;
         }

         break;
      case 'y':// xzy
         for (int i = 0; i < o.length; i++) {
            x[i] = widthProp((int) o[i].center.x) + offsetX;
            y[i] = heightProp((int) o[i].center.z) + offsetY -
                heightProp(zOffset);
         }
         break;
      case 'z':// xyz
         for (int i = 0; i < o.length; i++) {
            x[i] = widthProp((int) o[i].center.x) + offsetX;
            y[i] = heightProp((int) o[i].center.y) + offsetY;
         }
         break;
      default:// xyz
         for (int i = 0; i < o.length; i++) {
            x[i] = widthProp((int) o[i].center.x) + offsetX;
            y[i] = heightProp((int) o[i].center.y) + offsetY;
         }
         break;
      }

      for (int i = 0; i < o.length; i++) {
         int r = (int) o[i].size / 3;
         int r_2 = r / 2;
         g.setColor(new Color((float) o[i].color.x, (float) o[i].color.y,
             (float) o[i].color.z));
         g.fillOval(x[i] - r_2, y[i] - r_2, r, r);
         overlapEarth(i,a, g);

      }

   }

   public void overlapEarth(int i, char axis, Graphics g) {

      int r = (int) o[0].size / 3;
      int r_2 = r / 2;

      switch (axis) {
      case 'x':
         if (o[0].center.x < o[i].center.x) {
            g.setColor(new Color((float) o[0].color.x, (float) o[0].color.y,
                (float) o[0].color.z));
            g.fillOval(x[0] - r_2, y[0] - r_2, r, r);
```

```
            }
            break;

         case 'y':
            if (o[0].center.y < o[i].center.y) {
               g.setColor(new Color((float) o[0].color.x, (float) o[0].color.y,
                   (float) o[0].color.z));
               g.fillOval(x[0] - r_2, y[0] - r_2, r, r);
            }
            break;

         case 'z':
            if (o[0].center.z < o[i].center.z) {
               g.setColor(new Color((float) o[0].color.x, (float) o[0].color.y,
                   (float) o[0].color.z));
               g.fillOval(x[0] - r_2, y[0] - r_2, r, r);
            }
            break;

         default:
            if (o[0].center.z < o[i].center.z) {
               g.setColor(new Color((float) o[0].color.x, (float) o[0].color.y,
                   (float) o[0].color.z));
               g.fillOval(x[0] - r_2, y[0] - r_2, r, r);
            }
            break;
         }

      }

      public int widthProp(int x) {
         int x_new;
         x_new = (x * width) / (width * 3);
         x_new /= 2;
         return x_new;
      }

      public int heightProp(int y) {
         int y_new;
         y_new = (y * height) / (height * 3);
         y_new /= 2;
         return y_new;
      }
   }
```

Scene.java:

```
/*
  La classe Scene crea le liste di luci e di sfere dei pianeti, ed anima i pianeti
     richiamando il metodo simulate.
  /*
import java.util.ArrayList;

public class Scene {

   /*
      classe che crea tutti gli elementi della scena (luci, pianeti)
   */

   public ArrayList<Light> lgt; // Lista delle luci nella scena
   public double amb; // Colore della luce ambientale
   public ArrayList<Sphere> obj;

   public Scene(double ambientLight) {
      amb = ambientLight;
      lgt = new ArrayList<Light>();
      obj = new ArrayList<Sphere>();
   }

   public Sphere get(int i) {
      return obj.get(i);
   }

   public void add(Sphere o) {
      obj.add(o);
   }

   public void addLight(Light l) {
      lgt.add(l);
   }

   public int size() {
      return obj.size();
   }

   public Sphere addSphere(Sphere pianeta) {

      this.add(pianeta);
      return this.get(this.size() - 1);
   }

   /*
      Scene.simulate() e' un metodo fondamentale per il programma
      Per ogni oggetto/pianeta nella scena, richiama il metodo Obj.simulate(), che
         richiama,
```

```
      a sua volta, il metodo Obj.shift(Point v), grazie a cui e' possibile animare
          i pianeti.
      Quest'ultimo metodo opera una traslazione su tutti i punti della mesh,
          prendendo come offset
      le coordinate del vettore velocita'
    */

   public void simulate() {
      for (int i = 0; i < obj.size(); i++) {
         obj.get(i).simulate();
      }
   }


}
```

Scene.java:

```
/*
  La classe PhysicsEngineEuler gestisce la dinamica dei pianeti, integrando sia le
      equazioni del moto dei baricentri sia quelle del corpo rigido tramite il
      metodo di Eulero.
 */
public class PhysicsEngineEuler implements Runnable {

   static final int numPlanets = 6;
   static final double timestep = 0.5;
   static final double g = 23;
   Point cdm;
   public double mTot = 0;
   static double vp = -0.005;
   public static double a;
   public static Point Fg;
   public static Point vel;
   public static Point L0;
   public static Point L1;
   public static Point tau;
   public static Point[] poloNord;
   long millisOLD, millisNEW;
   public Sphere[] pianeti;
   public static Sphere[] tmp;
   static double rTest = 35;
   private boolean isRunning = true;
   public boolean isDead = false;
   public static double verse;
   public static Point[] fg_eu = new Point[numPlanets];
   public static Point[] acc2Eu = new Point[2];
```

```
public PhysicsEngineEuler(Sphere[] planets, Sphere[] planets_tmp) {

   super();
   verse = 1.0;
   a = 0.0;
   Fg = new Point(0.0);
   vel = new Point(0.0);
   L1 = new Point(0.0);
   L0 = new Point(0.0);
   tau = new Point(0.0);

   for (int i = 0; i < numPlanets; i++) {
      fg_eu[i] = new Point(0.0);
   }

   for (int i = 0; i < 2; i++) {
      acc2Eu[i] = new Point(0.0);
   }

   //cdm = new Point();

   pianeti = planets;
   tmp=planets_tmp;
   poloNord = new Point[numPlanets];
   for (int i = 0; i < numPlanets; i++) {
      poloNord[i] = new Point(0.0);

      poloNord[i].set(pianeti[i].center);
      poloNord[i].y += pianeti[i].size;
      poloNord[i].rotateY(Main.aPrecStart[i]);
   }

}

////////////////////////////// attrazione di gravita
public void eu(Sphere[] pianeti) {

   /*
    * Algoritmo di integrazione numerica: Eulero. L'algoritmo in questione e' un
    * algoritmo che si occupa della risoluzione di equazioni differenziali di
      primo
    * grado.
    *
    * Lo pseudo-codice seguente illustra come risolvere il problema della forza
      di
    * attrazione gravitazionale, che e' di secondo grado, dividendo il problema
      in
    * due equazioni differenziali di primo grado, a cui viene applicato il metodo
```

```
 * di Eulero.
 *
 * v.new = v.old + a * dt pos.new = pos.old + v * dt
 *
 * Per la simulazione nel nostro programma, e' sufficiente calcolare
    l'equazione
 * relativa alla velocita', per poi shiftare tutti i punti della mesh
    prendendo
 * il vettore velocita' come offset.
 *
 * L'algoritmo di Eulero e' un metodo meno preciso di quello di Runge Kutta al
 * quarto ordine e cio' comporta una propagazione degli errori molto piu'
 * significativa, portando i calcoli a divergere e facendo "fuggire" i pianeti
 * dall'orbita in molto meno tempo.
 *
 */

for (int i = 0; i < numPlanets; i++) {
   fg_eu[i] = new Point(0.0);
}

for (int i = 0; i < 2; i++) {
   acc2Eu[i] = new Point(0.0);
}

for (int i = 0; i < numPlanets - 1; i++) {
   for (int j = i + 1; j < numPlanets; j++) {

      acc2Eu = accFg(pianeti[i], pianeti[j]); // attrazione di gravita' con
         eulero
      fg_eu[i].add(acc2Eu[0].per(pianeti[i].mass));
      fg_eu[j].add(acc2Eu[1].per(pianeti[j].mass));

   }
}



if (pianeti[0].precessione) { //Precessione
   /*
    la precessione e' approssimata tramite una rotazione della texture in
       direzione orizzontale rispetto
    al suo asse di rotazione, sommata ad una composizione di 2 rotazioni che
       ruotano il pianeta nella
    posizione del nuovo momento angolare
    */
   for (int i = 0; i < numPlanets; i++) {
      if (i != 0) {
         precession(i, pianeti, fg_eu[i].per(pianeti[i].mass));
```

```
            }
         }
      }

      for (int i = 0; i < numPlanets; i++) {
         pianeti[i].rotateTexture(pianeti[i].texture_vel);
      }
   }

   public static Point[] accFg(Sphere i, Sphere j) {
      /*
       metodo per il calcolo della forza di attrazione gravitazionale, in cui i
           pianeti sono presi in
       considerazione a due a due
        */
      Point[] acc2 = new Point[2];

      for (int a = 0; a < 2; a++) {
         acc2[a] = new Point(0.0);
      }
      Point i2j = i.center.to(j.center);
      double i2j_n = i2j.normalize();
      double verse = 1.0;

      if (i2j_n < rTest)
         return acc2;
      double denom = 1.0 / (g * i2j_n * i2j_n);

      acc2[0] = i2j.per(timestep * verse * denom * j.mass);
      acc2[1] = i2j.per(timestep * -verse * denom * i.mass);

      i.v.add(acc2[0]);
      j.v.add(acc2[1]);

      return acc2;
   }

   ////////////////////////////////// precessione

   public static void precession(int i, Sphere[] pianeti, Point _Fg) {

      /*
         Per approssimare il calcolo della precessione assiale dei pianeti, mi
             affido alla teoria del giroscopio.
         Le quantita' che vado a calcolare sono: momento angolare e momento
             torcente o coppia del pianeta.
         L'equazione del momento angolare L e' L=rXp, con r braccio del momento
             angolare e p=mv, quantita' di moto,
```

```
      mentre l'equazione del momento torcente t (tau) e' t=rxF, con r braccio
         del momento e F la forza, in questo,
      caso, di attrazione gravitazionale.
      Per calcolare il momento angolare, approssimo il braccio del momento con
         il vettore che congiunge i due
      poli del pianeta e calcolo la quantita' di moto del pianeta come prodotto
         della massa del pianeta per la sua
      velocita' e vado a calcolare il prodotto vettoriale tra questi due vettori.
      Per quanto riguarda il momento torcente, vado a calcolare il prodotto
         vettoriale del braccio del momento,
      approssimato come sopra, con il vettore della forza di attrazione
         gravitazionale, in questo caso definito
      come la somma vettoriale di tutte le forze di attrazione gravitazionale
         subite dal pianeta, rispetto a tutti
      gli altri.
      Infine, vado a calcolare il mio nuovo momento angolare con l'algoritmo di
         Eulero, sfruttando l'equazione
      tau=dL/dt in questo modo: L1 = L0+t*tau, con t timestep temporale.
      In questo modo, ricavo l'angolo di precessione dell'asse facendo
         l'arcocoseno del prodotto scalare tra L0 e L1,
      i momenti angolari prima e dopo il calcolo.
      La precessione e' poi simulata con la rotazione del pianeta rispetto
         all'angolo di precessione, accompagnata
      con lo scorrimento della texture sul pianeta.

