Project NMCGJ 2024-2025: Sokoban Game

The aim of the project is to design and implement a variation of the video game Sokoban. Sokoban is a classic puzzle game, and fits in the category of legendary games like Pacman and Tetris.

Gameplay

The purpose of the game is to push crates (or boxes) around in a warehouse, trying to get them to designated storage locations.

The player can navigate a “pusher”, either a man or a push-truck, through the warehouse. The pusher can move in four directions (left, right, up, and down) but not diagonal. The player should steer the pusher so as to push each of the crates to one of the storage locations, with a minimal number of moves.

In the beginning of the game, all the crates are spread over the warehouse. The game ends when these crates are all placed at the designated storage locations. The score is computed by earning 50 points per correctly stored crate minus the number of moves the pusher needed to complete the task.

The figures on the right show three snapshots at different moments during the game. Figure 1 illustrates a possible initial game setup: the pusher (yellow truck) is parked somewhere in a given warehouse (its floor plan is indicated with gray walls); in this case there are 16 crates to be stored (brown boxes) and the same number of storage locations (pink disks). Figure 2 depicts the game situation after the pusher has already stored 5 crates and is completing another one; crates that are stored at a correct location get a different color. In Figure 3 we see the final game situation where all crates have been stored correctly. Here, the pusher needed 669 moves to complete the task...

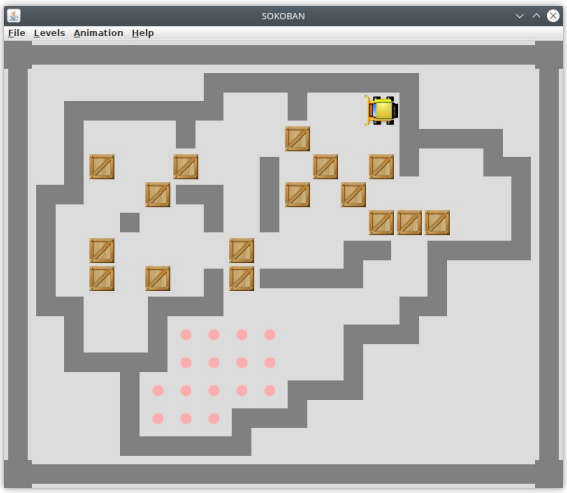

Figure 1: Snapshot of the initial game setup

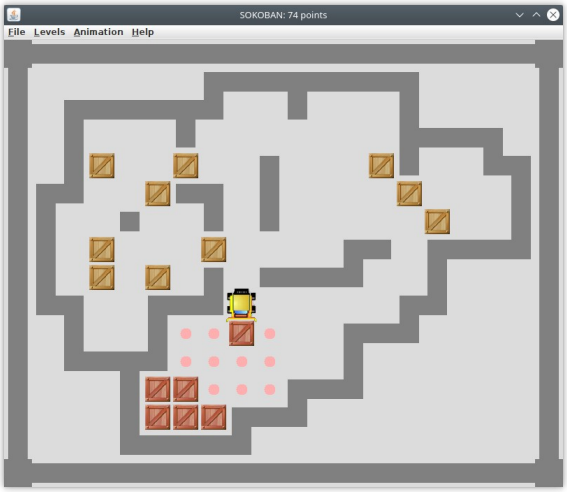

Figure 2: Snapshot of the game after a while

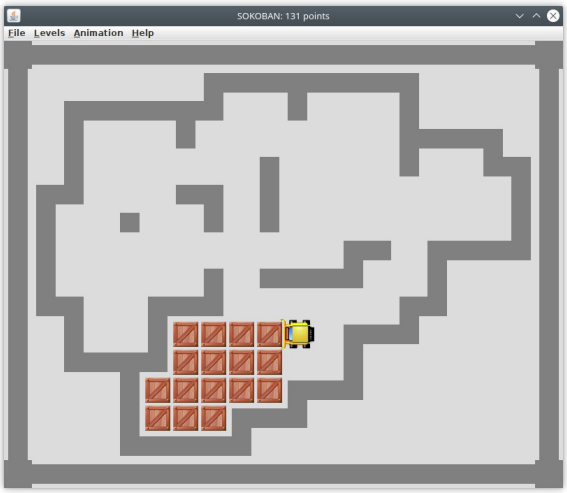

Figure 3: Snapshot of the game when all the crates have been stored at a correct location

Problem

The puzzle game Sokoban has to be implemented for this project. The game is played on a board of virtual squares. The board contains the floor plan of a warehouse. This must be completely surrounded by a wall (like in the example), and any number of walls can be placed in the interior to build up the warehouse. The initial position of the crates and the storage locations must be chosen so that there exists a feasible way for the pusher to get each of the crates to one of the storage locations.

