Project document - Labyrinth

Severi Koivumaa
100133819
Degree program: Bachelor’s and master’s in science and technology (Computer science)
Year of studies: 2022
Date: 18.4.2023

General description

The program creates a graphical labyrinth with overlapping and undercutting routes using a "weave-type" algorithm. The labyrinth size can be customized using adjustable parameters. A player-controlled character is placed inside the labyrinth, and the player's objective is to escape. The program generates one or more exit points based on the current round, and an option to allow the player to view the shortest possible path.

The game includes a polished gameplay experience with built-in tutorials to explain the game mechanics, and a graphical user interface that is intuitive to use. The players’ scores are tracked through a scoreboard, and the screen moves dynamically with the player, allowing for larger labyrinth sizes. Additionally, players can save and load generated labyrinths. Overall, I would argue that the project reaches the requirements for the demanding category.

User interface

You can launch the program by running the Main.scala file that can be found in the directory “src/main/scala/labyrinthGame”. When you open the project in IntelliJ, please open the root folder as a project to make the assets work properly. If you test on a laptop make sure that program scaling is turned off in display settings. Alternatively, if you have Java JDK installed on your computer, the program can be launched using the accompanying .jar file that is found inside the root directory by name “Fox Run.jar”. The program is controlled through a graphical user interface that is intuitive to control. Menu items can be accessed by clicking associated buttons.

In the menu you can click “How-to” to see controls and the goal for the game. Clicking “Scoreboard” displays current scores: players’ placement, points and usernames. Clicking “Options” allows you to change a few key parameters such as map size, starting round length and set the game to fullscreen. Change map size and round length by writing a positive integer to the associated text box and press “enter” -key. Fullscreen can be enabled and disabled by clicking its associated checkbox. Exit the program by clicking “Exit game”.

Remaining two menu buttons can be used to launch the game. By clicking “Start game” a new labyrinth is generated, with settings as set in the options menu, and a player character is placed inside it. After this the timer starts to decrease and the player can control the character using W, A, S and D -keys. Press W to move up, A to move left, S to move down and D to move right. Run through the labyrinth until you find an edge without a wall. Run

through it to complete a level. Wooden bridges represent overpasses in game and when a player is on an overpass, the character can only move in the direction of the overpass to not allow the player to “drop down” from the overpass. The player is then awarded points based on the current round and remaining time. After this a new level is generated with starting time and number of exits decreased. Players can also choose to “Give up” by pressing Y -key. After this the shortest path to an exit is shown. Path is marked with yellow circles indicating path (Note! Circle dots are not placed on overpasses. If a dot is on an overpass it indicates that the path goes under that overpass.). Player character indicates the starting position of the labyrinth. The map can be moved with W, A, S and D keys in case the whole map does not fit on your screen. Players can then continue to save their score by pressing space -key. After this player can write his/hers name into the associated text box and press enter to save their score. If the box is left empty, no score will be saved.

“Custom levels” menu button can be used to access and load saved labyrinths. Each saved labyrinth is displayed as a button which can be clicked to load the level. After this the game functions similarly to as it would when starting a new game. A labyrinth can be saved using the in-game menu on the top left of the screen that is displayed when a labyrinth is loaded. Access the menu by clicking “Game”. In-game menu can be used to generate a new game with “New game” -button, give up and solve the level with “Solve -button, Save a level with “Save labyrinth” -button, load a saved level with “Load labyrinth” -button and exit to main menu with “Quit” -button. After the player clicks “Save labyrinth”, they can save a labyrinth by writing a name for it and pressing “enter” -key (Note! If a player writes a name that is already in use, then that level is replaced.). “Load labyrinth” functions similarly to custom levels menu except here players can write the name of the labyrinth they want to load and press enter -key to load it. “Solve” functions identically to pressing Y -key.

