Tower Defense Documentation

1 Details

Kasper Henriksson
647214
Computer Science
2017
27.4.2020

2 General Information

Tower Defense is a game where attackers are trying to get through the player's defenses along a pre-planned path, and the player needs to stop them by adding defensive towers along the route. These towers then kill attacker either by shooting or through other measures, for example freezing them.

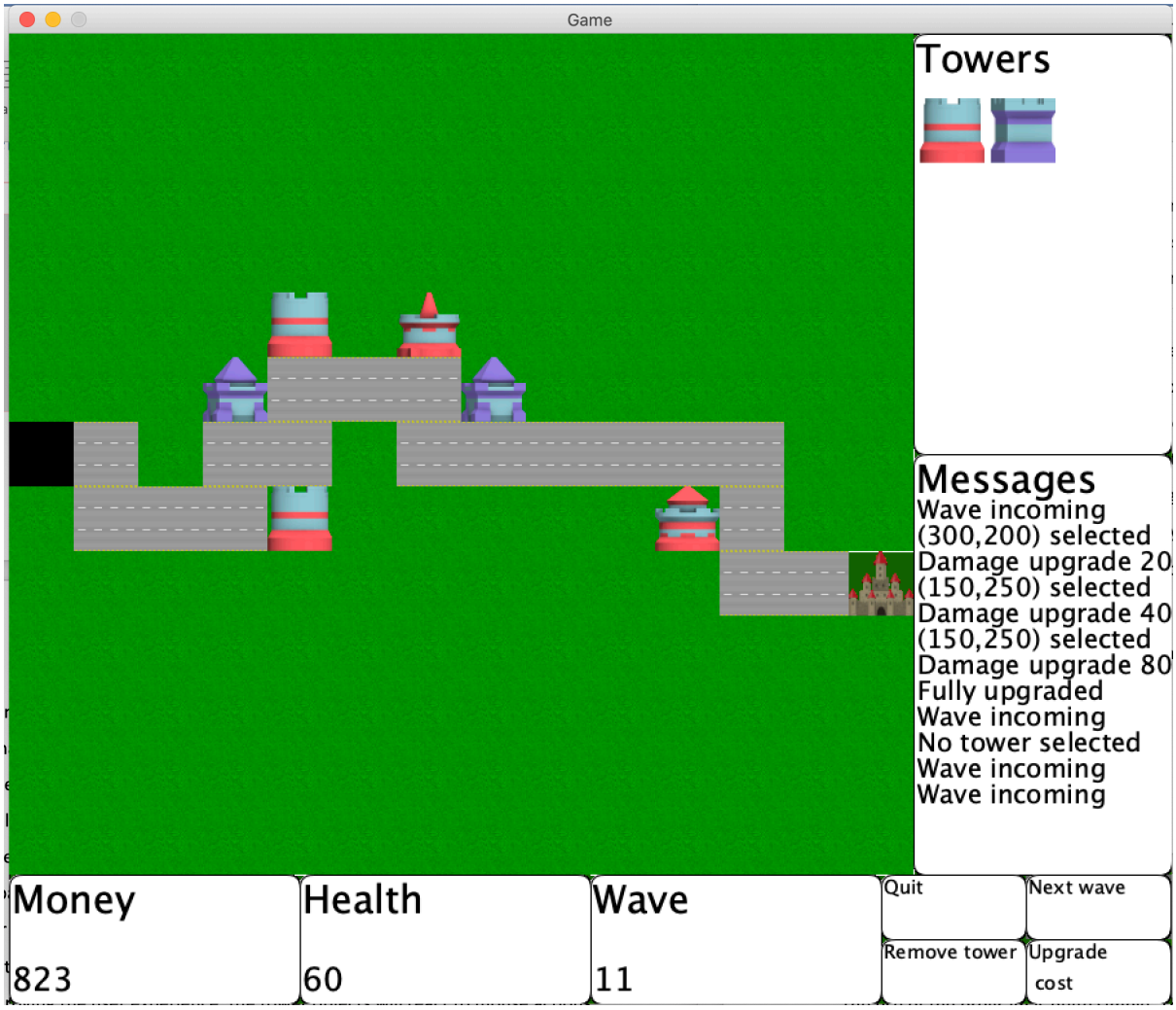
The player starts with a definite amount of lives, for example 100, and if the enemies get through the defenses, the player loses lives, until eventually the player dies completely, resulting in a game over. The player accumulates money by killing enemies, money that then can be used for building and upgrading towers.

