PS4a: Sokoban UI

Sokoban (Japanese for warehouse keeper) is a tile-based video game in which the player controls a worker pushing boxes around a warehouse into their designated locations. It is one of the earliest block-pushing video games, and has inspired many clones, derivatives, and references.

Other games in the genre add other features, such as a hex-based grid in Hexoban, holes and teleporters in Block-o-mania, enemies that must be dodged in Chip's Challenge and Adventures of Lolo, all the way up to encoding the rules of the game in blocks that can be pushed and manipulated in Baba is You.

Block-pushing puzzles are a recurring element in the Legend of Zelda series, many RPGs, such as Golden Sun and Lufia II, make frequent use of them, and NetHack has a sidequest branch based on Sokoban.

In this assignment, you will load level data from a file and draw it to the screen. You will add the gameplay mechanics in part B.

1 Details

For Part A of this assignment, we will create a program that loads and displays a Sokoban level. In Part B, we will add the gameplay elements.

- Make sure to download the level specification files and images files from `sokoban.zip`
- You should build a command-line app which takes a file name as a parameter(e.g. `level1.txt`). Name your executable `Sokoban`, so you would run it with e.g.

```
./Sokoban level1.lvl
```

1.1 Reading in the level

The input format is a text file that contains the information for a particular level On the first line will be two integers containing the height h and width $w$of the level. This will be followed by h lines containing w symbols each representing one of the cells of the initial game board. Further lines are ignored by the game and can be used as comments to describe the level.

The symbols used are as follows:

@ The initial position of the player. Each level contains exactly one @.

. An empty space, which the player can move through.

A wall, which blocks movement.

A A box, which can be pushed by the player.

a A storage location, where the player is trying to push a box.

1 A box that is already in a storage location.

The player can move in any of the four cardinal directions, but not diagonally. They can move into either an unoccupied space or storage location, or into a box provided there is an unoccupied space behind it to push the box into. Note that the player can only push one box at a time and the can only push (not pull) boxes. If the player pushes a box onto each storage location, they win.

Figure 1: level1.lvl

As an example, `level1.lvl` contains the following level data.

```
10 10
##########
#....a...#
#....A...#
#........#
#...##...#
#...##...#
#..@..A..#
#.......a#
#........#
##########
```

This yields the following initial game state.

1.2 Implementation

For this project, you'll create a namespace `SB` that has a `Sokoban` class which will contain information about the level and a `Direction` enumeration.

The `Sokoban` class should have the following features:

- It must be `sf::Drawable`, with a `protected virtual void` method named `draw` (as required of `sf::Drawable` subclasses).

- `const` `width()` and `height()` methods that return the dimensions of the game board.
- A `const` `playerLoc()` method that returns the player's current position, with (0, 0) being the upper-left cell in the upper-left corner.
- A `movePlayer()` method that takes a `Direction` that changes the player's location. For this part, the behavior of `movePlayer()` is undefined and is allowed to do nothing. You will complete this for part (b).
- A `isWon()` method that returns whether the player has won the game. For this part, `isWon()` may return a fixed value. You will complete this for part (b).
- It may contain one or more `sf::Sprite` objects (as well as the `sf::Texture` objects needed to hold the `Sprite`s' images).
- A `>>` operator which reads the level from a stream. You may want to add a `<<` operator that writes the level back to a stream.

The `Direction` enumerations will be `Up`, `Down`, `Left`, and `Right` (in keeping with SFML's naming conventions).

1.3 Matrices

You will need to use a two-dimensional grid (or *matrix*) to store the game state. There are two ways to store the matrix; row-major order or column-major order, as can be seen in Figure 2. You could do this with a two-dimensional array (or `vector` of `vector`s), but you can also use a one-dimensional array. If you have the x and y coordinates in matrix-space, you can convert to an index i in array-space by the formula $i = x + y \cdot w$ if the matrix is stored in row-major order. You can convert back using $x = i\%w$ and $y = i/w$.