Program structure

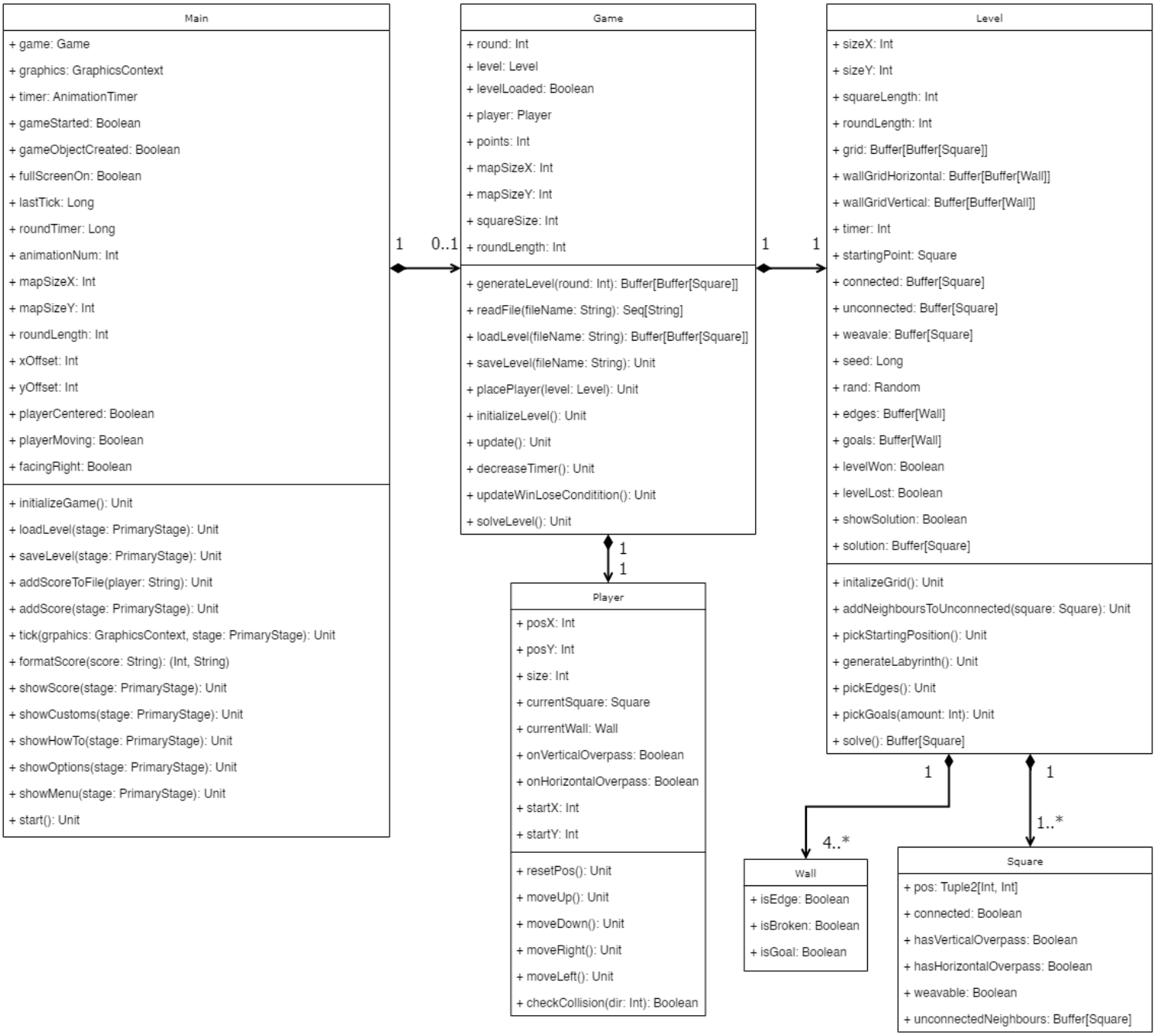

Figure 1. UML class diagram.

The program is structured into classes as presented in UML class diagram (Figure 1). Classes include Main, Game, Player, Level, Wall and Square. Main class's purpose is to run the game and draw a picture. It is also responsible for all the menu functions of the program. Game class runs the overall game, mainly updates game loop and interacts with player and level. Player class represents the player's movable character and its main purpose is to handle player movements and collision checking. Level class represents a specific labyrinth. It handles level generation and level specific tasks such as timer. Wall represents all of the

walls in a level, both invisible “broken” walls and regular walls. Square represents a square in a level that a player can walk on.