The enemies usually come in waves, thereby giving the game a clear "level" basis and a feeling of accomplishment by completing them. The game can either go on forever, this however requires upgrading towers, or then if the player beats 10 levels they win the game. This game has been implemented to be an infinite tower defense game, where there can always be more waves, and no timing to ensure that the player has enough time to decide on where to put towers. If the player wants to remove and sell a tower, however they only get 80% of the purchase value back.

3 Use of Program

The use of the program is quite simple. The main file is the *Game.scala* file, so simply run it by right/double clicking it and selecting *Run As -> Scala Application.* First of all, you get to choose the map, and by clicking one of the map buttons the game starts. But no need to hurry, because you get to choose when the waves come.
The lower bar tells you the amount of money, health and wave number, and the right lower corner are buttons that can be clicked, with either quit, remove tower, starting the next
The game doesn't have any kind of timed waves, as it gives you the opportunity to choose your towers locations carefully.

The first step is to choose the different towers to set out. All the green areas are markable, and towers can be put everywhere except on the castle, the black spawn for attackers and on the routes. When clicking a tower in the right sidebar, it will be selected, and then you can simply assign to a certain cell on the map. By selecting a tower on the map it's possible to upgrade the tower and their damage.
When hovering over a cell that is buildable, the cell sides will light up to make it easier to see which cell you're on. To start the first wave, simply click "Next wave" in the lower right corner. To remove a tower, just select it by clicking on it and then click "Remove tower".



