Project document

Personal Information 16.04.2020

Tower Defense

Tommi Vinh Pakarinen, 778109

Bachelor's Degree Programme in Computer Science

1st Year

General Description

The idea is to make a fun tower defence game, where you build towers to kill monsters which generates money for building even more towers. This is the general game loop for every tower defence game, but the details are what differentiate the great ones from the rest. I'm doing a top down view with 2D graphics. The graphics are mostly procedurally generated because variety is important for my project.

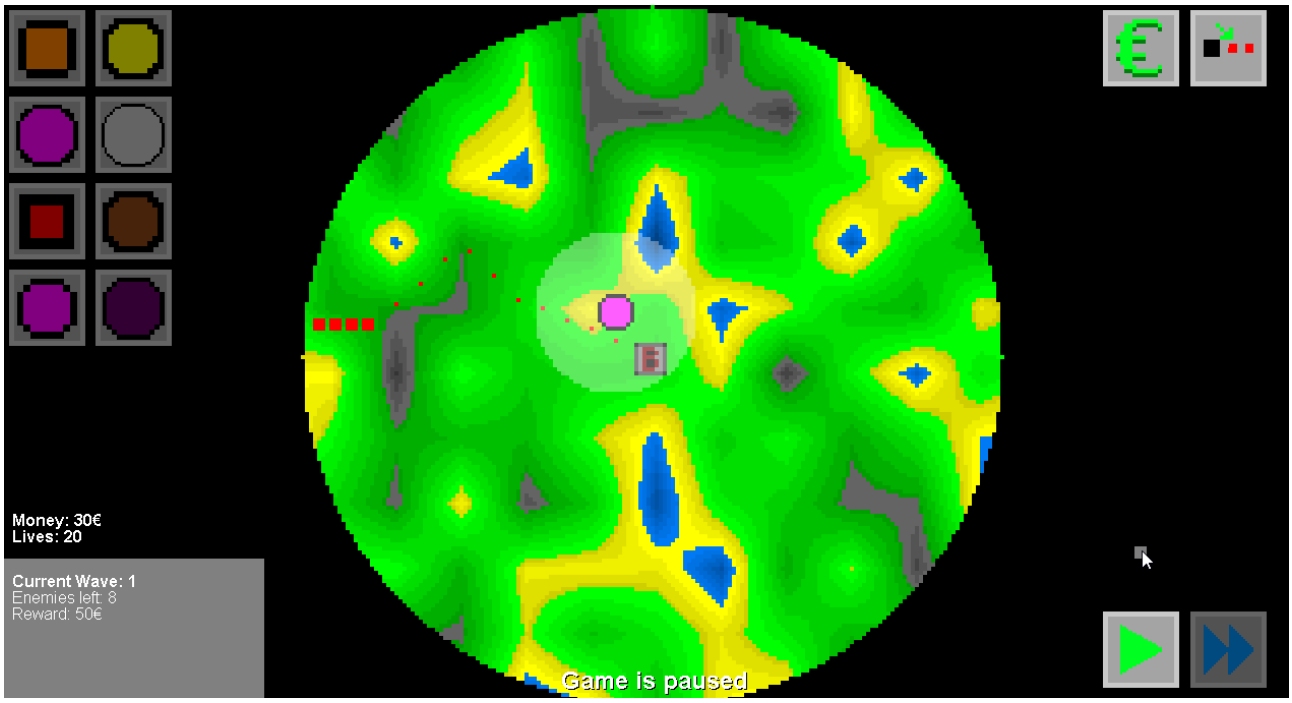
Most tower defence games can generally be split into two categories: linear and dynamic. In linear tower defence games, the enemies' path is fixed and mostly linear (like Kingdom Rush) and the player actions don't have an impact on the pathfinding at all. I am making a dynamic tower defence game, where the game is located on a grid and you can place tower's anywhere, and the enemy can go anywhere (like Bubble Tanks TD). This is programmatically more challenging which makes it a much more interesting direction for a tower defence game. I will implement a pathfinding algorithm and I have accounted for a lot of the unexpected stuff a player can do in a dynamic environment (like blocking the only paths).