Main object is firstly responsible for launching the game. Its most essential methods are start and tick. Start handles starting the game and also tracks player input. It overrides the method of JFXApp3 of ScalaFX library and handles starting the app. When first opened it only starts the menu using showMenu method, but when the actual game is started it will create the in-game menu and configure user inputs. Tick method is called constantly by a timer that is AnimationTimer from the ScalaFX library. Tick then handles everything that is needed for the constant updating of the game. This also includes drawing a picture on the screen. Tick method draws a different picture based on multiple factors including game state, player position and level layout. It also constantly checks if a level is lost or won. Other methods of main include methods starting with “show” that handles menu functionality by changing the scene. In addition to variables in UML, main also has multiple variables for all of the images in the game including different variations of player model that are required for the animation effect. Animations itself are also done inside the tick method by changing the picture rapidly after every odd tick.

Game object runs the game loop and handles interactions between level and player. It also keeps track of information that is related to one game instance instead of level such as points, round number and round length. Its most important methods are generateLevel, saveLevel, loadLevel, readFile and update. Method generateLevel generates a level and calls all methods of the Level class that are required for it. It also calls some other helper methods of game class such as initializeLevel. It also handles file I/O with the readFile method that reads contents from a .txt file. This method is used by both loadLevel and saveLevel, which save and load information of a level from a .txt file. Finally the update method is used to update the game loop, which mainly includes updating win and lose conditions since movement is handled by Player class.

Player class’s main purpose is to handle all tasks related to the player character. This includes moving the player and checking collisions when doing so. Moving the player is done with methods moveUp, moveDown, moveRight and moveLeft. Each of these methods first calls the checkCollision method before moving the player to determine whether the player's movement is obscured by a wall.

Level class handles everything that is specific to a specific labyrinth. It also includes methods that function as the level generation algorithm. These include initializeGrid, generateLabyrinth, pickStartingPosition and helper methods addNeighboursToUnconnected, pickEdges and pickGoals. All of these methods combined are used to generate a unique labyrinth which uses seed from systems time. Another significant task of level class is the solve method that is used to generate the shortest solution to a labyrinth.

Wall and Square classes are used to represent elements of a labyrinth. As suggested, the wall class represents every wall between squares, be it invisible “broken wall” or regular wall. It also tracks which walls are on the edge of the map and which are goals. Square represents a square that the player character can run on. Each square tracks its own location and whether it has an overpass or not. It also includes some variables that are used in level generation such as weavable and unconnectedNeighbours.

Algorithms

Game loop algorithm

The most integral algorithm for the game to function will be the game loop algorithm implemented in game class that updates the game state constantly while the game is running. Algorithm updates the image that is drawn and tracks game state. Algorithm will first update all the states necessary taking into account inputs by user and then render the game in the main object. Main class will be responsible for tracking user inputs and handling menu functions and launching the game. Game class will then be responsible for running the game itself.

Labyrinth generation algorithm

The other important algorithm is the level generation algorithm. It takes into account parameters for map size and amount of exits and generates an appropriate maze. Algorithm generates a “perfect labyrinth” (Think Labyrinth, 2022) meaning that there are no inaccessible areas or loops. Generated labyrinth has a so-called “weave” dimension, meaning that there are overpasses and underpasses in addition to regular paths. The algorithm takes inspiration from Prim’s algorithm to generate mazes (Think Labyrinth, 2022).

To generate a labyrinth with overpasses and underpasses, the algorithm first starts with a two-dimensional buffer of squares that contain all of the squares inside the level. We also maintain two lists of squares: connected and unconnected. The connected list contains squares that have been connected to the labyrinth, while the unconnected list contains squares that haven't been connected yet.

To begin generating the labyrinth, we randomly select a square from the buffer and mark it as the starting point for the labyrinth. We then add this square to the connected list and add all of its unconnected neighbors to the unconnected list.

Next, we randomly select an unconnected neighbor from the current square and check if it's on the edge of the grid to avoid creating overpasses or underpasses that go beyond the edge of the grid. We then check if the square on the other side of the potential underpass or overpass is already visited or not. If it's already visited, we mark that there cannot be an overpass or underpass. If not, we mark that it can possibly create an overpass or underpass.