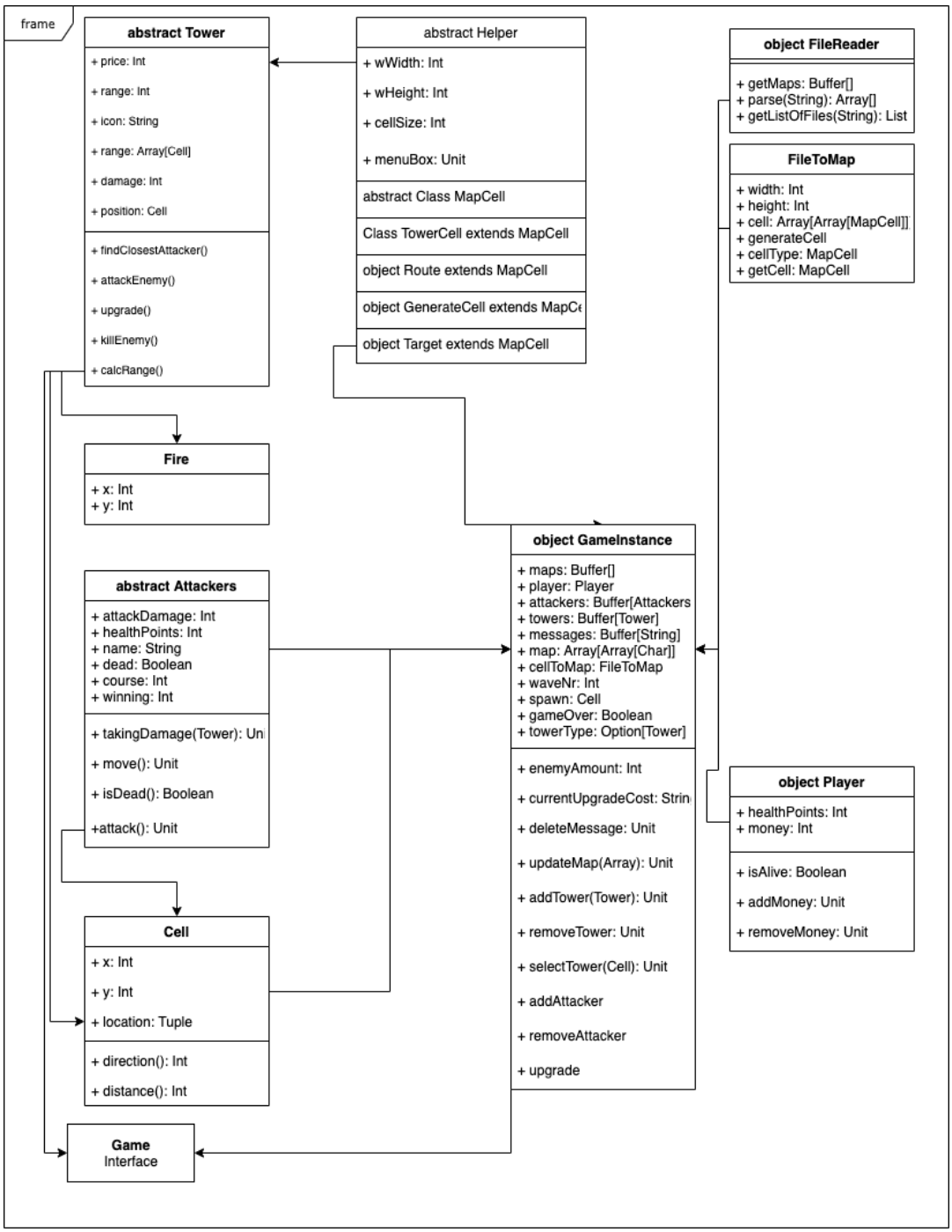
4. Game Structure

4.1 Packages

There are five different packages for this program: **Game, Attackers, Gamemaps, Tower and Test.** The Game package contains all the info related to running the game and handling the instance, along with the player object. Attackers handles everything related to the enemies. Gamemaps contains everything reading in the map files. Towers handle player

towers and the fires that are shot. Test contains the unit tests. In the next chapter, the classes will be described more thoroughly, except for units tests as it's only unit tests.

4.2 Classes



4.2.1 Helper

Helper is an abstract class that contains all the constants that are used when drawing the game using processing. In the same class, we also got the MapCell abstract, that helps to

keep the cells of the right types. This doesn't contain any important methods, but set the state of the interface, for example window width and height, and also makes it possible to use Processing functions in the Tower class.

4.2.2 Game

Handles the Processing and drawing of the game. The most important functions here are the *draw(), mouseClicked(), mouseMoved(),* and *drawCell.* Draw is the main Processing function, that draws the game board. mouseClicked handles button clicks, for example the next wave button. The drawCell draws the cells according to the parsed .txt files, and shows the grass, routes and towers.

4.2.3 GameInstance

GameInstance is the "text" class for the game, this is where all the magic happens related to adding and removing towers, and it keeps track of the player statistics as well. All the functions in this class are of high relevance to the game.

4.2.3 Attackers

Attackers contains the abstract class for the attackers, and the basic algorithms and details of the different enemies that attack the tower. The abstract overhead class Attackers contains all the functions, and the BasicAttacker and AdvancedAttacker only have different damage and healthpoints. The most important function is the *move()* function, a pathfinding algorithm for the attackers.

4.2.4 Tower

Similar to the package class of the attackers, the abstract class for towers and details of their damage. However, the difference here is that the towers handle their own function for drawing to the board. This was done to keep all the drawing in different classes, but unfortunately I ran into problems as the attackers were moving. This could have been built in the game interface class, but I wanted to try out something else from the beginning. The abstract class contains most of the information, and the most important function is *findClose()* as it searches for the closest attacker and attacks.

5 Algorithms

5.1 Getting the map files

To make the game more customizable, I made an algorithm that reads in all the files in the map folder, to make it possible to choose between different maps. The first reason I wanted to make this algorithm was actually for a unit test to check that the maps are of the right

size, but this could also be used in the game interface. If I hadn't created this one the only way would have been to change it manually in the code.

5.2 Pathfinding

The pathfinding algorithm was a little bit harder to code, as it required some math and thinking and bringing it together with the cell build-up. How it works is that it checks the course of the attacker, and decides on what kind of cell the one in front is, if it's not a route then it checks one side, otherwise the opposite to that one. This could have been done with many different algorithms but this was probably the simplest one.

5.3 Distance

I'm using a simple distance-algorithm for calculating the distance between two cells, which is based on Pythagoras and the hypothenuse. However, due to some problems with Processing and the updating, the towers don't have a specific range for their fires.

5.4 Algorithm for dynamic map-buttons

The user interface is also somewhat dynamic in the beginning. If a map is added to the resource folder with gamemaps, the buttons move accordingly and always stay in the middle, so that the buttons are not in any way bound to a certain coordinate, but if you add a button then the others move as well. This can be seen in Game.scala in drawChooseMap() and in the if-else statements for the mouse clicks. If this wouldn't have been implemented, adding maps would have made the new button go out of screen.