The game has lives, economy, towers, waves, and enemies. The towers, waves and enemies are generated from a file which the user can modify freely. They can add as many waves or towers as they see fit. The files are in the config folder. I have already setup these files to be balanced from the get-go if the user doesn't want to modify these things. I have also implemented my own UI system to get full control of the UI and get rid of most of the bloat that comes with swing UI elements.

I'm aiming for the maximum grade on this project so I have implemented features which can be considered "difficult" in my opinion. I have also paid a lot of attention on extensibility and testing.

User Interface

The game opens immediately after launching the game. After that you are ready to play. First you must place you base which you must protect with towers. After placing the base, you get access the all the features from UI to the keyboard inputs. You can move to map using WASD or arrow keys. On the left you can select a tower to place if you can afford it. On the bottom left you can see your money and lives. There's also an information box which displays relevant information on when you hover on buttons. On the bottom right are two buttons: The "time" button and the "start wave" button. You can pause the game using the "time" button or pressing space. The "start wave" button starts the next wave and can only be clicked if you have cleared the current wave (Pressing "Q" also starts the next wave). You can also select a tower by clicking on it on the map to see its range and do some actions with it. Whenever you have selected a tower, two buttons appear on the top right: "Sell" button and "Targeting mode" button. You can sell the selected tower for half the price with the "Sell" button. The "Targeting mode" button changes the towers targeting mode. See more info in the information box when you hover over the "Targeting mode" button.



Picture of the User Interface

Program Structure (UML next page)

These are the most important classes in the project:

class Game: The whole program revolves around this class. When other classes need access to another class, they usually must go through the Game class first. This class has references to the most integral class instances and allows for communication between them. It also initializes almost everything and runs the game logic. The main loop is in the update() method.

class Parser: Responsible for parsing the text files and transforming them to class instances.

object InputHandler: I made this a singleton because there should be only one instance of this object and the easy access was convenient. This handles all the things related to inputs. It handles the mouse clicks, the keyboard presses and listening to UI events.