We then repeat this process for all of the unconnected neighbors of the current square and add those that can be “weaved” to a list of weavable neighbors. We randomly select a weavable neighbor and by random chance either just break down the wall between the neighbor and the current square or create an overpass or underpass over it. We then add the neighbor to the connected list, add its unconnected neighbors to the unconnected list and do the same for the “destination square” of the potential underpass or overpass.

We continue this process by randomly selecting a square from the unconnected list, adding its unconnected neighbors to the unconnected list, and selecting a weavable neighbor to create an overpass or underpass over. We repeat this process until we have no squares left in the unconnected list.

Resulting from the level generation is a new two-dimensional buffer of squares that can then be set as the level. By doing this new algorithms for generating levels can be added easily as long as they have the same return value type. Algorithm also changes a variable to indicate a starting position for the player character.

Solving algorithm

In addition to labyrinth generation the game has a separate solving algorithm. Algorithm takes inspiration from the “Recursive backtracker” algorithm introduced on Think Labyrinth. At the start of the algorithm, the player's starting position is set as the starting point, and the algorithm randomly selects a neighboring square to move to. The algorithm records the movements made in a buffer. When it encounters an intersection, the algorithm records it in a separate buffer, in which each instance contains all the possible options for the next square to move from that intersection. Then when the algorithm reaches a deadend, the algorithm takes a different option from the latest intersection, deletes the moves that led to the deadend from the movement buffer and deletes the choice it took from the intersection instance saved in the intersection buffer. When the intersection in the buffer has no more choices left it is deleted from the buffer. This process is repeated until the algorithm reaches the end goal. Once it reaches the goal the resulting movement buffer is returned and can be showcased to the player. The length of the solution will be used to calculate the time for a level timer. Since the format is again standardized to a buffer of squares in the order of movements made, additional solving algorithms can be easily implemented as long as they return a buffer in the same format.

Data structures

For my level generation, I use a two-dimensional buffer to store the labyrinth squares. This is a buffer of buffers of squares that makes it easy to access each individual square. Level generation also requires three lists: one to store connected squares, one to store unconnected squares, and finally one to store weavable neighboring tiles to create overpasses and underpasses. The first two lists are integral to generating a labyrinth, as I stated above. Both of these lists need to be dynamic data structures, such as buffers, as their size will change throughout the level generation algorithm. The third list also needs to use a dynamic structure since its size will change based on the number of weavable neighbors. The labyrinth-solving algorithm also requires dynamic structures to save the movements of the solver and the intersections it encounters. I believe that other structures, such as a map that includes the squares as values with keys being coordinates, would unnecessarily complicate the application, and thus I did not consider other data structures.

In addition to the solving and generating algorithms, the game itself requires some additional data structures. For example, the player's position needs to be saved in a pair of integers representing the x and y coordinates in the labyrinth. The current game state needs to be stored in variables that include such details as the current round, the level state (ongoing/win/lost), and the level timer. The menu also needs some additional variables to store its functions, and file input/output uses streams to read and write to files. Many boolean

type variables are used to create flags that store information, such as whether a game has been created or not. Finally, the textures are saved as images to make the game look good.

The program also uses many data types of the ScalaFX library. For example GraphicsContext data type is used to render an image on the screen. AnimationTimer data type is used to update the game loop each tick. The ScalaFX stage is used to keep track of game states and different states are displayed using scenes. ActionEvents are used to take inputs from the user. In addition to ScalaFX, some data types of Java are also used. For example GraphicsEnvironment and DisplayMode are used to calculate screen size and BufferedWriter and File types are used for file I/O.

Files and internet access

First of all the game uses .txt files to save levels. As the level generation algorithm takes a random seed number to generate the level we can only save this seed and let the game generate a level based on the seed number. This makes sharing mazes a lot easier as well. In addition to seed number we store additional information of the level that include level size, square size and starting round length. The game saves each labyrinth into a separate .txt file. The game uses these files to display saved levels through custom levels in the menu.

In addition to labyrinths the game saves scoreboard data. This is a simple txt file, with each row representing a player's name and score. The game showcases these scores and puts them in order with scoreboard menu functionality.

In addition to these files, the game uses a bunch of assets from the opengamert.org site that offers open source textures and assets for games. This project only uses images that include textures for level and playable character. The game does not use internet access.