    */

    double _t = 1;
    Fg = _Fg;
    L0 = angularMomentum(i, pianeti);
    tau = torque(i, Fg);

    tau = tau.per(_t);
    L1.x = L0.x + tau.x;
    L1.y = L0.y + tau.y;
    L1.z = L0.z + tau.z;

    poloNord(i, L0, L1);

    if (i != 0) {

      pianeti[i].rotateObj(L1, pianeti[i].center, tmp[i]);

    }

}

public static Point torque(int i, Point f) {//momento torcente o coppia
```

```
      Point t = new Point(0.0);

      t = (poloNord[i].normalized()).vector((f.normalized()));

      return t;

   }

   public static Point angularMomentum(int i, Sphere[] pianeti) {//momento angolare

      Point L = new Point(0.0);
      Point p = new Point(0.0);
      p = pianeti[i].v.per(pianeti[i].mass);
      L = (poloNord[i].normalized()).vector((p.normalized()));

      return L;
   }


   public static void poloNord(int i, Point L0, Point L1) {//braccio dei momenti

      Point dL = new Point(0.0);

      dL.x = L1.x - L0.x;
      dL.y = L1.y - L0.y;
      dL.z = L1.z - L0.z;

      poloNord[i].x += dL.x;
      poloNord[i].y += dL.y;
      poloNord[i].z += dL.z;

   }

   public void kill() {

      isRunning = false;
      isDead = true;
      Thread.currentThread().interrupt();
   }

   public void startEngine() {
      System.out.println("Eu");
      isDead = false;
      while (isRunning) {
         eu(pianeti);

         try {
            Thread.sleep(85);
         } catch (InterruptedException e) {
```

```
        }
      }
      Thread.currentThread().interrupt();
      isRunning = true;
      isDead = true;

   }

   @Override
   public void run() {
      this.startEngine();
   }

}
```

Scene.java:

```
/*
  La classe PhysicsEngineRK gestisce la dinamica dei pianeti, integrando sia le
      equazioni del moto dei baricentri sia quelle del corpo rigido tramite il
      metodo di Runge-Kutta.
 */
public class PhysicsEngineRK implements Runnable {

   static final int numPlanets = 6;
   static final double timestep = 0.0003;
   static final double g = 57.5;
   Point cdm;
   double mTot = 0;

   double half_step = timestep / 2;
   double sixth_step = timestep / 6;
   long millisOLD, millisNEW;
   static Point kr1, kr2, kr3, kr4, kv1, kv2, kv3, kv4, c1, c2, c3, c4, pos, vel;
   public Sphere[] pianeti;
   public static Sphere[] tmp;
   static double rTest = 35;
   private boolean isRunning = true;
   public boolean isDead = false;
   public static Point[] poloNord;
   static double a, phi = 0;
   static double vp = -0.05;
   static Point L0, L1, tau, Fg;
   public static double verse;

   public PhysicsEngineRK(Sphere[] planets, Sphere[] planets_tmp) {
```

```
      super();
      pianeti = planets;
      tmp=planets_tmp;
      a = 0.0;
      Fg = new Point(0.0);
      vel = new Point(0.0);
      L1 = new Point(0.0);
      L0 = new Point(0.0);
      tau = new Point(0.0);
      poloNord = new Point[numPlanets];
      for (int i = 0; i < numPlanets; i++) {
         poloNord[i] = new Point(0.0);

         poloNord[i].set(pianeti[i].center);
         poloNord[i].y += pianeti[i].size;
         poloNord[i].rotateY(Main.aPrecStart[i]);
      }

   }

   ////////////////////////////////////// attrazione di gravita

   public void rk4(Sphere[] pianeti) {

      mTot = 0;
      cdm = new Point();

      //
      for (int i = 0; i < numPlanets; i++) {
         /*
          * Runge Kutta al 4 ordine per la risoluzione di equazioni differenziali
              del 1o
          * ordine. Forza di Attrazione Gravitazionale -> equazione differenziale
              del 2o
          * ordine Divido il problema nella risoluzione di 2 equazioni
              differenziali del
          * 1st ordine concatenate r' = v && v' = a le condizioni iniziali del
              sistema
          * sono i parametri: centri e velocita' dei corpi
          */

         /*
          Algoritmo di integrazione numerica: Runge Kutta del quarto ordine.
          L'algoritmo in questione e' un algoritmo iterativo in 4 steps che si
              occupa della risoluzione
          di equazioni differenziali di primo grado.
          Per risolvere il problema della forza di attrazione gravitazionale, che
              risulta essere un
```

```
 problema di equazioni differenziali di secondo grado, con l'algoritmo di
    runge kutta,
 devo considerare un sistema di 2 equazioni differenziali di primo grado,
    da calcolare in maniera
 sequenziale.
 Essendo l'equazione della forza F=ma, le mie equazioni di primo grado del
    sistema sono:
 a=(v)'=dv/dt e v=(r)'=dx/dt
 Per risolvere le equazioni differenziali, necessito di condizioni
    iniziali, che, in questo caso,
 saranno i centri e le velocita' dei corpi dati all'inizio del programma

 */

kr1 = new Point();
kr2 = new Point();
kr3 = new Point();
kr4 = new Point();
kv1 = new Point();
kv2 = new Point();
kv3 = new Point();
kv4 = new Point();
c1 = new Point();
c2 = new Point();
c3 = new Point();
c4 = new Point();
pos = new Point();
vel = new Point();

c1 = pianeti[i].center;

for (int iter = 0; iter < 10; iter++) {

   // K1

   // calcolo l'accelerazione
   kv1 = acc(pianeti[i], pianeti, c1);

   // copio le velocita' correnti
   kr1 = pianeti[i].v;
   // uso le velocita per predirre le posizione ad un halfstep nel futuro
   c2 = c1.addP(kv1.per(half_step));

   // K2

   // calcolo l'accelerazione
   kv2 = acc(pianeti[i], pianeti, c2);
   // Uso le velocita' per predirre le posizioni mezzo step nel futuro
   kr2 = pianeti[i].v.addP(kr1.per(half_step));
```

```
            c3 = c1.addP(kv2.per(half_step));

            // K3

            // calcolo l'accelerazione
            kv3 = acc(pianeti[i], pianeti, c3);
            // Uso le velocita' per predirre le posizioni mezzo step nel futuro
            kr3 = pianeti[i].v.addP(kr2.per(half_step));
            c4 = c1.addP(kv3.per(timestep));

            // K4

            // calcolo l'accelerazione
            kv4 = acc(pianeti[i], pianeti, c4);
            kr4 = pianeti[i].v.addP(kr3.per(timestep));

            // A questo punto, abbiamo 4 velocita' e 4 accelerazioni per ogni corpo
            // Ora combiniamo questi valori per predirre il risultato
            vel.x += sixth_step * (kv1.x + 2 * kv2.x + 2 * kv3.x + kv4.x);
            vel.y += sixth_step * (kv1.y + 2 * kv2.y + 2 * kv3.y + kv4.y);
            vel.z += sixth_step * (kv1.z + 2 * kv2.z + 2 * kv3.z + kv4.z);

            pos.x += sixth_step * (kr1.x + 2 * kr2.x + 2 * kr3.x + kr4.x);
            pos.y += sixth_step * (kr1.y + 2 * kr2.y + 2 * kr3.y + kr4.y);
            pos.z += sixth_step * (kr1.z + 2 * kr2.z + 2 * kr3.z + kr4.z);

        }

        // aggiorno i valori dei pianeti
        pianeti[i].v.add(vel);
        pianeti[i].center.add(pos);

        if (pianeti[0].precessione ) {//Precessione

            /*
             la precessione e' approssimata tramite una rotazione della texture in
                 direzione orizzontale rispetto
             al suo asse di rotazione, sommata ad una composizione di 2 rotazioni
                 che ruotano il pianeta nella
             posizione del nuovo momento angolare
             */
            if(i!=0) {
            precession(i, pianeti, kv4.per(pianeti[i].mass));
            }

        }
        pianeti[i].rotateTexture(pianeti[i].texture_vel);
    }
```

```
}

public static Point acc(Sphere o, Sphere bodies[], Point p) { //metodo per il
    calcolo dell'accelerazione di gravita'
   Point acc = new Point(0.0);                // per un singolo pianeta,
       rispetto a tutti gli altri
   Point i2j = new Point();
   double r, d;

   for (int j = 0; j < 1; j++) {
      if (o.nome.equals(bodies[j].nome)) {
         continue;
      } else {
         i2j = bodies[j].center.to(p);
         r = i2j.norm();
         if (r < rTest) {
            acc.set(0.0, 0.0, 0.0);
            return acc;
         }

         d = -1.0 / (g * r * r);

         acc.x += d * bodies[j].mass * i2j.x;
         acc.y += d * bodies[j].mass * i2j.y;
         acc.z += d * bodies[j].mass * i2j.z;
      }
   }

   return acc;
}

// ////////////////////////////////// precessione
//

public static void precession(int i, Sphere[] pianeti, Point _Fg) {

   /*
      Per approssimare il calcolo della precessione assiale dei pianeti, mi
          affido alla teoria del giroscopio.
      Le quantita' che vado a calcolare sono: momento angolare e momento
          torcente o coppia del pianeta.
      L'equazione del momento angolare L e' L=rXp, con r braccio del momento
          angolare e p=mv, quantita' di moto,
      mentre l'equazione del momento torcente t (tau) e' t=rxF, con r braccio
          del momento e F la forza, in questo,
      caso, di attrazione gravitazionale.
      Per calcolare il momento angolare, approssimo il braccio del momento con
          il vettore che congiunge i due
```

```
        poli del pianeta e calcolo la quantita' di moto del pianeta come prodotto
            della massa del pianeta per la sua
        velocita' e vado a calcolare il prodotto vettoriale tra questi due vettori.
        Per quanto riguarda il momento torcente, vado a calcolare il prodotto
            vettoriale del braccio del momento,
        approssimato come sopra, con il vettore della forza di attrazione
            gravitazionale, in questo caso definito
        come la somma vettoriale di tutte le forze di attrazione gravitazionale
            subite dal pianeta, rispetto a tutti
        gli altri.
        Infine, vado a calcolare il mio nuovo momento angolare con l'algoritmo di
            Eulero, sfruttando l'equazione
        tau=dL/dt in questo modo: L1 = L0+t*tau, con t timestep temporale.
        In questo modo, ricavo l'angolo di precessione dell'asse facendo
            l'arcocoseno del prodotto scalare tra L0 e L1,
        i momenti angolari prima e dopo il calcolo.
        La precessione e' poi simulata con la rotazione del pianeta rispetto
            all'angolo di precessione, accompagnata
        con lo scorrimento della texture sul pianeta.