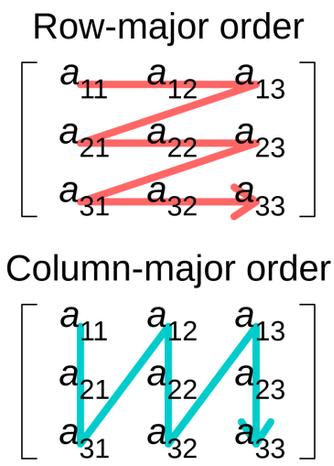

Figure 2: Row major vs column major order
Source: `https://en.wikipedia.org/wiki/Row-_and_column-major_order`

2 Resources

The `sokoban.zip` file contains the following image files:

The files were created by Kenney Vleugels and are licensed under the Creative Commons Zero license and can be found at `https://kenney.nl/assets/sokoban`. Each image is 64×64 pixels in size.

You may use alternate images instead of the provided ones. If you do so, you **must** source your images in the `readme`. **Using unsourced resources will result in a significant penalty.** The Kenney Sokoban pack is already cited in the provided `readme`.

Note that `sf::Texture` objects are heavyweight but `sf::Sprite` objects are lightweight. You should read a `Texture` just once for each image and share them between multiple `Sprite` objects to improve load speeds and reduce memory usage.


block_06 Wall


crate_03 Box


ground_01 Empty


ground_04 Storage


player_05


player_08


player_17


player_20

Figure 3: Tiles

3 Extra Credit

You can earn extra credit by drawing a move counter so the player can see how many moves they took to solve the puzzle. If you do any of the extra credit work, make sure to describe exactly what you did in `Readme-ps4a.md`.

4 Development Process

There are a lot of parts to this assignment. We'd suggest the following incremental development process:

1. Create a bare-bones version of your `Sokoban` class.
2. Overload the `>>` operator and read the map size from the file.
3. Create a stub for the `draw` method that draws walls for every tile location.
4. Setup the main draw loop in your main file.
   - Hint: Your main will need to know how big to make the window, which is based on the level's grid size.
   - Verify that the correct number of tiles are drawn.
5. Create a structure to represent the game grid. You will have to decide how to represent this.
6. Implement the `playerLoc()` method.
7. Implement the `<<` operator and write the map back to the terminal. Verify that it is as expected.
8. Read in the grid information from the file. For now, only worry about wall vs non-wall tiles.
9. Update the `draw` method to correctly draw your grid.
10. Read the player, box, and storage locations from the file. You might want to store this information separately from the tile data.

5 What to turn in

Your makefile should build a program named `Sokoban` and a static library `Sokoban.a`.

Submit a zip archive to Blackboard containing:

- Your main file `main.cpp`
- Your Sokoban (`Sokoban.cpp`, `Sokoban.hpp`) class
- The makefile for your project. The makefile should have targets `all`, `Sokoban`, `Sokoban.a`, `lint` and `clean`. Make sure that all dependencies are correct.

- Your `Readme-ps4a.md` that includes
  1. Your name
  2. Statement of functionality of your program (e.g. fully works, partial functionality, extra credit)
  3. Key features or algorithms used
  4. Any other notes
- Any other source files that you created.
- Any images or other resources not provided (such as for the extra credit)
- A screenshot of program output

Make sure that all of your files are in a directory named `ps4a` before archiving it and that there are no `.o` or other compiled files in it.

6 Grading rubric

Feature	Points	Comment
Autograder	10	Full & Correct Implementation
	4	Contains all code files and builds
	3	Reads files from >> and gets correct width and height
	2	Has `playerLoc()` function and gets correct position
	1	Has `movePlayer()` and `isWon()` methods (behavior undefined)
Drawing	23	
	2	`Sokoban` is `Drawable` with *protected* `draw()` method
	4	Draws the wall and floor
	4	Draws the player
	4	Draws the crates and storage
	3	Gets positions from level file
	3	Handles non-square levels correctly
	3	Window sized to play area
Readme and screenshot	5	Complete
	2	Describes the algorithms or data structures used.
	2	Describe how the level was drawn.
	2	Describes how the features were implemented.
	1	Screenshot representative of game behavior.
Extra Credit	2	
	+2	Shows turn counter
Penalties		
	-25	Uses unsourced images or other resources
	-5	Memory leaks or issues
	-5	Linting problems
	-3	Non-private fields
	-10%	Each day late
Total	40	