Testing

The program was mainly tested through the graphical user interface. As the game is very graphically heavy I could test that the labyrinths are generated correctly by launching the GUI. In addition to testing with GUI, few unit tests were created to test that labyrinth is generated. The program passed all of these unit tests. The testing was pretty straightforward during the implementation since I had some sort of working graphical interface from a very early point.

Known bugs and missing features

I've encountered one instance of a player being able to go through a wall when interacting with a corner by moving back and forth repeatedly. However, I found that this is a pretty rare bug, since I haven't been able to recreate it in some instances. There's also an issue that the program will slow down quite heavily if the map sizes are too big. The actual gameplay will work no problem, but when the result is shown it gets slow. This is due to the fact that more

squares are drawn with map overview than in game screen. There's also an issue where the scoreboard does not properly update when the .jar file is used to launch the game. To make the scoreboard work properly please test it with running the game through the Main.scala file inside source code.

One missing feature that I hinted at in my plan is the ability to zoom in and out. It did not fit in my final design that had the camera following the playable character. It could have been implemented differently, but I did not think it was that important.

3 best sides and 3 weaknesses

One thing that I'm pretty proud of is the idea to use seeds in generation so that they can be later saved when implementing level saving. I planned for this from the very beginning which allowed me to create the labyrinth generation in a way that made this possible. This saved me a bunch of work with implementing labyrinth saving and also makes it more efficient.

Also I think it's worth mentioning here since it may not be obvious that every square on the map has a path to an exit. This means that my algorithm does not create any dead-end squares that have no path to the finish line.

Finally I would like to mention that the actual in game drawing is efficiently done. Only those squares that are in a certain distance from the player are drawn which saves a bunch of resources. This is not obvious from playing the game which is obviously by design.

One of the main weaknesses in my program is its structure. Many of the functionalities of the Main object should have been moved to a separate class to make main less complex. This decision was made due to the difficulties with working with ScalaFX since I encountered some obstacles when initially trying to split drawing to a separate class. Also in terms of method structure, many of the methods should have been split into smaller helper methods to make them more clear. Some of the methods have repetition which could have been avoided with helper methods.

Second weakness is a lack of proper unit testing. If the program would be expanded upon then the lack of unit tests would expose the program for bugs with expansion. There should be way more unit tests to test every method and make the program safer to expand.

Finally I will mention the problem with drawing the solution. When drawing the solution and "zooming out" to a different view, every square is drawn. This makes it so that when the map is very big the program becomes very laggy.

Deviations from the plan, realized process and schedule

Working schedule along with the work order followed my plan pretty well. The first thing that was implemented was level generation with simple GUI, followed by movable player

character and game loop. Then level saving and loading was added along with the scoreboard and main menu. Finally the graphics were overhauled and the following player character was implemented. The only deviation from the schedule was the moving view that I forgot from the plan. This was then done in the polish phase during the last two weeks.

The time estimates were by average bigger than I ended up needing for implementation. My estimations were even double the time spent especially during the early phases. However, towards the end the polishing took more time than expected. So the main thing I learned in terms of project planning is to leave even more time for polishing since there's probably always a big feature, such as a moving view in my case, that hasn't been thought of during planning and needs to be implemented.

Final evaluation.

Overall, I'm very happy with my project. Even though the structure for code is quite messy I ended up achieving all the goals that I set for my project. It functions as I visioned at the beginning and nothing major was left unfinished. I could have made the main menu more pretty, but I think this is enough for this project. If the project would be expanded upon then perhaps there could be added multiple labyrinth generation algorithms with ability for the player to choose between options. There's also the rare bug that allows players to enter walls that could be researched and fixed in future.

If I would start the project again then I would try to make the project structure more clear. I would try to split rendering to another class with it having GraphicsContext as its constructor parameter.

References

Foxy 2D Character asset. overcrafted. Available at: https://opengameart.org/content/foxy-2d-character-asset

JavaFX library

Large nature background. Julien. Available at: https://opengameart.org/content/large-nature-background

LPC Wooden Bridge Rework. AntumDeluge. Available at: https://opengameart.org/content/lpc-wooden-bridge-rework

Opengameart. Available at: https://opengameart.org

Scala standard library

ScalaFX library

Seamless Grass Texture II. athile. Available at: https://opengameart.org/content/seamless-grass-texture-ii

Seamless Wall. GrumpyDiamond. Available at: https://opengameart.org/content/seamless-wall

Think Labyrinth. [referenced 18.4.2023]. Available at: https://www.astrolog.org/labyrnth/algrithm.htm.