5.5 Hovering

The hovering algorithm in *mouseMoved()* calculates where the mouse is currently, and lights the boundaries for that cell up, so that you can always see where the current tower will be placed.

5.6 Other

Other notable algorithms are the one adding towers which checks if cells are buildable ones, and the upgrade towers.

6 Structures

This game mostly stuck to built-in Scala structures such as buffers, arrays and maps, but the algorithm that finds the map resources also uses a List. Buffers were used to towers and attackers to be able to easily add and remove them, and maps were used to store the images and icons for the towers and attackers, to be able to level them up and change the tower icons between levels. To parse the map arrays were used to make them immutable. When the text files we get a matrice out, but it's of no other importance.

7 Files

7.1 .txt files

The maps are parsed from a .txt file of width 13 and 14 rows. The file consists of 4 different characters: **0, 1,2** and **-.** 0 is the spawn and generate cell for the attackers, and should be placed on the first row of the.txt file. 1 is the route, and these should follow each other to make a simple route. 2 is the castle point and target for the attackers, and – are the buildable cells.

It's easy to make a new one, just add a .txt file to *resources/gamemaps,* and it's usable.

```
1  ------0------
2  ------1------
3  ------1------
4  ---1111------
5  --11---------
6  --1----------
7  --1----------
8  --11111------
9  ------1------
10 ------1------
11 ------1------
12 ------111----
13 --------1----
14 --------2----
```

7.2 .png files

PNG files are used for the towers and attackers icons, and are for now not addable without making changes to the code.

8 Testing

The game has a few unit tests but not too many, most of the simpler algorithms are tested with unit tests such as adding towers, and one more advanced with checking that the maps are of the right size. Testing was mostly done by trying out the game constantly and seeing what works and what doesn't.

If I ran into problem I mostly tried and failed with printing statements to get an understanding of what is going wrong. Some of the drawing algorithms has exception tests, and for that I'm using a *getCause* function that gets the fault that the program ran into. For now, there are no known bugs that makes the program crack.

9 Bugs and missing features

I had a huge problem with implementing the tower range feature and therefore the towers can instead shoot from all over the screen. This may be due to Processing and the way the

towers are implemented in the drawing function, but I was not able to find a problem with it.

Another thing is the time difference between enemies. For now, all the enemies are coming at the same time, and there's no delay between them. I tried lots of things, like calculating the time to add them to the buffer, and to use Thread.sleep(1000), but the problem with .sleep is that it freezes the whole loop and the program is constantly in the same loop. If it would have been implemented with parallell programming it may have been possible to use Thread.sleep.

10 Strengths and weaknesses

One of the strengths lies in the algorithms, three superbly working algorithms are pathfinding move(), the getListOfFiles() that parses all maps in resources/gamemaps, and the button algorithm that makes the buttons dynamic.

The project is also structured very well, with 5 different packages and the division between the programs are in my opinion well-defined and everything is well commented to make sure the user gets an understanding of the program easily. The drawing function is also quite clean, even though the whole GUI-file could be cleaner but Processing makes it a little bit messy.

I think one of the strengths is also that I haven't managed to find any clear bugs in the program, and that it is built up in such a way that there shouldn't be any big bugs causing the game to crash.

One weakness is the fact that I did not manage to program the tower ranges correctly, as I would have wished. I ran into some major problems while trying it out, and the same goes for nothing getting the delay and speed of the enemies to work, as they currently switch between cells.

I would also have wished for more unit tests, but the reason I did not want to implement any more of them is due to the fact that there are quite many simple algorithms and I did not want to use my time to create many small unit tests, but I would have liked to test out bigger concepts. I should also have familiarized myself more with exceptions and how they work. The last weakness is the lack of functions. I would have liked to implement a save function, and functions such as create your own map, and the wave functionality with 10 seconds between waves, but the largest problem here was with the time handling as mentioned earlier.

11 Devations and realized features

I did not deviate too much from my original plan, but some changes were made when it comes to features. For example, I wanted to develop delays between enemies, different

speeds and different tower features, but was not able to. Some reasons may be due to Processing not being completely configurable with Scala.

I also used way more time to unnecessary things and having problems, when I should have focused on getting more things to work. What I did learn however, was that programming should always be done with continuous testing to make sure everything works as planned. This is something that I will take with me in the future.