     */

    double _t = 1;
    Fg = _Fg;
    L0 = angularMomentum(i, pianeti);
    tau = torque(i, Fg);

    tau = tau.per(_t);
    L1.x = L0.x + tau.x;
    L1.y = L0.y + tau.y;
    L1.z = L0.z + tau.z;

    poloNord(i, L0, L1);

    if (i != 0) {


       pianeti[i].rotateObj(L1, pianeti[i].center, tmp[i]);
    // pianeti[i].rotate_Obj(L0, L1, pianeti[i].center);
    }

  }

  public static Point torque(int i, Point f) {//momento torcente o coppia

    Point t = new Point(0.0);

    t = (poloNord[i].normalized()).vector((f.normalized()));
```

```
      return t;

   }

   public static Point angularMomentum(int i, Sphere[] pianeti) {//momento angolare

      Point L = new Point(0.0);
      Point p = new Point(0.0);
      p = pianeti[i].v.per(pianeti[i].mass);
      L = (poloNord[i].normalized()).vector((p.normalized()));

      return L;
   }


   public static void poloNord(int i, Point L0, Point L1) {//braccio dei momenti

      Point dL = new Point(0.0);

      dL.x = L1.x - L0.x;
      dL.y = L1.y - L0.y;
      dL.z = L1.z - L0.z;

      poloNord[i].x += dL.x;
      poloNord[i].y += dL.y;
      poloNord[i].z += dL.z;
//    poloNord[i].x = L0.x + dL.x;
//    poloNord[i].y = L0.y + dL.y;
//    poloNord[i].z = L0.z + dL.z;

   }

   public void kill() {

      isRunning = false;
      isDead = true;
      Thread.currentThread().interrupt();
   }

   public void startEngine() {
      System.out.println("RK");
      isDead = false;
      while (isRunning) {
         rk4(pianeti);

         try {
            Thread.sleep(60);
         } catch (InterruptedException e) {
         }
```

```
      }
      Thread.currentThread().interrupt();
      isRunning = true;
      isDead = true;
   }

   @Override
   public void run() {

      this.startEngine();
   }

}
```

RenderingEngine.java:

```
/*
  La classe RenderingEngine gestisce il rendering 3D, ottenuto dal metodo di
     z-Buffer modificato per recuperare le
  coordinate spaziali da cui proviene ciascun punto interpolato di un triangolo
     proiettato centralmente sul piano di
  visuale, al fine di utilizzarle per tracciare un raggio d'ombra ed il
     corrispondente contributo di illuminazione di Lambert
  e di Phong-Blinn, dopo aver applicato al punto una tessitura animata;
     l'illuminazione viene poi interpolata con il metodo
  di Gouraud.
 */


  import java.util.*;
import java.awt.image.BufferedImage;


public class RenderingEngine{
   public BufferedImage image;
   public Triangle[][] itemBuffer;
   public double[][] zBuffer;
   public int f, w, h, w_2, h_2;
   public Point p1, p2, p3, p4,left,right;
   public Point p[] = new Point[4];
   public Point camera, ux, uy, u0;
   public Scene scn;
   public double kDif, kRef;
   public ArrayList<Point> fragments;
   public Sphere object;
   public Point color;
   public boolean texture_mapping, gouraud, zShading = false;
```

```
public int reflection_exponent;
double u = 0.001;

public RenderingEngine(int width, int height, int focal, Point camera_, Point
    normal, Scene scn_){
   //Costruttore del renderer
   w = width;
   h = height;
   f = focal;
   w_2 = w/2;
   h_2 = h/2;
   scn = scn_;
   camera = camera_;
   camera.normal = normal;
   camera.normal.normalize();
   u0 = camera.sum(camera.normal.per(f));
   uy = new Point(0, camera.normal.z, -camera.normal.y);
   uy.normalize();
   ux = uy.vect(camera.normal);
   image = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
   zBuffer = new double[w][h];
   itemBuffer = new Triangle[w][h];
   fragments = new ArrayList<Point>();
   texture_mapping = true;
   gouraud = true;
}

public void setFocal(int focal) {
   f = focal;
   u0 = camera.sum(camera.normal.per(f));
}

public int getFocal() {
   return f;
}

public BufferedImage render(){
   // render() esegue tutte le funzioni necessarie a fere il rendering della
       scena

   // inizializzo la matrice dei pixel con il colore dello sfondo
   // e quella dello z-buffer con la profondita massima
   for(int i=0; i<w; i++)
      for(int j=0; j<h; j++){
         image.setRGB(i, j, 40);
         zBuffer[i][j] = Double.POSITIVE_INFINITY;
      }
```

```
      // per ogni triangolo della scena, calcolo il suo rendering
      Point vertex, frag;
      for(int i=0; i<scn.obj.size(); i++){
         object = scn.obj.get(i);
         for(int j=0; j<object.pts.size(); j++){
            vertex = object.pts.get(j);
            frag = vertex.project(camera, f);
            // salvo i valori di illuminazione di ogni vertice nel suo proiettato
                (fragment)
            setIllumination(vertex, frag);
            fragments.add( frag );
         }
         render(object);
         fragments.clear();
      }

      return image;
   }

   public void render(Sphere obj){
      kDif = object.kDif;
      kRef = object.kRef;

      if(!texture_mapping || obj.texture == null)
         color = obj.color;

      for (int i=0; i<obj.trn.size(); i++)
         render(obj.trn.get(i));
   }

   public void render(Triangle tr){
      // Questa funzione trova la proiezione del triangolo tr sul viewplane
      // e aggiorna i valori delle matrici image e zBuffer

      Point n = tr.getNormal(); // normale del triangolo tr
      ////////////////////////////////////////////////////////////////
      // backface culling
      //if( n.dot(tr.a.min(camera)) > 0 ) return;
      ////////////////////////////////////////////////////////////////
      /*if( n.dot(camera.normal) > 0 ) return;*/

      // carico in p[i] il vertici del triangolo, gia' proiettati sul view port
          (fragments)
      p[1] = fragments.get(tr.a.id);
      p[2] = fragments.get(tr.b.id);
      p[3] = fragments.get(tr.c.id);


      // trasformo i punti nelle coordinate relativa alla camera
```

```
      toCameraSpace(p[1]);
      toCameraSpace(p[2]);
      toCameraSpace(p[3]);

////////////////////////////////////////////////////
      // frustum culling
      /*if(
         (Math.abs(p[1].x)>w_2 || Math.abs(p[1].y)>h_2) &&
         (Math.abs(p[2].x)>w_2 || Math.abs(p[2].y)>h_2) &&
         (Math.abs(p[3].x)>w_2 || Math.abs(p[3].y)>h_2)
      ) return;*/

////////////////////////////////////////////////////
      // eseguo scambi per far si che p[1] sia il vertice piu' in alto sul
          viewplane (coordinata y piu bassa),
      // p[2] al centro e p[3] sia il piu basso
      if(p[1].y>p[2].y) swap(1,2);
      if(p[1].y>p[3].y) swap(1,3);
      if(p[2].y>p[3].y) swap(2,3);


      double zSlopeX = -n.x / n.z; // e' l'incremento di z per ogni spostamento di
          1 pixel lungo l'asse x
      double zSlopeY = -n.y / n.z; // e' l'incremento di z per ogni spostamento di
          1 pixel lungo l'asse y

      /*
         Adesso occorre fare un clipping del triangolo proiettato. Infatti la
             funzione scan() che disegna
         il triangolo su image funziona solo se uno dei lati e' parallelo all'asse
             x del view plane.
         Si opera dunque un taglio orizzionatle all'altezza del vertice p[2]
             ottentedo due triangoli (uno
         superiore e uno inferiore), entrambi con un lato orizzontale (in comune)
      */

      // Caso banale: se due vertici hanno la stessa y, allora un lato e'
          orizzontale. Posso dunque eseguire
      // direttamente la funzione scan() e disegnare l'intero triangolo

      if(p[1].y == p[2].y){
         if(p[1].x>p[2].x) swap(1,2);
         scan(p[3], p[1], p[2], -1, zSlopeX, zSlopeY, tr);
         return;
      }

      // Nel caso generale devo trovare l'intersezione del triangolo proiettato sul
          viewplane con
```

```
    // la retta orizzontale passante per p[2]. Un punto d'interzezione e'
       ovviamente p[2], l'altro
    // si trova sul lato opposto e si ottiene tramite interpolazione lineare

    // coefficiente di interpolazione lineare
    double cc = (p[2].y - p[3].y) / (p[1].y - p[3].y);
    p[0] = Point.getInterpolation(cc, p[1], 1-cc, p[3]);

    // Ora bisogna disegnare i due triangoli proiettati sul viewplane (ora
       entrambi
      // con un lato orizzontale).
    // Opero degli scambi affinche' p[1],p[2],p[0] sia il triangolo superiore e
    // p[3],p[2],p[0] quello inferiore
    if(p[2].x>p[0].x) swap(2,0);

    // si possono ora disegnare le due porzioni di triangolo con la funzione
       scan()
    scan(p[1], p[2], p[0], 1, zSlopeX, zSlopeY, tr);
    scan(p[3], p[2], p[0], -1, zSlopeX, zSlopeY, tr);
  }


  public void scan(Point p1, Point p2, Point p3, int verse, double zSlopeX, double
     zSlopeY, Triangle tr){
    /*
      scan() trova, con un procedimento incrementale, tutti i pixel interni del
         triangolo p1,p2,p3 e per ognuno
      di essi richiama draw(), la funzione che aggiorna di fatto le matrici
         image e zBuffer.
      Il valore 'v' indica se si sta operando lo scan di un triangolo superiore
         (+1) o inferiore (-1) del clipping
    */