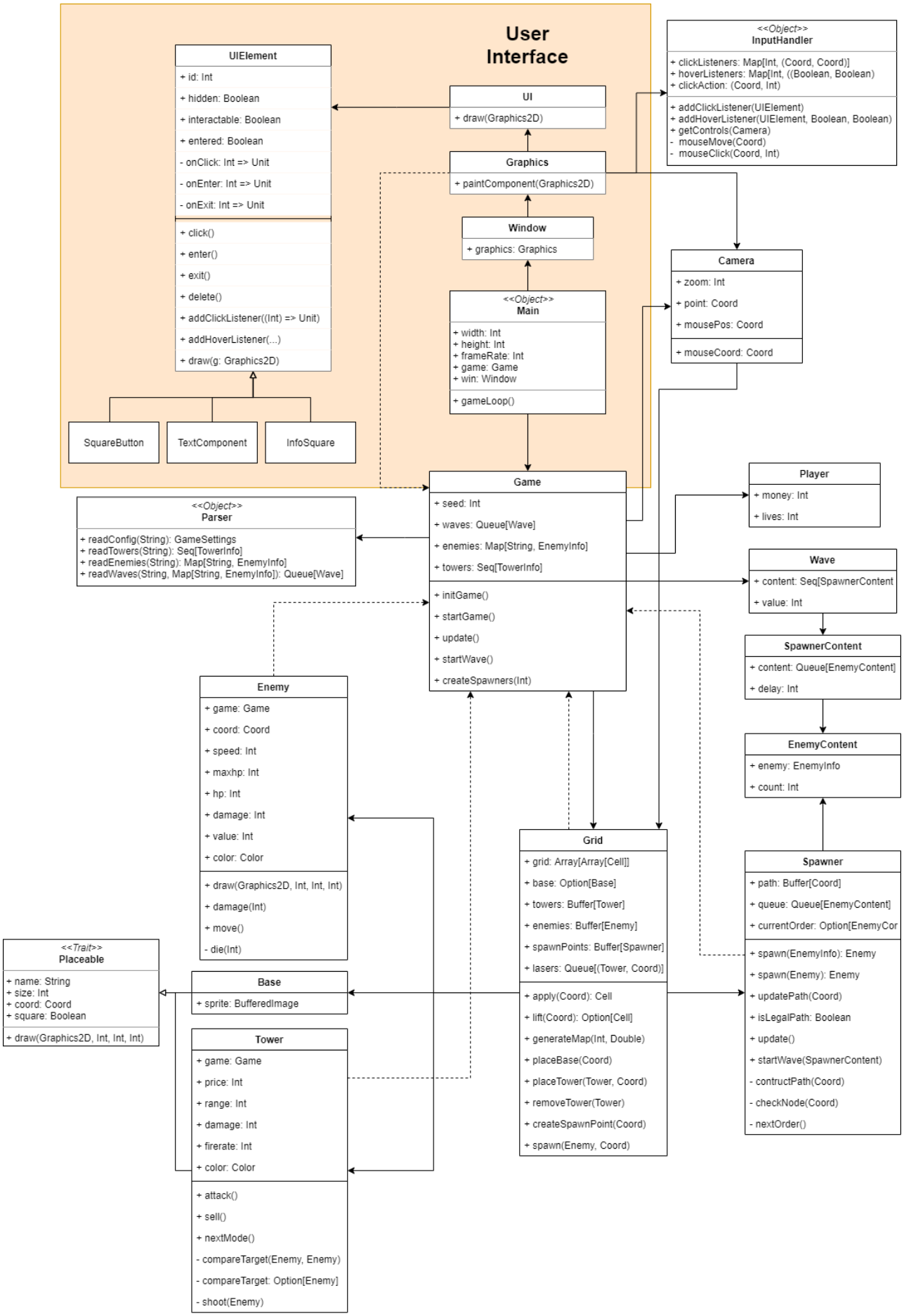
class Graphics: This handle all drawing all the elements ranging from placeable objects to the map itself. The drawing pipeline is this:

Map -> Path -> Enemies -> Base -> Towers -> Lasers -> Fog of War -> Selection highlight -> Mouse highlight

object UI: Singleton because only one instance is needed and being able to add UI elements from anywhere was convenient. Holds all the information about the UI and its elements and is responsible for drawing the UI

class Grid: This holds all the information about objects that are on the map. It is also responsible for generating the map using Perlin noise.

class Tower/Enemy/Spawner: Represents an element that can be placed on the map. Spawner differs from the other two by not having a draw function. The other two can be drawn and thus seen on the map. Spawners spawn enemies. Enemies move and attack the base. Towers shoot enemies in range.