Appendixes

Source code included in directory src/main/scala/labyrinthGame

Figure 2. Main menu screen

Figure 3. In-game screen

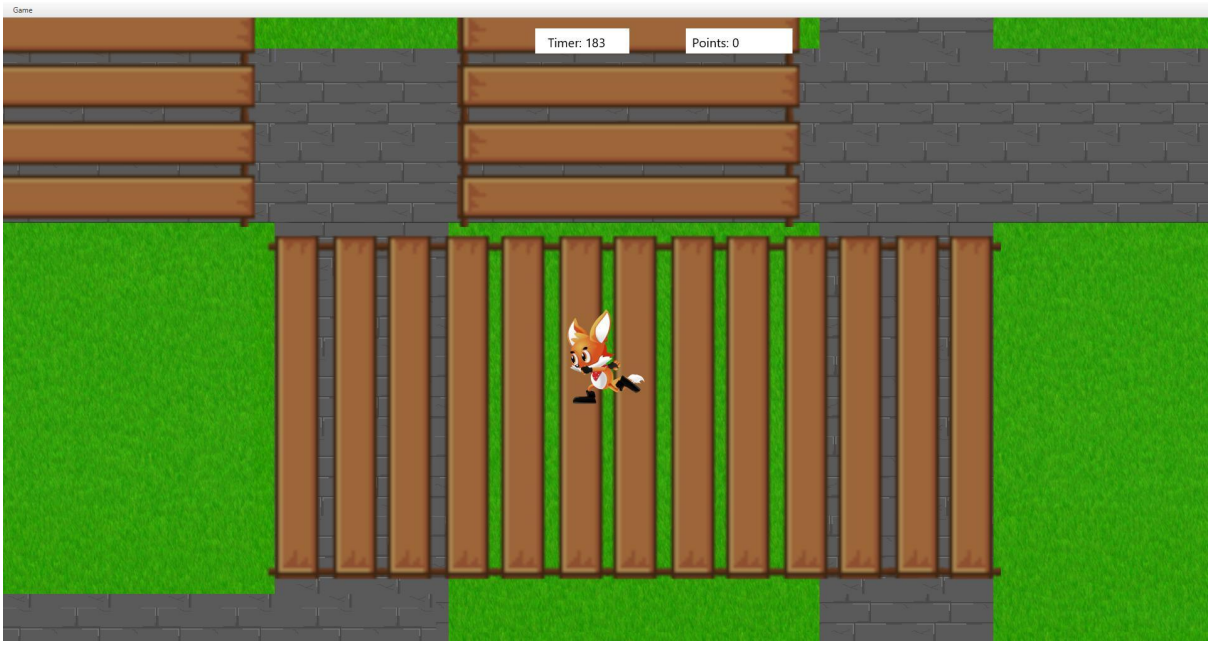