    //  Se si considera la retta che passa lungo il lato sinistro del triangolo,
    //    ls rappresente l'incremento di x per ogni spostamento di 1 pixel lungo
       l'asse y del view plane
    // (dunque l'inverso del coefficiente angolare)
    double ls = (p1.x - p2.x) / (p1.y - p2.y);

    // rs rappresenta la stessa quantita' per il lato destro del triangolo
    double rs = (p1.x - p3.x) / (p1.y - p3.y);

    double x, y, z, xl, xr, zx, c1, c2, c3, u, v, width;
    //int i, j;

    double diffuse, specular;
    int clr;
    //double p1u, p2u, p3u,p1v, p2v, p3v;
```

```
// Nel caso in cui v=+1, si sta facendo lo scan del triangolo superiore,
    dall'alto verso il basso
// (dunque dal vertice in alto verso il lato orizzontale)

double inv_height = 1.0/Math.abs(p2.y - p1.y);
if(verse==1){
   c1 = 1.0;
   z = p1.z;
   xl = xr = p1.x;
   for(y=p1.y; y<=p3.y; y++){
      // questo for() viene eseguito per ogni riga orizzonatale del triangolo
      zx = z;
      c2 = 1 - c1;
      width = xr - xl;
      for(x=xl; x<=xr; x++){
         //questo for() viene eseguito per ogni pixel della stessa riga
         // con draw(), scrivo sulla matrice di pixel il valore del colore c
             nel punto x,y del viweplane
         // e aggiorno in tali coordinate anche il valore dello zBuffer
         c3 = 1 - c1 - c2;
         u = c1*p1.u + c2*p2.u + c3*p3.u;
         v = c1*p1.v + c2*p2.v + c3*p3.v;
         if(gouraud){
            diffuse = c1*p1.diffuse + c2*p2.diffuse + (1-c1-c2)*p3.diffuse;
            specular = c1*p1.specular + c2*p2.specular + (1-c1-c2)*p3.specular;
         }
         else{
            diffuse = (p1.diffuse + p2.diffuse + p3.diffuse)/3.0;
            specular = (p1.specular + p2.specular + p3.specular)/3.0;
         }
         clr = getShading(u, v, diffuse, specular).toRGB();
         if(zShading) clr = new Point(-0.3+1000.0/zx).toRGB();

         draw(x, y, zx, clr,tr);

         // ottengo il nuovo valore di profondita' per la prossima iterazione
             (pixel di destra)
         zx += zSlopeX;
         c2 -= (1-c1)/width;
      }
      z += ls*zSlopeX + zSlopeY; // ottengo il nuovo valore di profondita'
          per la prossima iterazione (riga sottostante)
      xl += ls; // ottengo l'estremo sinistro dei valori di x per la prossima
          riga
      xr += rs; // ottengo l'estremo destro dei valori di x per la prossima
          riga
      // xl e xr sono gli estremi della riga per il ciclo interno, sono usati
          nella condizione
      // di permanenza del ciclo for() interno
```

```
            c1 -= inv_height;
         }
      }

      // Nel caso in cui v=-1, si sta facendo lo scan del triangolo inferiore,
          dall'alto verso il basso
      // (dunque dal lato orizzontale verso il vertice in basso)
      // le operazioni sono analoghe
      if(verse==-1){
         c1 = 0.0;
         z = p3.z;
         xl = p2.x;
         xr = p3.x;
         for(y=p3.y; y<=p1.y; y++){
            zx = z;
            c2 = 1 - c1;
            width = xr - xl;
            for(x=xl; x<=xr; x++){

               c3 = 1 - c1 - c2;
               u = c1*p1.u + c2*p2.u + c3*p3.u;
               v = c1*p1.v + c2*p2.v + c3*p3.v;
               if(gouraud){
                  diffuse = c1*p1.diffuse + c2*p2.diffuse + (1-c1-c2)*p3.diffuse;
                  specular = c1*p1.specular + c2*p2.specular + (1-c1-c2)*p3.specular;
               }
               else{
                  diffuse = (p1.diffuse + p2.diffuse + p3.diffuse)/3.0;
                  specular = (p1.specular + p2.specular + p3.specular)/3.0;
               }
               clr = getShading(u, v, diffuse, specular).toRGB();
               if(zShading) clr = new Point(-0.3+1000.0/zx).toRGB();

               draw(x, y, zx, clr,tr);
               zx += zSlopeX;
               c2 -= (1-c1)/width;
            }
            z += ls*zSlopeX + zSlopeY;
            xl += ls;
            xr += rs;
            c1 += inv_height;
         }
      }
   }

   public void draw(double xs, double ys, double z, int c, Triangle tr){
      // Se necessario, la funzione draw() aggiorna image con il colore c nel punto
          (xs,ys)
      // e inoltre aggiorna lo zBuffer nello stesso punto con il valore z
```

```
   int x, y;
   // passo dalle coordinate dello spazio a quelle dello schermo (in pixel)
   x = round(xs + w_2);
   y = round(ys + h_2);

   // Se le coordinate (x,y) del pixel non sono contenute nel viewplane, esco da
       draw()
   if( !(x<w && x>=0 && y<h && y>=0) )
      return;


   /* Se la coordinata z di profondita' e' minore di quella gia' presente nello
      zBuffer, allora aggiorno i valori nelle due matrici con il nuovo colore e
          la nuova profondita'*/
   double zbuffer = zBuffer[x][y];
   if( (z<zbuffer && z>0) ){
      zBuffer[x][y] = z;
      //itemBuffer[x][y] = tr;
      image.setRGB(x, y, c);
   }
}


public void swap(int i, int j){
   // swap() e' una funzione che scambia i nomi (puntatori) di due oggetti di
       tipo Point
   Point t;
   t = new Point(p[i]);
   p[i] = new Point(p[j]);
   p[j] = new Point(t);
}


public void setIllumination(Point p, Point fragment){
   // getShade ottiene l'illuminazione del punto p nella scena, con normale N,
   // calcolando il contributo ambientale, diffusivo (Lambert) e riflessivo
       (Phong)

   // inizializzazione di variabili
   double intensity;
   double lambert=0, phong=0, phong_base, diffuse=0, specular=0, d;
   Light light=null;
   double amb = scn.amb; // amb e' il colore della luce ambientale
   Point /*shade ,*/ N = p.normal;
   Point L, V;
   Point o = camera; // centro di proiezione
   Ray shadow_ray;
   boolean visible;
```

```
for(int i=0; i<scn.lgt.size(); i++){
   // per ogni luce della scena, calcolo il suo contributo d'illuminazione
   visible = true;
   light = scn.lgt.get(i);

   // V e' il vettore che congiunge il centro del triangolo al centro di
       proiezione o
   V = p.to(o);

   // L e' il vettore che congiunge il centro del triangolo alla posizione
       della luce
   // (chiamato anche 'raggio d'ombra')
   L = light.getDirection(p);



   if(L.dot(N)<0){
      // Se la superficie del triangolo non 'vede' la luce,
      // non calcolo affatto il contributo di questa sorgente e
      // salto direttamente alla prossima luce
      continue;
   }

   shadow_ray = new Ray(light, p);
   for(int j=0; j<scn.obj.size() && visible; j++){
      if(p.ob != scn.obj.get(j))
         visible = shadow_ray.getVisibility(scn, j);
   }

   if(!visible) continue;
   /*
   RenderingEngine shadowCheck = new RenderingEngine(50,50,1, p, L,scn,
       generation+1);
   print(shadowCheck.render()[25][25].getRed());
   //itemBuffer[w_2][h_2].a.print();
   if(shadowCheck.itemBuffer[25][25] != null) { continue;}
   */
           // d e' la distanza del triangolo dalla luce
   intensity = light.i;
   if(light.type != 'd'){
      d = L.norm();
      intensity = light.i * light.getAttenuation(d); // intensity e'
          l'intensita' luminosa che raggiunge il punto
      L.normalize(); // normalizzo il raggio d'ombra
   }

   V.normalize(); // normalizzo il raggio di vista
   // L e V sono ora versori
```

```
        /* diff e' il contributo d'illuminazione, diffusivo calcolato con la
            formula
           di Lambert: diff = <L,N>. Notare che a questo punto della funzione diff
              non
           puo' essere negativo per il controllo effettuato prima */
        lambert = L.dot(N);
        diffuse += lambert*intensity; // incremento il contributo diffusivo

        /* ref e' il contributo di luce riflettente calcolato con la formula
           di Phong: ref = 2 <N,L> <N,V> - <L,V> */
        //phong = 2*N.dot(L)*N.dot(V) - L.dot(V);

        // H = L+V/||L+V|| blinn = <H, N>

        if(object.blinn){
           phong_base = (L.sum(V).normalized()).dot(N);
           phong_base*=phong_base; phong_base*=phong_base; //Blinn
        }
        else
           phong_base = 2 * N.dot(L) * N.dot(V) - L.dot(V);

        // se il contributo e' positivo
        if(phong_base>0){
           phong = phong_base;
           for(int j=1; j<object.reflection_exponent; j++)
              phong *= phong_base;

           specular += phong*intensity;
        }
     }

     diffuse += amb;
     fragment.diffuse = diffuse;
     fragment.specular = specular;
}

   public Point getShading(double u, double v, double diffuse, double specular){
     Point clr;
     if(object.texture != null && texture_mapping){
        clr = object.getColor(u, v);
     }
     else
        clr = color;
     Point shading = clr.per(diffuse*kDif);
     //specular = 0.1 * clr.z/(clr.x+clr.y+clr.z);
     //specular *= Point.clamp( 1.2*(2*clr.z-(clr.x+clr.y))
         ,0,1);//(clr.x+clr.y+clr.z);
     specular *= Point.clamp( 2*(clr.z-clr.x-clr.y) ,0,1);//(clr.x+clr.y+clr.z);
     shading.add(specular);
```

```
      shading.clamp(0,1);
      return shading;
   }

   public int round(double x){
      // round() arrotonda valori double all'intero piu' vicino
      // e' utilizzata per passare dalle corrdinate dello spazio (double)
      // alle coordinate della image (int)
      return (int) (Math.round(x));
   }

   public static void print(double x){
      System.out.println(x);
   }

   /*public void toCameraSpace(Point p){
      Point pu = p.min(camera.normal.per(f));
      p.set(p.dot(ux), p.dot(uy), p.z);
   }*/
   public void toCameraSpace(Point p){
      //Point pu = p.min(camera.normal.per(f));
   // p.set(p.dot(ux), p.dot(uy), p.min(camera).dot(camera.normal) );
      p.set(p.dot(ux), p.dot(uy), p.z);
   }

   public void shiftCamera(Point v){
      camera.shift(v);
      u0.shift(v);
   }

   public void rotateCameraSTEP(double theta){
      if(theta!=0) camera.normal = new Point(1,0,0);
   }
   public void rotateCamera(double theta){
      Point N = camera.normal;
      double cos = Math.cos(theta), sin = Math.sin(theta);
      double xr = N.x*cos + N.z*sin;
      double zr = -N.x*sin + N.z*cos;
      camera.normal.set(xr, N.y, zr);

      camera.normal.normalize();