User Interface
UIElement
+ id: Int
+ hidden: Boolean
+ interactable: Boolean
+ entered: Boolean
- onClick: Int => Unit
- onEnter: Int => Unit
- onExit: Int => Unit
+ click()
+ enter()
+ exit()
+ delete()
+ addClickListener((Int) => Unit)
+ addHoverListener(...)
+ draw(g: Graphics2D)
SquareButton
TextComponent
InfoSquare
UI
+ draw(Graphics2D)
Graphics
+ paintComponent(Graphics2D)
Window
+ graphics: Graphics
<<Object>>
Main
+ width: Int
+ height: Int
+ frameRate: Int
+ game: Game
+ win: Window
+ gameLoop()
<<Object>>
InputHandler
+ clickListeners: Map[Int, (Coord, Coord)]
+ hoverListeners: Map[Int, ((Boolean, Boolean)
+ clickAction: (Coord, Int)
+ addClickListener(UIElement)
+ addHoverListener(UIElement, Boolean, Boolean)
+ getControls(Camera)
- mouseMove(Coord)
- mouseClick(Coord, Int)
Camera
+ zoom: Int
+ point: Coord
+ mousePos: Coord
+ mouseCoord: Coord
Game
+ seed: Int
+ waves: Queue[Wave]
+ enemies: Map[String, EnemyInfo]
+ towers: Seq[TowerInfo]
+ initGame()
+ startGame()
+ update()
+ startWave()
+ createSpawners(Int)
<<Object>>
Parser
+ readConfig(String): GameSettings
+ readTowers(String): Seq[TowerInfo]
+ readEnemies(String): Map[String, EnemyInfo]
+ readWaves(String, Map[String, EnemyInfo]): Queue[Wave]
Player
+ money: Int
+ lives: Int
Wave
+ content: Seq[SpawnerContent
+ value: Int
SpawnerContent
+ content: Queue[EnemyContent]
+ delay: Int
EnemyContent
+ enemy: EnemyInfo
+ count: Int
Enemy
+ game: Game
+ coord: Coord
+ speed: Int
+ maxhp: Int
+ hp: Int
+ damage: Int
+ value: Int
+ color: Color
+ draw(Graphics2D, Int, Int, Int)
+ damage(Int)
+ move()
- die(Int)
Grid
+ grid: Array[Array[Cell]]
+ base: Option[Base]
+ towers: Buffer[Tower]
+ enemies: Buffer[Enemy]
+ spawnPoints: Buffer[Spawner]
+ lasers: Queue[(Tower, Coord)]
+ apply(Coord): Cell
+ lift(Coord): Option[Cell]
+ generateMap(Int, Double)
+ placeBase(Coord)
+ placeTower(Tower, Coord)
+ removeTower(Tower)
+ createSpawnPoint(Coord)
+ spawn(Enemy, Coord)
Spawner
+ path: Buffer[Coord]
+ queue: Queue[EnemyContent]
+ currentOrder: Option[EnemyCor
+ spawn(EnemyInfo): Enemy
+ spawn(Enemy): Enemy
+ updatePath(Coord)
+ isLegalPath: Boolean
+ update()
+ startWave(SpawnerContent)
- contructPath(Coord)
- checkNode(Coord)
- nextOrder()
<<Trait>>
Placeable
+ name: String
+ size: Int
+ coord: Coord
+ square: Boolean
+ draw(Graphics2D, Int, Int, Int)
Base
+ sprite: BufferedImage
Tower
+ game: Game
+ price: Int
+ range: Int
+ damage: Int
+ firerate: Int
+ color: Color
+ attack()
+ sell()
+ nextMode()
- compareTarget(Enemy, Enemy)
- compareTarget: Option[Enemy]
- shoot(Enemy)

Algorithms

For the monster pathfinding I used the A* algorithm which is very efficient and easy to implement in a grid-based map. A* basically has two weights on each cell which are the cost start node and the cost to end node. Using these we can efficiently find a path while calculating a relatively small number of cells. The towers have a finite range and they must calculate which monsters to shoot. By default, they should shoot the monster closest to the base. You are also able to choose different targeting modes. We can get the monster by calculating its distance to the goal node and checking if it is in the radius of the circle (by calculating the distance between the monster and the tower). Level generation uses the Perlin noise algorithm to generate the map. This way we will have a randomly generated map that also looks coherent and sensible. Perlin noise is a gradient noise algorithm which smoothly interpolates from point a to point b. This will allow for smooth transitions in our map generation.

