Project technical plan - Labyrinth

Severi Koivumaa
100133819
Degree program: Bachelor’s and master’s in science and technology (Computer science)
Year of studies: 2022
Date: 19.2.2023

Class structure

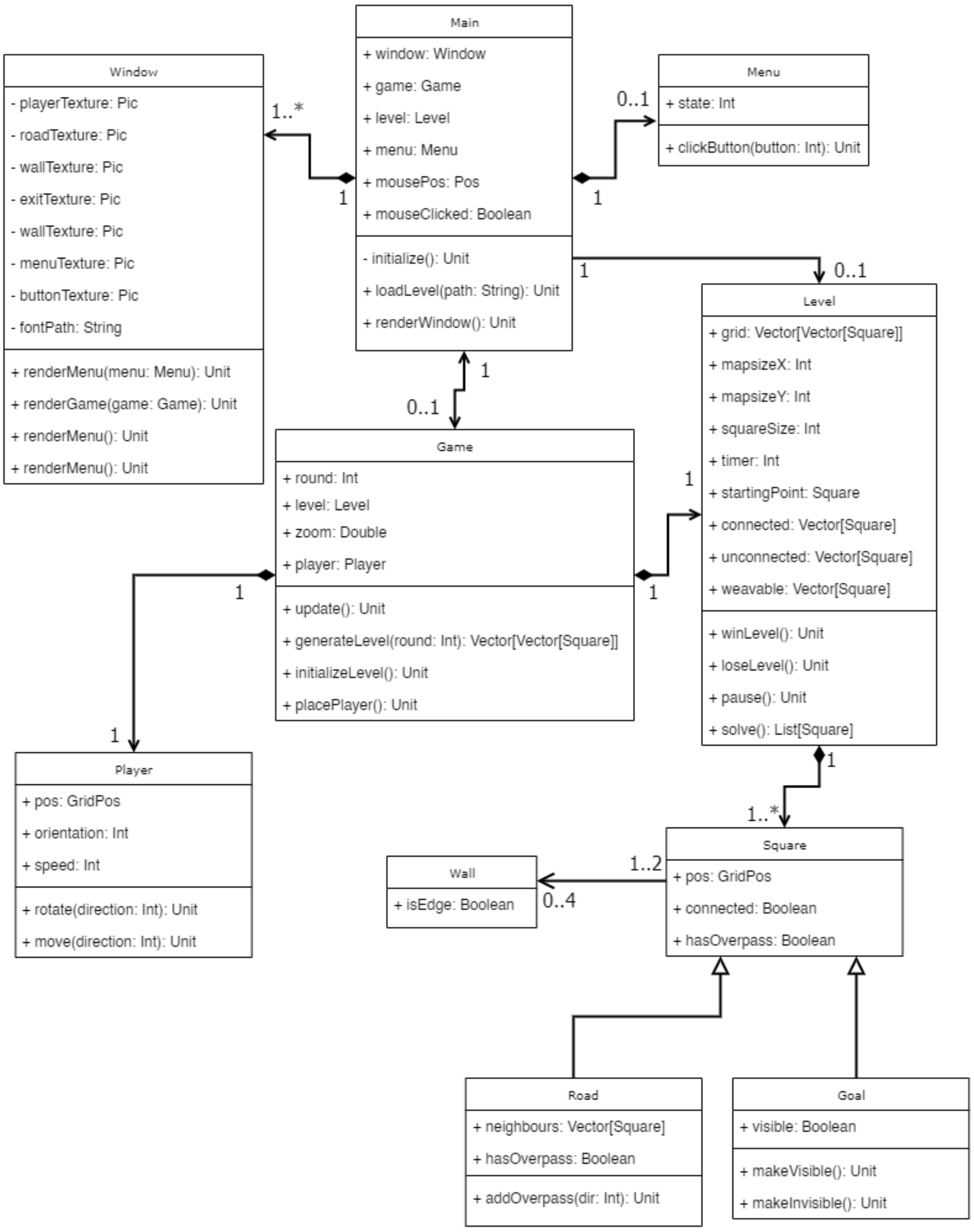

Figure 1. UML class diagram.

The most important part of the program is the game loop that is handled by game class. The most integral method to handle game loop within game class is a method called update. Update is called constantly when the game is active and it will in turn call other methods to update the game's state. It will then in turn call main to render a picture using a window class. Game class is also responsible for one of the most important algorithms: level generation. It will then track player movement and overall game state.

Main object is used to launch the game and run it. It also acts as a sort of hub for other classes that handle specific tasks like menu class handling main menu related tasks and window handling GUI and textures. Main will also handle tracking player inputs which are then passed to menu and game classes respectively.

Level class is used to handle tasks specific to individual level. This includes square size, map size, winning/losing conditions including timer and level solver function. It will use square class to represent individual squares in a level which can be either road or a goal class. Additionally each square initially has four walls for each of its sides that can then be broken down when generating a level.

The relationships between the classes are as follows. Window is used by Main to render the graphical user interface. Main uses Menu to run the main menu. Main uses the Game to run the game loop. Game uses Level to generate levels and manage level state. Level uses Square and its subclasses to model the level. Square uses Wall to display walls surrounding it. Game uses Player to place and track playable character in levels.

Other possible solutions could maybe introduce even more inheritance to the class structure. For example all individual objects inside a level like Square, Player and Wall could inherit a class called LevelObject which would handle generic tasks to interact with level and to indicate that they are all objects that appear in a level. However, I think that all of these classes have distinct enough purposes to justify them being separate from each other.