   /* u0 = camera.sum( camera.normal.per(f) );

      ux = uy.vect(camera.normal);
      ux.normalize();*/
      u0 = camera.sum(camera.normal.per(f));
      uy = new Point(0, camera.normal.z, -camera.normal.y);
      uy.normalize();
```

```
      ux = uy.vect(camera.normal);
      ux.normalize();
   }

}
```

Ray.java:

```
/*
La classe Ray traccia un raggio nella scena e lo interseca con gli oggetti,
    restituendo il punto di intersezione piu' vicino;
serve per calcolare il contributo del raggio d'ombra.
*/
 public class Ray{

  // direzione raggio
  Point rd;

  // origine del raggio
  Point ro;

  public Ray(Point origin, Point direction){
    //costruttore
      rd = direction.min(origin);

      ro = origin;
  }

  public boolean getVisibility(Scene scn, int i){
    double t0;
    t0 = this.intersect(scn.obj.get(i));
    if( (t0<0.001 || t0>0.9) ) return true;
    else return false;
  }

  public boolean getVisibility2(Sphere[] bodies, int i){
       double t0;
       t0 = this.intersect(bodies[i]);
       if( (t0<0.001 || t0>0.9) ) return true;
       else return false;
     }

  public double intersect(Sphere ob){
    Point o_c = ro.min(ob.center);
    double b = rd.dot(o_c);
    double a = rd.dot(rd);
    double c = o_c.dot(o_c) - ob.size*ob.size;
```

```
    double delta = b*b - a*c;
    if(delta<0)
      return -10;
    delta = Math.sqrt(delta);
    double t1 = (-b + delta) / a;
    double t2 = (-b - delta) / a;
    if(t1<0) t1 = t2;
    if(t2<0) t2 = t1;
    double t = (t1<t2)? t1 : t2;
    if(t<0) return -20;
    return t;
  }


  public double intersect(Triangle tr){

    Point N = tr.getNormal();
    double rdn = N.dot(this.rd);
    double d = N.dot(tr.a);
    double t = (d - this.ro.dot(N)) / rdn;
    if(t >= 0.99 || t<0.01 ){
     return 0;
    }
    return t;
  }
}
```

Point.java:

```
/*
La classe Point definisce e costruisce punti in 3D (che formano i vertici delle
    maglie della scena) ed i loro metodi;
in particolare,il costruttore di un Point gli associa anche un colore RGB.
*/
public class Point {

   // cooridnate del punto nello spazio
   public double x;
   public double y;
   public double z;

   // colore
   public double diffuse;
   public double specular;
   public double u, v;
   // versore normale al punto
```

```
public Point normal;

public int id;

public Sphere ob;

public Point() {
   // costruttore
   this.set(0, 0, 0);
}

public Point(double a) {
   // costruttore
   this.set(a, a, a);
}

public Point(double a, double b, double c) {
   // costruttore da coordinate
   this.set(a, b, c);
}

public Point(Point P) {
   // costruttore che copia un altro punto P
   this.set(P.x, P.y, P.z);
   normal = P.normal;
   diffuse = P.diffuse;
   specular = P.specular;
   u = P.u;
   v = P.v;
}

public Point copy(Point P) {
   // costruttore che copia un altro punto P
   Point cp = new Point(P.x, P.y, P.x);
   cp.normal = P.normal;
   cp.diffuse = P.diffuse;
   cp.specular = P.specular;
   cp.u = P.u;
   cp.v = P.v;

   return cp;
}

public void set(double a, double b, double c) {
   // set delle coordinate e colore a grigio 70% circa
   x = a;
   y = b;
   z = c;
}
```

```
public void set(Point P) {
   // set delle coordinate
   x = P.x;
   y = P.y;
   z = P.z;
}

public Point sum(Point P) {
   // somma vettoriale
   return new Point(x + P.x, y + P.y, z + P.z);
}

public Point sum(double a, double b, double c) {
   // somma vettoriale
   return new Point(x + a, y + b, z + c);
}

public void add(Point P) {
   x += P.x;
   y += P.y;
   z += P.z;
}

public Point addP(Point P) {
   Point t=new Point();

   t.x = x + P.x;
   t.y = y + P.y;
   t.z = z + P.z;

   return t;
}

@Override
public String toString() {
   return "Point [x=" + x + ", y=" + y + ", z=" + z + "]";
}

public void add(double s) {
   x += s;
   y += s;
   z += s;
}

public void shift(Point v) {
   // traslazione di vettore v
   x += v.x;
   y += v.y;
```

```
   z += v.z;
}

public void shift(double a, double b, double c) {
   // traslazione da coordinate
   x += a;
   y += b;
   z += c;
}

public Point min(Point P) {
   // differenza vettoriale
   return new Point(x - P.x, y - P.y, z - P.z);
}

public Point per(double l) {
   // moltiplicazione per uno scalare l
   return new Point(x * l, y * l, z * l);
}

public void scale(double l) {
   x *= l;
   y *= l;
   z *= l;
}

public Point star(Point P) {
   return new Point(x * P.x, y * P.y, z * P.z);
}

public double dot(Point P) {
   // prodotto scalare
   return x * P.x + y * P.y + z * P.z;
}

public double dot(double x_, double y_, double z_) {
   // prodotto scalare
   return x * x_ + y * y_ + z * z_;
}

public Point vector(Point p){
   Point vect = new Point(0.0);

   vect.x=(this.y*p.z)-(this.z*p.y);
   vect.y=(this.z*p.x)-(this.x*p.z);
   vect.z=(this.x*p.y)-(this.y*p.x);

   return vect;
}
```

```
public Point vect(Point P) {
   // prodotto vettoriale
   return new Point(this.y * P.z - this.z * P.y, this.z * P.x - this.x * P.z,
       this.x * P.y - this.y * P.x);
}

public Point to(Point P) {
   // resituisce il vettore applicato che va da "this" al punto P
   return P.min(this);
}

public double norm() {
   // norma euclidea
   return Math.sqrt(this.dot(this));
}

public double squareNorm() {
   // norma euclidea al quadrato
   return this.dot(this);
}

public Point normalized() {
   // noramlizzazione
   double n = this.norm();
   if (n == 0)
      return this;
   return this.per(1 / n);
}

public double normalize() {
   // noramlizzazione
   double n = this.norm();
   if (n == 0)
      return 0;
   x /= n;
   y /= n;
   z /= n;
   return n;
}

public Point normalizeL1() {
   // normalizzazione secondo la norma l1
   double n = Math.abs(x) + Math.abs(y) + Math.abs(z);
   if (n == 0)
      return this;
   return this.per(1 / n);
}
```

```
public Point getNormal() {
   // getNormal() restituisce la normale, se esiste
   if (normal == null) {
      System.out.println("Normal is null!:");
      return new Point(0, 0, 0);
   }
   return this.normal;
}

public void setNormal(Point v) {
   // imposta normale al punto
   this.normal = v;
   normal.normalize();
}

public void clamp(double min, double max) {
   // clamp() costringe i valori del vettore fra min e max
   x = clamp(x, min, max);
   y = clamp(y, min, max);
   z = clamp(z, min, max);
}

public static double clamp(double value, double min, double max) {
   // clamp() costringe i valori di un double fra min e max
   if (value < min)
      value = min;
   if (value > max)
      value = max;
   return value;
}

public void print() {
   // stampa a schermo delle coordinate del punto
   System.out.println("p(" + x + " " + y + " " + z + ")");
}

public Point pow(double p) {
   return new Point(Math.pow(x, p), Math.pow(y, p), Math.pow(x, p));
}

public double getDistFrom(Point P) {
   // distanza fra due punti
   Point V = this.to(P);
   return Math.sqrt(V.dot(V));
}

public double getSquareDistFrom(Point P) {
   // distanza fra due punti
   Point V = this.to(P);
```

```
    return V.dot(V);
  }

  public Point project(double f) {
    /*
     * project() calcola la proiezione centrale di un punto sul piano z=0 rispetto
     * al centro (0,0,-f) f rappresenta la distanza del viewplane dal punto di
     * vista, in termini fotografici e' la lunghezza focale
     */
    double k = f / (f + z);

    // risultato della matrice di proiezione, conservo la coordinata z dello
        spazio
    return new Point(k * x, k * y, z);
  }

  public static Point comb(double a, Point A, double b, Point B) {
    // comb calcola la combinazione lineare di due vettori
    // usando i coefficienti a e b.
    return (A.per(a)).sum(B.per(b));
  }

  public static Point getInterpolation(double a, Point A, double b, Point B) {
    // Point p = comb(a,A,b,B);
    Point p = new Point(a * A.x + b * B.x, a * A.y + b * B.y, a * A.z + b * B.z);
    p.u = a * A.u + b * B.u;
    p.v = a * A.v + b * B.v;
    p.diffuse = a * A.diffuse + b * B.diffuse;
    p.specular = a * A.specular + b * B.specular;
    return p;
  }

  public static Point comb(double a, Point A, double b, Point B, double c, Point
      C) {
    // comb calcola la combinazione lineare di tre vettori
    // usando i coefficienti a,b e c.
    Point p = new Point(a * A.x + b * B.x + c * C.x, a * A.y + b * B.y + c * C.y,
        a * A.z + b * B.z + c * C.z);
    return p;
  }

  public static Point average(Point P, Point Q) {
    // punto medio fra due punti
    return (P.sum(Q)).per(0.5);
  }

  public static Point average(Point P, Point Q, Point U) {
    // baricentro di tre punti
    return (P.sum(Q).sum(U)).per(1 / 3.0);
```

```
}

public int toRGB() {
   // normalizzo a valori compresi fra 0 e 255
   Point shade = this.per(255);