12 Evaluation

All in all, I'm super proud of how the game looks and feels, even though it's simpler than I had hoped. I've put a lot of effort into it and there are algorithms that I'm very proud of, even though the game lacks some functionality that it should have. If I would start all over, I would probably stick to having all drawing functions in the same class, even though my this time was not to, but it may make the code cleaner. I would also not start from drawing the game, as it made it more of a top-to-bottom approach, and rather create functions that are necessary but not used in the Processing file. I would also improve the wave functionality, and think more of how I implement stuff and develop unit tests for everything. I'd really like to give myself grade 5 due to the hard worked algorithms, even though there's functionality missing, but I hope you will enjoy it anyway.

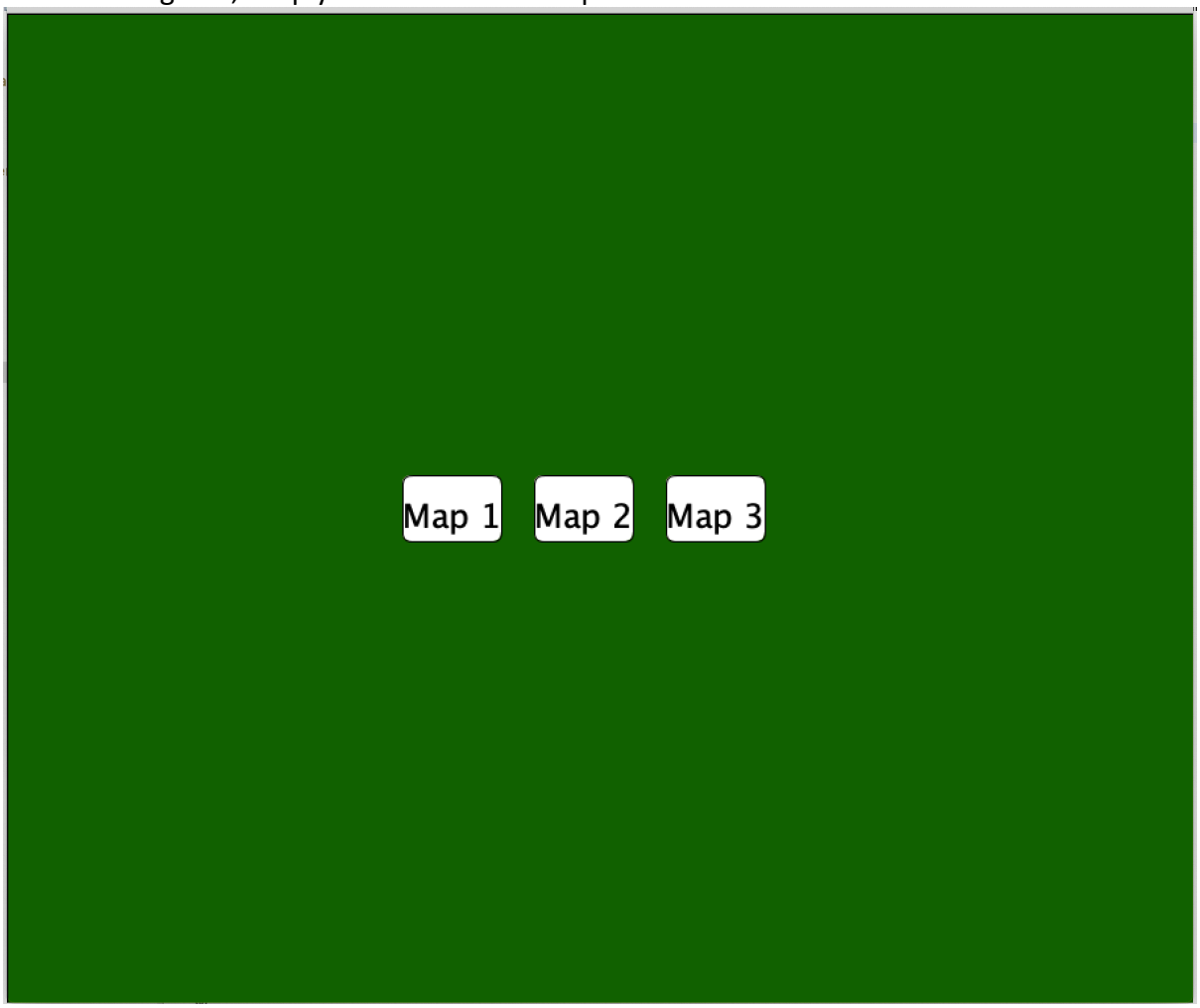
13 References

Processing.org was used a lot, as well as stackoverflow.com. I also discussed some things with friends to get their input, because human resources is the most important resource when coding.