Use case description

An user can use the program when playing the game. Typical actions include first opening the main menu by launching the program. When the program is launched the main object creates a menu class instance that includes the menu functions. Main also creates a window class instance which draws a GUI based on inputs of menu instances. Menu instance's state updates as the user interacts with the menu and window object is updated constantly. When a user opens a level, the game class creates an instance of the game class which then initializes the game by creating a level class instance and player class instance. Player is then placed in the level and when the user sends inputs, the player is moved to the level. When a level is cleared the game class will create a new level instance that takes a bit more difficult parameters compared to the last level.

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. This function may be split into several other functions to make it simpler such as “updatePlayer” or “updateView”. Algorithm will first update all the states necessary taking into account inputs by user and then render the game using methods of window class. 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 and window class will handle graphical user interface.

Labyrinth generation algorithm

The other important algorithm is the level generation algorithm. It will take into account at least parameters for map size and amount of exits and generate an appropriate maze. Algorithm will aim to generate a “perfect labyrinth” (Think Labyrinth, 2022) meaning that there are no inaccessible areas or loops. Generated labyrinth will have a so-called “weave” dimension, meaning that there are overpasses and underpasses in addition to regular paths. The actual algorithm will take 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 vector 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 vector 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 vector 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 will also change a variable to indicate a starting position for the player character. The game will be able to start without a set player start position to avoid crashes. When spawning a player the game will first check whether a set starting position exists and if not it will pick a random square.

Solving algorithm

In addition to labyrinth generation the game will have a separate solving algorithm. Algorithm will take 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 vector. When it encounters an intersection, the algorithm records it in a separate vector, 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 vector and deletes the choice it took from the intersection instance saved in the intersection vector. When the intersection in the vector has no more choices left it is deleted from the vector. This process is repeated until the algorithm reaches the end goal. Once it reaches the goal the resulting movement vector 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 vector of squares in the order of movements made, additional solving algorithms can be easily implemented as long as they return a vector in the same format.

Data structures

With my level generation I will use a two-dimensional vector to store the labyrinths squares. This will be a vector of vectors of squares, that will make it easy to access each individual square. Level generation also needs three lists: one to store connected squares, one to store the unconnected squares and finally one to store weavable neighboring tiles to create overpasses and underpasses. First two of the lists are integral while generating a labyrinth like 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. Third list also needs to use a dynamic structure since its size will change based on the number of weavable neighbors. Labyrinth solving algorithm also similarly requires dynamic structures to save the movements of the solver and the intersections it encounters. I think that other structures such as a map that includes the squares as values with keys being coordinates would just make the application unnecessarily complicated.

In addition to solving and generating algorithms the game itself requires some additional data structures. For example player position needs to be saved in a pair of integers representing the x and y coordinates in the labyrinth. Current game state needs to be stored in variables that include such details as current level, level state (ongoing/win/lost) and level timer. Menu also needs some additional variables to store its functions and file input/output uses streams to read and write to files. Finally the textures also need to be saved as sprites to make the game look good as well.

Files

First of all my game will require a file to save levels. As the level generation algorithm will need to take a random seed number to generate the level we should be able to 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. The game would save all of the saved labyrinths in one .txt file where it would store one seed on every row. The game could then display all of these levels in the custom menu section. As a backup plan the game could save the labyrinth file in binary values in a .txt with characters representing different squares (i.e. in which direction it has walls). For example a “1” character could represent a square that has only one opening from it to the left and “2” representing a square with both sides open etc.

In addition to labyrinths the game also needs to save leaderboard data. This can be a simple csv file, with each row representing a player’s name, level reached and score. The game can then showcase up to ten of the best scores stored in the csv file.

Finally a configuration file is needed to save options such as controls and level generation details. File format can be a simple text file with a key-value pair on each of the rows.

Schedule

Weeks 8 - 9. Develop level generation. 20 hours
Week 10. Develop player movement and game loop. 15 hours
Week 11. First playable version. 15 hours
Week 12. Develop map saving and loading from file. 15 hours
Week 13. Develop leaderboard and all remaining menu functionalities. 10 hours
Week 14. Polish graphics and gameplay. 20 hours.
Week 15. Final week. Polish. 10 hours.
Week 16. Return project

Testing plan

System testing

For labyrinth testing I should test that the level will generate a correctly sized labyrinth based on the parameters. For all of the sizes the labyrinth should have a start point and at least one ending point, while ensuring that the labyrinth is indeed solvable. I should also make sure that the labyrinth generated is different and that it doesn’t generate the same labyrinth all the time. I should also test that the solving algorithm works for different labyrinths.

For the actual gameplay I should test that the movement works smoothly and correctly, so that the player character cannot go through walls but can go under underpasses. I should also test that a level is lost when the timer runs out and that the level is won when the ending point is reached in time. I should also test that the new level is generated after completing one level with timer resetting.

I should also test that the main menu works as intended. This includes testing that changing options actually affects the game, that the leaderboard shows as expected, that custom levels show up as expected, “how to” -section displays information about the game and that the game launches normally. I should also test that file saving and loading works and that files have the levels listed in correct format. For error handling I should try to input incorrect inputs in all states of the program to try to cause an error and try special cases like when launching a labyrinth with incorrect parameters.

Unit testing

Most integral unit testing would probably test the method that generates the labyrinth. The key inputs for this method would be the parameters for the labyrinth such as width, height and the amount of exits. The method should return a two-dimensional vector containing the squares of the labyrinth. The expected result should be that there is at least one single path from the beginning to the goal and that there are no isolated sections.

There should also be a unit test for labyrinth solving algorithm that tests whether its output is in the correct format. This being a list of squares that represent the route taken by the solver. It should also test that the return value is not empty e.g. it has at least one square value. In order to carry out these tests the unit tests would require to create a sample labyrinth and then do these tests appropriately.

References

Think Labyrinth. [referenced 19.2.2023]. Available at: https://www.astrolog.org/labyrnth/algrithm.htm.