   // contengo l'illuminazione fra il valore massimo (255) e minimo (0)
   shade.clamp(0, 255);
   int r = round(shade.x);
   int g = round(shade.y);
   int b = round(shade.z);
   int clr = (r << 16) | (g << 8) | b;
   return clr; // restituisco l'illuminazione finale
}

public Point getConvexCoeff(Point v1, Point v2, Point v3) {
   Point l1 = v1.to(this);
   Point l2 = v2.to(this);
   Point l3 = v3.to(this);
   double A3 = (l1.vect(l2)).norm();
   double A2 = (l3.vect(l1)).norm();
   double A1 = (l3.vect(l2)).norm();
   double A = A1 + A2 + A3;

   return new Point(A1 / A, A2 / A, A3 / A);
}

public Point getConvexCoeffNew(Point v0, Point v1, Point v2) {
   Point w1 = v1.min(v0);
   Point w2 = v2.min(v0);
   Point q = this.min(v0);
   double alpha = q.dot(w1) / w1.dot(w1);
   double beta = q.dot(w2) / w2.dot(w2);
   return new Point(1 - alpha - beta, alpha, beta);
}

public Point getConvexCoeff(Triangle tr) {
   Point v1 = tr.a, v2 = tr.b, v3 = tr.c;
   return this.getConvexCoeff(v1, v2, v3);
}

public Point getNormalInterpolation(Point p1, Point p2, Point p3) {
   Point coeff = this.getConvexCoeff(p1, p2, p3);
   double alpha = coeff.x, beta = coeff.y, gamma = coeff.z;
   Point N = Point.comb(alpha, p1.normal, beta, p2.normal, gamma, p3.normal);
   return N;// .normalize();
}

public Point getNormalInterpolation(Triangle tr) {
```

```
      return this.getNormalInterpolation(tr.a, tr.b, tr.c);
   }

   public Point project(Point camera, double f) {
      Point pc = this.min(camera);
      double depth = pc.dot(camera.normal);
      pc.x *= f / depth;
      pc.y *= f / depth;
      pc.z *= -1;
      // pc.shift(camera);
      Point p = new Point(pc.x, pc.y, depth);
      p.diffuse = diffuse;
      p.specular = specular;
      p.u = u;
      p.v = v;
      return p;

   }

   public Point projectX(Point camera, double f) { // proiezione sul piano x=0
      Point pc = this.min(camera);
      double depth = pc.dot(camera.normal);
      double tmp;
      tmp = pc.x;
      pc.x = pc.z * f / depth; // pc.x *= f / depth;
      pc.y *= f / depth;
      pc.z = tmp * (-1);
      // pc.shift(camera);
      Point p = new Point(pc.x, pc.y, depth);
      p.diffuse = diffuse;
      p.specular = specular;
      p.u = u;
      p.v = v;
      p.id = id;
      return p;

   }

   public Point projectY(Point camera, double f) {
      Point pc = this.min(camera);
      double depth = pc.dot(camera.normal);
      double tmp;
      tmp = pc.y;
      pc.x *= f / depth;
      pc.y = pc.z * f / depth; // pc.y *= f / depth;
      pc.z = tmp * (-1);
      // pc.shift(camera);
      Point p = new Point(pc.x, pc.y, depth);
      p.diffuse = diffuse;
```

```
      p.specular = specular;
      p.u = u;
      p.v = v;
      p.id=id;
      return p;

   }

   public void rotateZ(double theta) {
      double yr, xr;
      double cos = Math.cos(theta), sin = Math.sin(theta);
      xr = x * cos + y * sin;
      yr = -x * sin + y * cos;
      x = xr;
      y = yr;
   }


   public void rotateY(double theta) {
      double zr, xr;
      double cos = Math.cos(theta), sin = Math.sin(theta);
      xr = x * cos + z * sin;
      zr = -x * sin + z * cos;
      x = xr;
      z = zr;
   }

   public static int round(double x) {
      // round() arrotonda valori double all'intero piu' vicino
      // e' utilizzata per passare dalle corrdinate dello spazio (double)
      // alle coordinate della pixelMatrix (int)
      return (int) (Math.round(x));
   }

}



    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  \bigskip
\begin{lstlisting}
```

Triangle.java:

```
/*
La classe Triangle definisce e costruisce i triangoli (che formano le maglie della
    scena) ed i loro metodi.
```

```
*/

 public class Triangle{
   public Point a, b, c;
   public int IDa, IDb, IDc;


   public Point center;



   public Triangle(Point v1, Point v2, Point v3){
      a=v1;
      b=v2;
      c=v3;

      if(a != null && b != null && c != null)
         center = Point.average(a, b, c);

      if(a.normal != null && b.normal != null && c.normal != null)
         center.normal = Point.average(a.normal, b.normal, c.normal);
      else
         this.makeNormal();
   }

   public Point getNormal(){
      if(center.normal != null)
         return center.normal;
      if(this.updateNormal() != null)
         return center.normal;

      return this.makeNormal();
   }

   public Point updateNormal(){
      if(a.normal != null && b.normal != null && c.normal != null){
         center.normal = Point.average(a.normal, b.normal, c.normal);
         return center.normal;
      }
      else return null;
   }

   public Point makeNormal(){
      Point v = a.min(b);
      Point w = c.min(b);
      center.normal = w.vect(v).normalized();
      return center.normal;
   }
```

```
   public void print(){
      System.out.println("tr:");
      a.print();
      b.print();
      c.print();
   }

   public Point getCenter(){
      Point cnt = Point.average(a,b,c);
      center = cnt;
      return center;
   }

   public Triangle sum(Point p){
      return new Triangle(a.sum(p), b.sum(p), c.sum(p));
   }

   public void shift(Point v){
       // traslazione di vettore v
       a.shift(v);
       b.shift(v);
       c.shift(v);
       center = this.getCenter();
   }



    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  \bigskip
\begin{lstlisting}
```

Sphere.java:

```
/*
La classe Sphere crea le sfere che formano la Terra ed i singoli pianeti e gli
    applica e ruota le tessiture.
*/
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import javax.imageio.ImageIO;

public class Sphere {

   static final double g = 23;
```

```
static final int numPlanets = 6;

public ArrayList<Point> pts;
public ArrayList<Triangle> trn;
public static boolean c = true;
public double size;
public Point center;
public int n; // numero di paralleli e meridiani
public double texture_vel;
public double mass;
public String nome;
public Point v;
public boolean still = false;
public boolean precessione;
public double alphaPrec;
public String texture_name;
public String texture_stripes;
public String texture_chess;
public Point color;
boolean bool = false;
public double kDif, kRef;
public int reflection_exponent = 2;
public boolean blinn = false;
public BufferedImage texture;
ClassLoader classLoader;
InputStream is;
public static BufferedImage clouds, night;

public static void createPlanets(Sphere[] pianeti, double[] aPrecStart, Sphere[]
    tmp) {

   //earth
   pianeti[0].makeSphere(90, new Point(0, 0, 1100), 30, "images/world.jpg",
       "images/world.jpg", "images/world.jpg", 0.1, 1.0, "Terra", 0.41, 2.4);
   pianeti[0].setColor(new Point(0, 0, 1));
   pianeti[0].bool=true;
   // earth.setVelocity(new Point(0.4,0.4,0.4));
   aPrecStart[0]=pianeti[0].alphaPrec;
   pianeti[0].texture_vel = -0.005;
   pianeti[0].still = true;
   pianeti[0].rotateObjectY(pianeti[0].alphaPrec, pianeti[0].center);
   tmp[0].makeSphere(90, new Point(0, 0, 1100), 30, 2.4);

   //moon
   pianeti[1].makeSphere(25, new Point(300, -150, 1050), 30,
       "images/moon_NASA0.jpg", "images/chess.jpg","images/bw_stripes.jpg", 0.2,
       0.8, "Luna", 0.3, 0.9);//
   pianeti[1].setColor(new Point(0.3, 0.6, 1));
   pianeti[1].setVelocity(new Point(0, -5, 0));
```

```
aPrecStart[1]=pianeti[1].alphaPrec;
pianeti[1].texture_vel = 0.09;
pianeti[1].rotateObjectY(pianeti[1].alphaPrec, pianeti[1].center);
tmp[1].makeSphere(25, new Point(300, -150, 1050), 30, 0.9);

//mars
pianeti[2].makeSphere(26, new Point(-400, -200, 1050), 30,
    "images/Mars3.jpg","images/chess_purple.jpg","images/pw_stripes.jpg",
    0.2, 0.8, "Marte", -0.4, 0.9);//
pianeti[2].setColor(new Point(0.2, 0.5, 0.1));
pianeti[2].setVelocity(new Point(3, 5, 0));
aPrecStart[2]=pianeti[2].alphaPrec;
pianeti[2].texture_vel = 0.03;
pianeti[2].rotateObjectY(pianeti[2].alphaPrec, pianeti[2].center);
tmp[2].makeSphere(26, new Point(-400, -200, 1050), 30, 0.9);

//venus
pianeti[3].makeSphere(26, new Point(250, -160, 1050), 30,
    "images/Venus2.jpg", "images/chess_red.jpg","images/rw_stripes.jpg", 0.2,
    0.8, "Venere", 0.1, 0.9);//
pianeti[3].setColor(new Point(0.6, 0.5, 0));
pianeti[3].setVelocity(new Point(10, -10, 0));
aPrecStart[3]=pianeti[3].alphaPrec;
pianeti[3].texture_vel = 0.001;
pianeti[3].rotateObjectY(pianeti[3].alphaPrec, pianeti[3].center);
tmp[3].makeSphere(26, new Point(250,-160,1050), 30, 0.9);

//mercury
pianeti[4].makeSphere(25.5, new Point(-300, -300, 1050), 30,
    "images/mercury.jpg", "images/chess_blue.jpg","images/blw_stripes.jpg",
    0.2, 0.8, "Mercurio", 0.3, 0.8);
pianeti[4].setColor(new Point(0.6, 0, 0.5));
pianeti[4].setVelocity(new Point(-10, -4, 0));
aPrecStart[4]=pianeti[4].alphaPrec;
pianeti[4].texture_vel = 0.01;
pianeti[4].rotateObjectY(pianeti[4].alphaPrec, pianeti[4].center);
tmp[4].makeSphere(25.5, new Point(-300,-300,1050), 30, 0.8);

//jupiter
pianeti[5].makeSphere(25, new Point(250, -200, 1050), 30,
    "images/Jupiter.jpg", "images/chess_green.jpg","images/gw_stripes.jpg",
    0.2, 0.8, "Giove", -0.2, 0.9);
pianeti[5].setColor(new Point(0.2, 0, 0.4));
pianeti[5].setVelocity(new Point(-5, -7, 0));
aPrecStart[5]=pianeti[5].alphaPrec;
pianeti[5].texture_vel = 0.005;
pianeti[5].rotateObjectY(pianeti[5].alphaPrec, pianeti[5].center);
tmp[5].makeSphere(25, new Point(250,-200,1050), 30, 0.9);
```