Figure 4. Character on an overpass

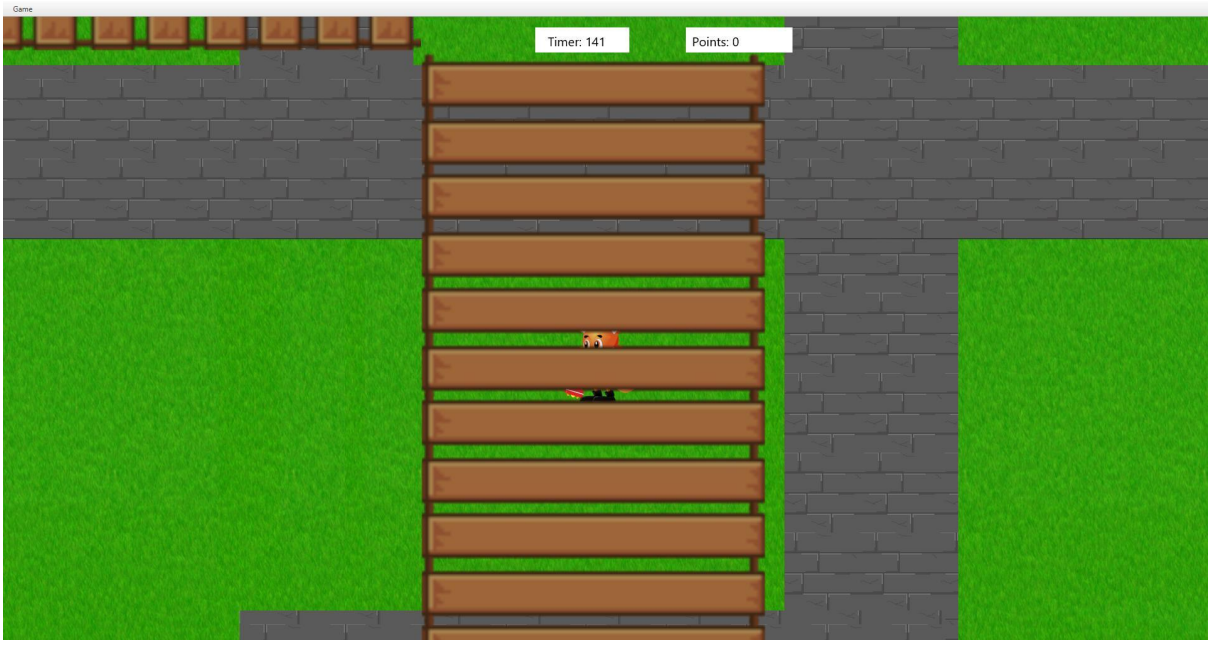

Figure 5. Character under an overpass

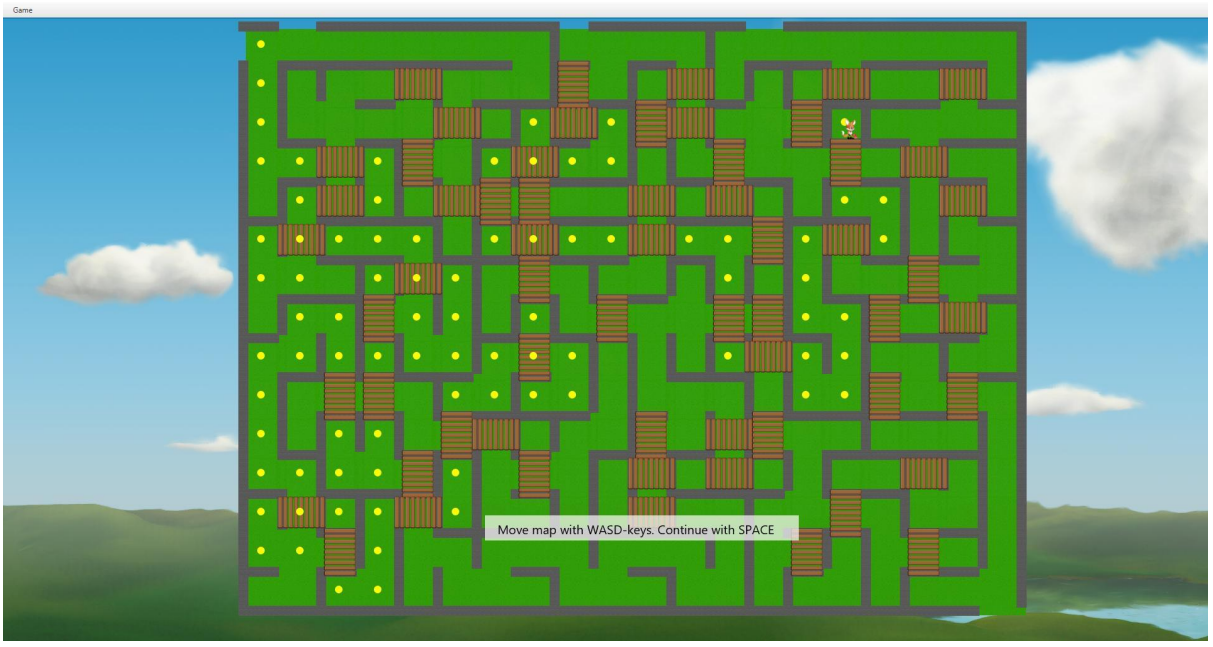

Figure 6. Solution screen