The implementation needs to be flexible: it should be easy to specify/modify the floor plan of the warehouse, the number and position of the crates, etc. Note that there are several "board objects" (namely the pusher, crates, storage positions, and the stones/walls identifying the warehouse) which share certain common features (like a board position). This suggests that such features could be collected in one or more super-classes, and that concrete classes for these objects could be constructed by means of inheritance.

The pusher is confined to the board and may move horizontally or vertically onto empty squares (never through walls or crates). In each step of the game, the pusher should check whether the player has pressed a key to make a move. Each direction is linked with a special key (for example, one of the arrow keys in the keyboard). The pusher can move a crate by bumping into it and pushing it to the square beyond. Crates cannot be pulled, and they cannot be pushed to squares with walls or other crates. The puzzle is solved when all crates are placed at storage locations. Note that any crate can be placed at any storage location to solve the puzzle.

It could happen that one or more crates cannot be moved any further. An example of such a "deadlock" situation is shown in Figure 4. In such case the puzzle cannot be solved anymore and so it is game over (without completion of the puzzle).

Figure 4: Example of a deadlock situation

In order to be able to analyze the game situation (and the performance of solving the puzzle), it is required to keep track of all the game steps done so far. These steps may be combined into an animated video that may be watched at any moment. It should also be possible to interrupt the video and continue the game from there on (that is the game situation in the video at the moment of interruption).

In order to simplify the task, the following classes have been prepared.

BoardPanel	A panel for working with a graphical gameboard.
BoardFrame AnimationBoardFrame KeyAnimationBoardFrame	A frame containing a BoardPanel ... with animation ... with keyboard interaction.

The class BoardPanel

The class `BoardPanel` provides a panel for simple drawings arranged into a board consisting of a certain number of rows and columns. This class is very similar to the class `GraphicsPanel`, but is more efficient in drawing and removing specific objects at certain board positions.

BoardPanel(int rows, int cols)	Constructs a board panel.
getRows()	Returns the number of rows of the board.
getColumns()	Returns the number of columns of the board.
isInside(int i, int j): boolean	Checks whether the position (i, j) is inside or not.
clear()	Clears the complete board.
clear(int i, int j)	Clears the component at board position (i, j).
drawRectangle(int i, int j, Color color, int density)	Draws a filled rectangle at board position (i, j) with a given color and density/size.
drawOval(int i, int j, Color color, int density)	Draws a filled oval at board position (i, j) with a given color and density/size.
drawLine(int i, int j, Color color, int dir, int density)	Draws an axis-aligned line at board position (i, j) with a given color, direction, and density/size.
drawCross(int i, int j, Color color, int[] dirs, int density)	Draws an axis-aligned cross at board position (i, j) with a given color, array of directions, and density/size.
drawImage(int i, int j, Image image, int density)	Draws a scaled version of a given image at board position (i, j) with a given density/size.
repaint()	Repaints the board panel

The class BoardFrame

The class `BoardFrame` provides a frame with a board panel. It inherits from the class `JFrame` which is part of the package `javax.swing`. The following methods are available in the class `BoardFrame`, which is very similar to the class `GraphicsFrame`:

BoardFrame(String title, int rows, int cols, int size)	Constructs a board frame with a given title, and the dimensions of the board.
start()	Visualizes the frame.
close()	Closes the frame.
setTitle(String title)	Sets the title of the frame.

setMenuVisible(boolean visible)	Indicates whether the menu bar is visible or not.
setGraphicsDimension(int width, int height)	Sets the preferred dimension of the board panel.
getBoardPanel(): BoardPanel	Gives the board panel of the frame.

Some simple dialog frames are also available:

showMessageDialog(String msg, String title)	Shows a dialog with a given message.
showInputDialog(String msg, String init): String	Shows a dialog requesting a string.
showInputDialogInt(String msg, int init): int	Shows a dialog requesting an integer.
showInputDialogDouble(String msg, double init): double	Shows a dialog requesting a floating point.

Some useful methods for reading and writing images

readImage(File file): BufferedImage	Reads an image from a file.
writeImage(File file, BufferedImage image)	Writes an image to a file.

To change the actions of the menu bar, the following methods should be overridden:

clearBoard()	Called by the menu File > New.
loadGraphicsFile(File file)	Called by the menu File > Open.
saveGraphicsFile(File file)	Called by the menu File > Save.