```
}

public Sphere() {
   pts = new ArrayList<Point>();
   trn = new ArrayList<Triangle>();
   texture = null;
   v = new Point(0);

   if (clouds == null)
      try {
          classLoader = this.getClass().getClassLoader();
          is = classLoader.getResourceAsStream("images/clouds_new_rs.jpg");
         clouds = ImageIO.read(is);
      } catch (IOException e) {
      }
   if (night == null)
      try {
         classLoader = this.getClass().getClassLoader();
          is = classLoader.getResourceAsStream("images/world_night.jpg");
         night = ImageIO.read(is);
      } catch (IOException e) {
      }


}

public Sphere(ArrayList<Point> points, ArrayList<Triangle> triangles, double
    k_dif, double k_ref) {
   pts = points;
   trn = triangles;
   kDif = k_dif;
   kRef = k_ref;
   texture = null;
}

public void setTexture(String bitmap_name) {
   if (bitmap_name == null) {
      texture = null;
   }

   try {
      classLoader = this.getClass().getClassLoader();
      is = classLoader.getResourceAsStream(bitmap_name);
      texture = ImageIO.read(is);
   } catch (IOException e) {
   }
   if (texture != null)
      return;
```

```
   try {
      classLoader = this.getClass().getClassLoader();
      is = classLoader.getResourceAsStream("/images/texture_not_found.png");
      texture = ImageIO.read(is);
   } catch (IOException e) {
   }
}

double length(double a, double b, double c, double d) {
   double x = a - c, y = b - d;
   return Math.sqrt(x * x + y * y);
}

@Override
public String toString() {
   return "Obj [size=" + size + ", center=" + center + ", n=" + n + ", mass=" +
       mass + ", nome=" + nome + ", v="
         + v + "]";
}

public Point getColor(double u, double v){
   u = (u % 1.0);
   v = (v % 1.0);
   if(u<0) u+=1.0;

   double d = 0;
   int xpixel = (int) ( u*texture.getWidth() );
   int ypixel = (int) ( v*texture.getHeight() );

   Color RGB = new Color(texture.getRGB(xpixel, ypixel));
   Point clr = new Point(RGB.getRed(), RGB.getGreen(), RGB.getBlue());
   clr = clr.per(1.0/255);

   // SHADER

   if( bool ){

      xpixel = (int) ( u*night.getWidth() );
      ypixel = (int) ( v*night.getHeight() );
      RGB = new Color(night.getRGB(xpixel, ypixel));
      Point clrn = new Point(RGB.getRed(), RGB.getGreen(), RGB.getBlue());
      clrn = clrn.per(1.0/255);
      double x = (u+Canvas.t*(0.01+0.000947)) + 0.5;
      x = x%1.0;
      if(x<0) x+=1.0;
      x-=0.5;
      double a = 2*Math.abs(x);
      double b = Math.cos(3*x);
      b *= b; b*=b; b*=b; b*=b;b*=10;
```

```
        clr = Point.comb(a,clr,b,clrn);



        if(c){

           double uc=u, vc=v, verse=1.0/50;
           Point p = new Point(u-0.5,-v+0.5,0);
           p.add( FlowField(p).per( 0.03 ));
           p.add( FlowField(p).per( 0.03 ));

           uc = p.x-0.3;
           vc = p.y+0.5;
           uc = (uc % 1.0);
           vc = (vc % 1.0);
           if(uc<0) uc+=1.0;
           if(vc<0) vc+=1.0;

           Color cloud = new Color(clouds.getRGB( (int) ( uc*clouds.getWidth() ),
               (int) ( vc*clouds.getHeight() )));
           d = cloud.getRed()/100.0 - 0.2;
           d*=d;
           d = d>1.8? 1.8:d;
           clr.add(d);
        }
     }

     return clr;
  }


  public void setColor(Point c) {
     color = c;
  }

  public void setkDif(double x) {
     kDif = x;
  }

  public void setBlinn(boolean b) {
     blinn = b;
  }


  public void copy(Sphere cp) {

     this.center =new Point();
     this.color=new Point();
```

```
   this.size = cp.size; // r in px
   this.texture_name=cp.texture_name;
   this.texture_chess=cp.texture_chess;
   this.texture_stripes=cp.texture_stripes;
   this.center.set(cp.center);
   this.n = cp.n; // numero di paralleli e meridiani
   this.updatePosition();
   this.mass = cp.mass;
   this.v.set(cp.v);

   this.still = cp.still;
   this.nome = cp.nome;
   this.precessione = false;

   this.color.set(cp.color);
   this.kDif = cp.kDif;
   this.kRef = cp.kRef;
   this.reflection_exponent = cp.reflection_exponent;
   this.blinn = cp.blinn;

   this.setTexture(texture_name);
   this.alphaPrec=cp.alphaPrec;
   this.updatePosition();

}



public void setkRef(double x) {
   kRef = x;
}

public void makeSphere(double r, Point centro, int n, double R, double G, double
    B, double ref, double dif, double ro) {
   this.makeSphere(r, centro, n, ro);
   color = new Point(R, G, B);
   kDif = dif;
   kRef = ref;
   texture = null;
}

public void makeSphere(double r, Point centro, int n, String bitmap, String
    bitmapChess, String bitmapStripes,double ref, double dif, String name,double
    alpha, double ro) {

   this.makeSphere(r, centro, n, 1, 1, 1, ref, dif,ro);
   texture_name=bitmap;
   texture_chess=bitmapChess;
   texture_stripes=bitmapStripes;
```

```
      this.setTexture(bitmap);
      this.nome = name;
      this.alphaPrec=alpha;
   }

// public void makeSphere(double r, Point centro, int n, String bitmap, String
    bitmapChess, double ref, double dif, String name,double alpha, double ro) {
//
//    this.makeSphere(r, centro, n, 1, 1, 1, ref, dif,ro);
//    texture_name=bitmap;
//    texture_chess=bitmapChess;
//    this.setTexture(bitmap);
//    this.nome = name;
//    this.alphaPrec=alpha;
// }

   public void makeSphere(double r, Point centro, int n, double ro) {
      /*
       * makeSphere() genra una mesh sferica di triangonli la sfera ha raggio r,
       * centro c ed e' divisa in n paralleli ed n meridiani. In tutto e' composta
          da
       * n^2 triangoli. La sfera ha colore (R,G,B) e coefficiente riflessivo e
       * diffusivo rispettivamente pari a ref e dif.
       */

      mass = r*r*r*ro;
      center = centro;
      size = r;
      this.n = n;
      precessione = false;
      this.updatePts();
      this.updateTrn();
   }

   public void updatePts() {

      pts.clear();

      double r = size;

      double x, y, z;
      double pi = Math.PI;

      Point p;
      int id = 0;
      double theta = 0, phi = 0, dphi = pi / n, dtheta = dphi * 2;
      double cosphi, costheta, sinphi, sintheta;

      for (int i = 0; i < n; i++) {
```

```
      sinphi = Math.sin(phi);
      cosphi = Math.cos(phi);
      theta = 0;
      for (int j = 0; j < n; j++) {
         sintheta = Math.sin(theta);
         costheta = Math.cos(theta);
         x = r * sinphi * costheta + center.x;
         z = r * sinphi * sintheta + center.z;
         y = -r * cosphi + center.y;
         p = new Point(x, y, z);
         p.setNormal(center.to(p));
         p.u = theta / (2 * pi);
         p.v = phi / pi;
         p.diffuse = -1.0;
         p.id = id;
         p.ob = this;

         pts.add(p);

         id++;
         theta += dtheta;
      }
      phi += dphi;
   }

}

public void updateTrn() {

   Triangle t;

   trn.clear();

   int a, b, c;
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n; j++) {
         a = (i * n + j) % (n * n);
         b = ((i + 1) * n + j) % (n * n);
         c = (i * n + j + 1) % (n * n);
         t = new Triangle(pts.get(a), pts.get(b), pts.get(c));
         trn.add(t);

         a = ((i + 1) * n + j) % (n * n);
         b = ((i + 1) * n + j + 1) % (n * n);
         c = (i * n + j + 1) % (n * n);
         t = new Triangle(pts.get(a), pts.get(b), pts.get(c));
         trn.add(t);
      }
   }
```

```
   }

   public void updatePosition() {

      this.updatePts();
      this.updateTrn();

   }

public void rotateObj(Point L1, Point center, Sphere tmp) {

   Point p_tmp, p_new, _L;
   double _x,_y,_z,_nx,_ny,_nz,zr, yr, xr, nxr, nyr, nzr;

   _L=L1.normalized();
   double costheta=_L.z, sintheta=Math.sqrt(1-costheta*costheta), cosphi=_L.x,
       sinphi=_L.y;

   tmp.center=this.center;
   tmp.updatePts();

   for (int i = 0; i < pts.size(); i++) {
      p_tmp = tmp.pts.get(i);
      p_tmp.shift(center.per(-1));
      _x = p_tmp.x * costheta + p_tmp.z * sintheta;
      _y = p_tmp.y;
      _z = -p_tmp.x * sintheta + p_tmp.z * costheta;
      _nx = p_tmp.normal.x * costheta + p_tmp.normal.z * sintheta;
      _ny = p_tmp.normal.y;
      _nz = -p_tmp.normal.x * sintheta + p_tmp.normal.z * costheta;


      xr = _x * cosphi + _y * sinphi;
      yr = -_x * sinphi + _y * cosphi;
      zr = _z;
      nxr =_nx * cosphi + _ny * sinphi;
      nyr = -_nx * sinphi + _ny * cosphi;
      nzr = _nz;

      p_new = this.pts.get(i);
      p_new.shift(center.per(-1));
      p_new.set(xr, yr, zr);
      p_new.normal.set(nxr, nyr, nzr);
      p_new.shift(center);

      p_tmp.shift(center);
   }
```

```
}

public void rotateObjectZ(double theta, Point center) {
   Point p;
   double zr, xr, nxr, nzr;
   double cos = Math.cos(theta), sin = Math.sin(theta);
   for (int i = 0; i < pts.size(); i++) {
      p = pts.get(i);
      p.shift(center.per(-1));
      xr = p.x * cos + p.z * sin;
      zr = -p.x * sin + p.z * cos;
      nxr = p.normal.x * cos + p.normal.z * sin;
      nzr = -p.normal.x * sin + p.normal.z * cos;
      p.set(xr, p.y, zr);
      p.normal.set(nxr, p.normal.y, nzr);
      p.shift(center);
   }
}

public void rotateObjectY(double theta, Point center) {
   Point p;
   double yr, xr, nxr, nyr;
   double cos = Math.cos(theta), sin = Math.sin(theta);
   for (int i = 0; i < pts.size(); i++) {
      p = pts.get(i);
      p.shift(center.per(-1));
      xr = p.x * cos + p.y * sin;
      yr = -p.x * sin + p.y * cos;
      nxr = p.normal.x * cos + p.normal.y * sin;
      nyr = -p.normal.x * sin + p.normal.y * cos;
      p.set(xr, yr, p.z);
      p.normal.set(nxr, nyr, p.normal.z);
      p.shift(center);
   }
}


public void rotateTexture(double theta) {
   Point p;

   for (int i = 0; i < pts.size(); i++) {
      p = pts.get(i);
      p.u += theta;