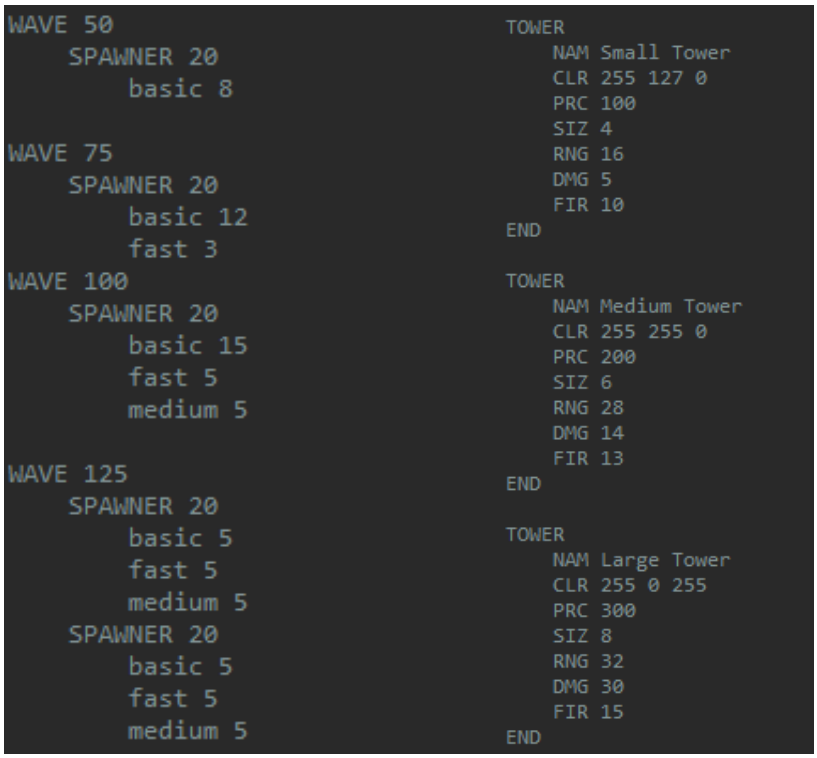
picture of Perlin noise output

Data Structures

We need collections for Waves and the Grid. The grid is a 2-dimensional array which holds placeable objects like monsters or turrets. Waves hold a Wave object which includes information about that wave like the list of monsters. I have been using both mutable and immutable data types on a case by case basis. Collections used were provided by the Scala standard library, there was no need to program my own data structures. Example of collections used: mutable Queue, mutable Map, immutable Seq, immutable Vector.

File and Internet Access

This game doesn't use the internet at all but is uses text files. These files are in the config folder. The towers, waves and enemies are parsed from a text file which is human readable. This allows for extensibility and variety. The user can modify and add as many towers as they want. The files use an easily understood custom syntax. The text files include an indepth guide to the syntax.



```
WAVE 50
    SPAWNER 20
        basic 8

WAVE 75
    SPAWNER 20
        basic 12
        fast 3
WAVE 100
    SPAWNER 20
        basic 15
        fast 5
        medium 5

WAVE 125
    SPAWNER 20
        basic 5
        fast 5
        medium 5
    SPAWNER 20
        basic 5
        fast 5
        medium 5
```

```
TOWER
    NAM Small Tower
    CLR 255 127 0
    PRC 100
    SIZ 4
    RNG 16
    DMG 5
    FIR 10
END

TOWER
    NAM Medium Tower
    CLR 255 255 0
    PRC 200
    SIZ 6
    RNG 28
    DMG 14
    FIR 13
END

TOWER
    NAM Large Tower
    CLR 255 0 255
    PRC 300
    SIZ 8
    RNG 32
    DMG 30
    FIR 15
END
```

Example of Waves.txt and Towers.txt

Testing

This program is tested with unit tests using the ScalaTest library. Writing extensive tests is the main way to maintain a working program and a readable code base. Since this a GUI game, it has been QA tested a lot, which has caught bugs which the unit tests did not catch. Gameplay testing was also an important part of testing since we need to gauge the game's balance and enjoyment.

Since my game is grid based, simulating a game environment in unit tests should be easy. I have separated the game components from GUI components which means that we can simulate a game environment in isolation without the GUI components. With the GUI it is hard to test some aspects like whether the UI displays correctly or if the animations work correctly or if events fire correctly. The GUI has instead been tested with extensive QA testing. In practice this means pushing the GUI to the edge by doing unexpected actions and so on. The classes that can be tested in isolation are tested with ScalaTest unit tests. You can run these tests by going to the project folder and running "sbt test" in the terminal if you have sbt installed.