The class AnimationBoardFrame

The class `AnimationBoardFrame` is prepared for making animated 2D graphics. It is a subclass of the class `BoardFrame`, and starts a new thread for each animation. The following specific methods are available to run an animation:

playAnimation()	Plays the animation when not currently active or resumes the animation when paused. It has no effect if the animation is already running.
pauseAnimation()	Pauses the animation.
stopAnimation()	Terminates the animation.
isAnimationEnabled(): boolean	Indicates whether the animation is enabled or not.

isAnimationPaused(): boolean	Indicates whether the animation is paused or not.
setAnimationDelay(long millis)	Sets the delay time (in milliseconds) between each animation step.

The class `AnimationBoardFrame` is an abstract class, and provides an animation environment. It requires the implementation of the following abstract methods (similar to the class `AnimationGraphicsFrame`):

animateInit()	Initializes a new animation (is called before the start of the animation).
animateNext()	Executes the next step in the animation.
animateFinal()	Finalizes the animation (is called after the end of the animation).

These methods have to be implemented by a subclass and define the specific animation.

The class KeyAnimationBoardFrame

The class `KeyAnimationBoardFrame` is prepared for interacting with the keyboard. It is a subclass of the class `AnimationBoardFrame`. The class is listening to key events, and whenever a key is pressed the following abstract method is called:

processKey(KeyEvent e)	Processes the given key event.

This method has to be implemented by a subclass and specifies the action that has to be undertaken when a key is pressed. Information about the specific pressed key is passed by means of an instance of the class `KeyEvent`.

The class `KeyEvent` is part of the package `java.awt.event`, and is designed for passing key information. Each key has a specific key code, and can be retrieved by the method

getKeyCode(): int	Returns the code associated with the key in this event.

The integer key codes are stored in static constants in the class `KeyEvent`. Some useful examples are given by

VK_<C>	Code of the key with the letter or number <C>. This can be A-Z or 0-9
VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT	The codes of the arrow keys.

More info can be found on the webpage: https://docs.oracle.com/en/java/javase/22/docs/api/java.desktop/java/awt/event/KeyEvent.html

Minimal requirements

- The game consists of a pusher (steered by the player) and a specific warehouse configuration with a certain number of crates to be stored (single game level).
- The implementation must allow to place the pusher, crates, storage locations and walls/stones on the board at arbitrary places. The number of crates should be flexible.
- The pusher, crates and walls may have simple graphical shapes (rectangles, ovals, lines, etc.). Use different shapes and/or colors to distinguish between the different board objects.
- The implementation must be able to combine all the game steps played so far into an animated video. It should be possible to interrupt the video and continue the game from there on.
- A good Java program design.

Suggestions for further options

- The option to load a user-defined warehouse configuration from a file. For this purpose, you can use the classes `CoordinateIO` and `Coordinate` (see the pre-project for their description).
- A multi-player option where there are several pushers. Each of them are steered with a different set of keys.
- The option to play with different puzzles (with different warehouse configurations, crate positions, etc.). The puzzles could be organized according to level of difficulty, and once a puzzle has been solved, the player(s) can move on to the next puzzle.
- Add new game elements to the basic puzzle. Examples include holes, teleports, moving blocks, one-way passages, destructible walls.
- Instead of composing the board objects (the pusher, the crates, the walls, ...) of simple graphical shapes (rectangles, ovals or lines) of different colors, they could also be composed of more fancy images. This could be done with the aid of the pre-defined methods `readImage` and `drawImage` provided by the classes `BoardFrame` and `BoardPanel`, respectively. Their use is illustrated with the following example code:

```
BoardFrame frame = new BoardFrame("My Board Frame", 10, 10, 40);
BufferedImage image = frame.readImage(new File("MyImage.png"));
BoardPanel board = frame.getBoardPanel();
board.drawImage(1, 1, image);
frame.start();
```

- Surprise me...

Notes

- A good Java program is not just a program that produces “the right result”; it should also be designed properly. In a good program design, every class (and method) should be responsible for a single well-defined job.
- Do NOT modify any of the pre-defined classes `BoardPanel`, `BoardFrame`, `AnimationBoardFrame`, `KeyAnimationBoardFrame`, `CoordinateIO`, and `Coordinate`.

Practical Information

The deadline for the project is **January 12, 2025**. Your report of the project can be turned in electronically by sending an email to speleers@mat.uniroma2.it. Such a report should include:

1. the source code of your program;
2. a class diagram of your program;
3. an overview of your program design decisions.

Good luck and have fun!

Hendrik Speleers