   }

}

public void precessione(double theta) {
```

```
      this.rotateObjectZ(theta, center);
      this.rotateTexture(theta);

   }

   void shift(Point v) {
      // shift() opera una traslazione di vettore v a tutti i punti
      // della scena compresi fra gli indici della lista dei punti della mesh
      for (int i = 0; i < pts.size(); i++) {
         pts.get(i).shift(v); // traslazione del punto
      }
      center.shift(v);
   }

   void scale(double s) {
      Point p;
      for (int i = 0; i < pts.size(); i++) {
         p = pts.get(i);
         p.shift(center.per(-1));
         p.scale(s);
         p.shift(center);
      }
      size *= s;
      mass *= s * s * s;
   }

   public static double smoothstep(double edge0, double edge1, double x) {
      double t;
      t = Point.clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
      return t * t * (3.0 - 2.0 * t);
   }

   Point VortF(Point q, Point c) {
      Point d = q.min(c);
      Point result = new Point(d.y, -d.x, 0);
      result = result.per(0.25 / d.dot(d) + 0.05);
      return result;
   }

   Point FlowField(Point q) {
      Point vr, c;
      double dir = 1.;
      c = new Point((MyFrame.canvas.t % 75 - 100) / 60.0, 0.6 * dir, 0);
      vr = new Point(0, 0, 0);
      for (int k = 0; k < 5; k++) {
         vr.add(VortF(q.per(4), c).per(dir));
         c = new Point(c.x + 1., -c.y, 0);
         dir = -dir;
```

```
   }
   return vr;
}

public void setVelocity(Point p) {
   v = p;
}

public void simulate() {

if (!still)
   this.shift(v);

}

public void setStill(boolean b) {
   still = b;
}

public void setMass(double mass) {
   this.mass = mass;

}
```

[1] A. Appel, *Some Techniques for Shading Machine Renderings of Solids*, S.J.C.C. (1968), 37–45.

[2] P. R. Atherton, *A Method of Interactive Visualization of CAD Surface Models on a Color Video Display*, Proceedings of SIGGRAPH 81, vol. 15, 1981.

[3] Ph. Dutré, K. Bala, and Ph. Bekaert, *Advanced Global Illumination,* 2^{nd} *Ed.*, A.K.Peters, 2006.

[4] J. F. Blinn, *Models of Light Reflection for Computer Synthesized Pictures*, Computer Graphics **11** (1977), 192–198.

[5] ———, *Simulation of Wrinkled Surfaces*, Computer Graphics **12** (1978), 286–292.

[6] J. F. Blinn and M. E. Newell, *Texture and Reflection in Computer Generated Images*, CACM **19** (1976), 542–547.

[7] E. E. Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Ph.D. Thesis, University of Utah, 1974.

[8] T. E. Hull and A. R. Dobell, *Random Number Generators*, SIAM Review **2** (1962), 230–254.

[9] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg, *A Progressive Refinement Approach to Fast Radiosity Image Generation*, Computer Graphics **23** (1989).

[10] M. F. Cohen and D. P. Greenberg, *The Hemi-Cube: A Radiosity Solution for Complex Environments*, Computer Graphics **19** (1985), 21–40.

[11] R. Cook, T. Porter, and L. Carpenter, *Distributed Ray Tracing*, Proceedings of SIGGRAPH 84, vol. 18, 1984.

[12] R. Cook and K. E. Torrance, *A Reflectance Model for Computer Graphics*, ACM Trans. on Graphics **1** (1982), 7–24.

[13] M. Dippé and J. Swensen, *An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis*, Proceedings of SIGGRAPH 84, vol. 18, 1984.

[14] F. Forti, *Rendering fotorealistico tridimensionale*, Tesi di Laurea, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", 2013. relatore: Prof. Massimo Picardello; correlatore: Dr. Fabrizio Bazzurri.

[15] G. Giordano (2020). Comunicazione personale (progetto didattico al corso di Computer Graphics, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", docente M. Picardello).

[16] A. S. Glassner, *Space Subdivion for Fast Ray Tracing*, IEEE Computer Graphics and Applications **4** (1994), 15–22.

[17] R. A. Goldstein and R. Nagel, Datamotion **69** (1968).

[18] ———, *3-D Visual Simulation*, Simulation **16** (1971), 25–31.

[19] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Bataille, *Modeling the Interaction of Light between Diffuse Surfaces*, Proceedings of SIGGRAPH 84, vol. 18, 1984.

[20] H. Gouraud, *Continuous Shading of Curved Surfaces*, IEEE Transactions on Computers **20** (1971), 623–628.

[21] R. A. Hall, *Hybrid Techniques for Rapid Image Synthesis*, Proceedings of SIGGRAPH 86, vol. 20, 1986. Tutorial Course Notes in "Image Rendering Tricks", T. Whitted & R. Cook, eds,.

[22] T. E. Hull and A. R. Dobell, *Random Number Generators*, SIAM Review **2** (1962), 230–254.

[23] D. S. Immel, M. F. Cohen, D. P. Greenberg, and B. Bataille, *A Radiosity Method for Non-Diffuse Environments*, Proceedings of SIGGRAPH 86, vol. 20, 1986.

[24] H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping, second edition*, AK Peters, Natick, Massachusetts, 2005.

[25] J. T. Kajiya, *The Rendering Equation*, Proceedings of SIGGRAPH 86, vol. 20, 1986.

[26] D. S. Kay, *Transparency, Reflection and Ray Tracing for Computer Synthesized Images*, Proceedings of SIGGRAPH 86, vol. 20, 1986.

[27] ———, *Transparency, Reflection and Ray Tracing for Computer Synthesized Images*, M.S. Thesis, Program of Computer Graphics, Cornell University, Ithaca, NY, 1979.

[28] D. S. Kay and D. Greenberg, *Transparency for Computer Synthesized Images*, Proceedings of SIGGRAPH 79, vol. 13, 1979.

[29] D. S. Kay and J. T. Kajiya, *Ray Tracing Complex Scenes*, Proceedings of SIGGRAPH 86, vol. 20, 1986.

[30] *Rendering fotorealistico in Java* (2019). relatore: Prof. Massimo Picardello.

[31] A. Mammen, *Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel mMps Technique*, Computer Graphics and Applications **9** (1989), 43–55.

[32] G. Nazzaro (2015). Comunicazione personale (progetto didattico al corso di Computer Graphics, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", docente M. Picardello).
[33] M. Pharr and G. Humphreys, *Physically Based Rendering, second edition*, Morgan Kaufmann, Burlington, Massachusetts, 2010.
[34] Phong Bui-Tong, *Illumination for Computer Generated Pictures*, Comm. A.C.M. **18** (1975), 311–317.
[35] M. Petreri (2016). Comunicazione personale (progetto didattico al corso di Computer Graphics, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", docente M. Picardello).
[36] ——— (2016). Comunicazione personale (progetto didattico al corso di Computer Graphics, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", docente M. Picardello).
[37] ——— (2017). Comunicazione personale (progetto didattico al corso di Computer Graphics, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", docente M. Picardello).
[38] ——— (2015). Comunicazione personale (progetto didattico al corso di Programmazione ad Oggetti e Grafica, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", docente F. Bazzurri).
[39] M. A. Picardello and L. Zsido, *Appunti di Algebra Lineare*, Università di Roma "Tor Vergata", 2006. www.mat.uniroma2.it/∼picard/SMC/didattica/materiali_did/Alg.Lin./ALG_LIN.pdf.
[40] J. R. Rossignac and A. A. G. Requicha, *Constructive non-regularized geometry*, Computer-Aided Design **23**, 21–32.
[41] H. Samet, *Neighbor finding techniques for images represented by quadtrees*, Computer Graphics and Image Processing **18** (1982), 37–57.
[42] R. Siegel and J. Howell, *Thermal Radiation Heat Transfer, 2^{nd} edition*, Hemisphere, Washington, DC, 1981.
[43] F. Sillion and C. Puech, *A General Two-Pass Method Integrating Specular and Diffuse Reflection*, Proceedings of SIGGRAPH 89, vol. 23, 1989.
[44] J. Stoer and R. Bulirsch, *Introduzione all'analisi numerica*, Vol. 2, Zanichelli, Bologna, 1979.
[45] K. E. Torrance, E. M. Sparrow, and R. C. Birkebak, *Polarization, Directional Distribution and Off-Specular Peak Phenomena in Light Reflected from Roughened Surfaces*, J.Opt.Soc.Am. **56** (1966), 916–925.
[46] K. E. Torrance and E. M. Sparrow, *Theory for Off-Specular Reflection from Roughened Surfaces*, J.Opt.Soc.Am. **57** (1967), 1105–1114.
[47] D. Verde (2020). Comunicazione personale (progetto didattico al corso di Metodi Numerici in Computer Graphis, Corso di Laurea in "Scienze e Tecnologie per i Media", Università di Roma "Tor Vergata", docente M. Picardello).
[48] E. Veach and L. J. Guibas, *Bidirectional Estimators for Light Transport*, 1994, pp. 147–162.
[49] J. R. Wallace, M. F. Cohen, and D. P. Greenberg, *A Two-pass Solution to the Rendering Equation: a Synthesis of Ray Tracing and Radiosity Methods*, Computer Graphics **21** (1987), 311–320.
[50] J. R. Wallace, K. A. Elmquist, and E. A. Haines, *A Ray Tracing Algorithm for Progressive Radiosity*, Computer Graphics **23** (1989), 315–324.
[51] D. R. Warn, *Lighting Controls for Synthetic Images*, Computer Graphics **17** (1983), 13–21.
[52] T. Whitted, *An Improved Illumination Model for Shaded Display*, Communications A.C.M. **23** (1980), 343–349.
[53] L. Williams, *Casting Curved Shadows on Curved Surface*, Computer Graphics **12** (1978), 270–274.