Known bugs and missing features

I have been rigorous with bug fixing because having major bugs can ruin the enjoyment. I can only think of some minor UI bugs or HIGHLY improbable situations with the random generations. For example, the if you hover too fast from button to button the OnEnterEvent triggers before the OnExitEvent trigger which means that the information box can display the wrong information, but this happens rarely, and it is very minor. The highly improbable situations I'm talking about are for example: The map generation creating a perfect enclave and placing the base there which means that there is no path there for the enemies. This is theoretically possible but nigh impossible, this has never even close to happening for me.

The only missing feature that I said I would implement in the general plan was upgrading towers. The idea was to have an upgrade tree for each tower. I had to cut this feature due to design reasons and not technical reasons. Adding the upgrade tree would make one of this project's main features, the parsing towers from file, feature needlessly complicated for the player. Therefore, I decided to cut that feature out

3 best sides and 3 weaknesses

Best sides:

- The extensibility of the program. You can add as many different towers, waves, and enemies by modifying the config files. The syntax is easy to learn, and the documentation is clear. Everyone can tailor the gameplay to their liking.
- The procedural generation. Every playthrough is completely different. You must design on whole new strategy for every map and skill comes from adapting to the new environment.
- The aesthetics. I think the aesthetics are good even though they are mostly procedurally generated. The only things that use sprites are the base and the icons on the buttons. Everything else is drawn in code. This allowed for procedural tower and enemy creation. You can give towers and enemies any size and colour you want, and it'll work. The map is also very nice looking, and it feels natural.

Weaknesses

- Performance and scalability. Even though I tried my best at optimizing different aspects of the code and reducing calls to expensive functions and algorithms, there are some places where the performance is not ideal. When you have hundreds of towers and enemies and the path is very intricate, the performance will suffer somewhat. But even after all that I think I did an okay job with the knowledge I had, and the program doesn't suffer much from bad performance my PC. I could have, for example, grouped the tiles so the A* pathfinding algorithm doesn't traverse as many nodes as it does now.
- Overreliance on Game class. Most classes have maybe a bit too many dependencies to the Game class. But the project was small enough that it was still very readable even though I probably could have made some classes a bit more isolated.
- Using my UI class can be laborious. I really like the UI system I implemented for this project, but it can take a few too many lines of code to implement something. I could fix this by adding more helper methods and streamlining some aspects.

Deviations from the plan, realized process and schedule

I deviated very little from the plan. This is almost identical to what I was imagining at the start of the project. The realized process was also like what I had in mind, only difference is the fact that I started adding tests pretty late into the development.

The schedule is the biggest thing that deviated from the original plan. I was too optimistic about the first 2 weeks in my original plan. It said I should have a working prototype by week 2. In reality I got a working prototype in week 6. I also didn't work on the project at all in weeks 2-4. But I got everything done by week 7 so in the end it wasn't that bad.

Final evaluation

I did a great job with this project. I have a very hard time finding faults in this project since I fixed them whenever I found them. I went further than a tower defence usually goes. The extensibility is also great, and you can tinker with many aspects of this game in the config files. I also think that I got the procedural generation right. Usually when people implement procedurally generated gameplay, it feels cheap and needless. I believe that the procedurally generated aspects of my game enhance the over all experience. Other than the weaknesses I listed previously; I can't really find any other shortcomings in this project but maybe I'm biased.

Only thing I would do differently is implementing tests a little bit earlier.

References

A* algorithm
https://en.wikipedia.org/wiki/A*_search_algorithm

Perlin noise
https://en.wikipedia.org/wiki/Perlin_noise

Scala Documentation
https://www.scala-lang.org/api/2.13.1/

ScalaTest Documentation
http://www.scalatest.org/user_guide

Appendix

The source code and a README.txt file which will include instructions on running the program.