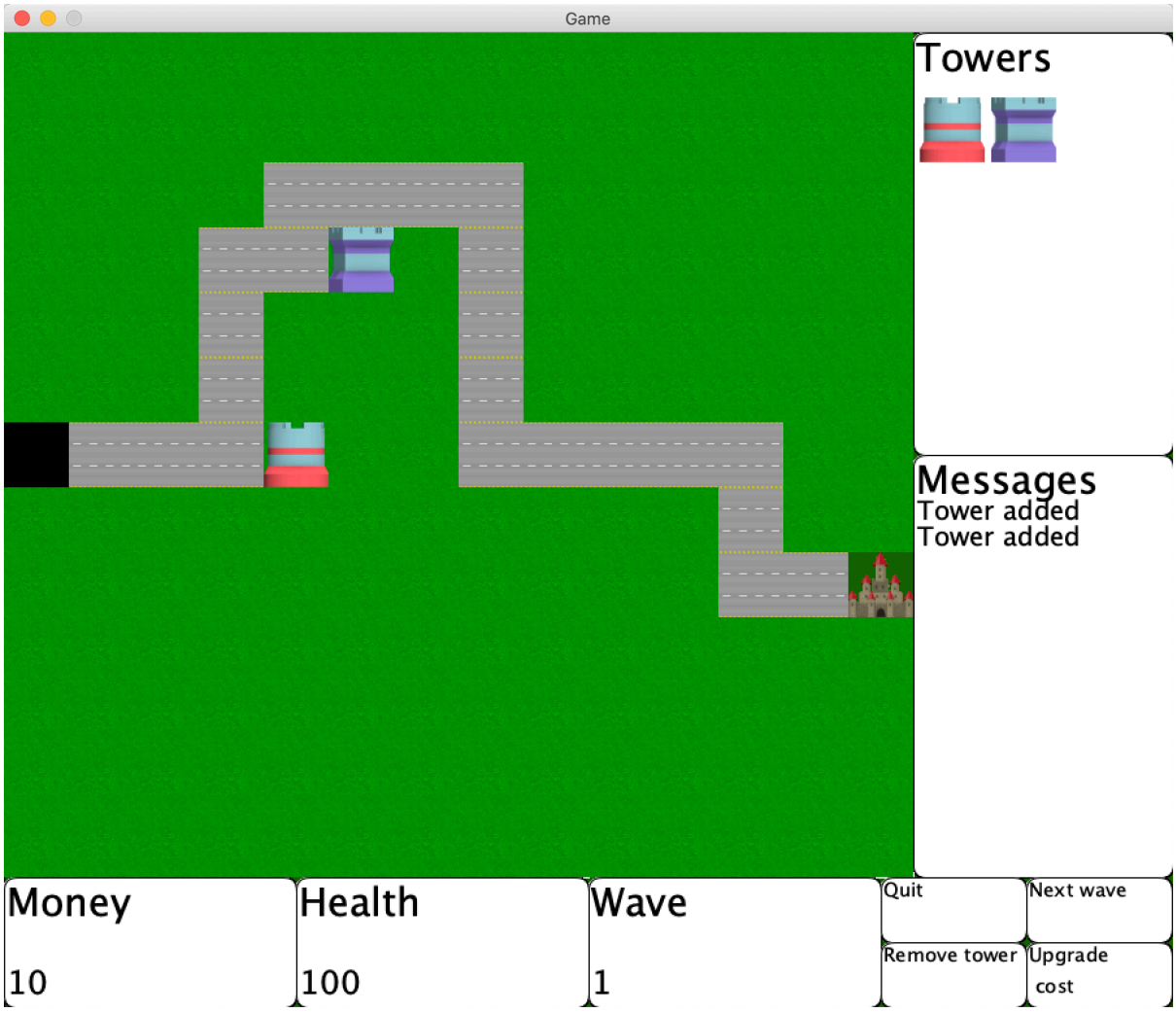
14 Appendix

14.1 Playing the game

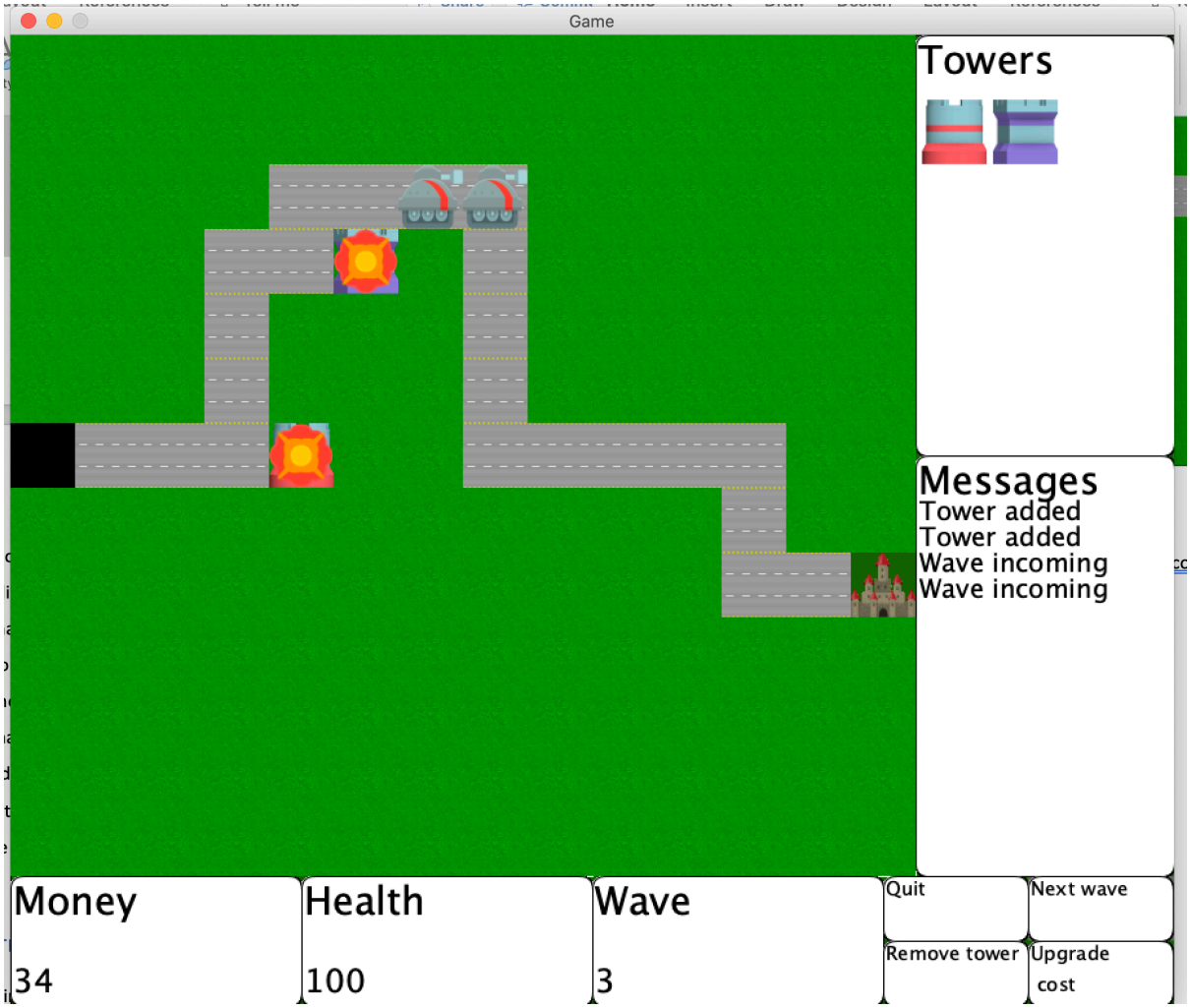
To start the game, simply click one of the map buttons.



There are 3 different buttons built in.



Choose a tower simply by clicking it in the upper right corner, and put it down somewhere on the map. To upgrade a tower, click on it on the map and select “Upgrade” in the lower right corner.



To start the game, click on next wave in the right corner. Voila!