1. **Overview**: what the software does, what it doesn’t do? (this can be taken/updated from the project plan)

When a game is launched, the program reads a map.txt-file and an enemy wave text file (also referred to as a difficulty file) to generate a game instance.

A map file is a 20x9 .txt-file with the following characters:

- Indicates a free tile, towers can be placed here

- Indicates a path tile, enemies run along these, towers cannot be placed on path tiles

S - Indicates the starting tile of the path. Has same characteristics as regular path tiles

E - Indicates the ending tile of the path. Has same characteristics as regular path tiles

The program reads the file, generates TileGraphicsItems for each character read and adds them to the scene. If the path has been programmed faultily, the program will notify the user that the map file is faulty. An example of a faulty path:

```
####################
####################
###----##-----######
S---#########------E
####################
####################
####################
####################
####################
```

However, if the dimensions of the map file are inconsistent, the program will crash. This means that the program expects the user to give a map file that has the correct dimensions. It also cannot include other characters than the four listed above.

An example of a map file that has inconsistent dimensions:

```
#############
##################
###----##-----######
S---#########------E
####################
####################
####################
################
####################
```

As long as the dimensions are consistent, smaller maps such as 5x10, for example can be generated but the game is designed to be played on 20x9 sized maps.

After clicking “Start Game” the program checks whether the given map can be used to generate a game as well as checking whether the corresponding difficulty file exists. Difficulty files are text files that include the waves for their specific game instance. Waves are separated from each other by the ‘*’-character. The difficulty files contain the following characters:

R - Indicating one enemy (Red enemy type)

B - Indicating one enemy(Blue enemy type)

G - Indicating one enemy (Green enemy type)

W - Indicating one enemy (White enemy type)

P - Indicating one enemy (Boss enemy type)

* - Indicating the beginning of a new wave

An example of a difficulty file:

```
RBRBR*RRRRR*BBR*RBRBRB*RGBRGB*RWRWRW*BWBWBW*RGBWRGBWGBRW*RBGWWWGGBBBWWGGB*WRBGBWBRGWPBBWWG
```

The program reads every wave and appends each one to a two dimensional vector (private member of the Game-object). The vector is used to spawn the waves of enemies; every element of the vector represents one wave. The elements are strings, which are read by a for-loop. For each character read, a corresponding enemy is created and spawned.

When towers are placed on the map, a QTimer object is created simultaneously as well as a range_ object (QGraphicsEllipseItem). At specific time intervals (determined by each tower’s fire rate) they will check whether an Enemy collides with the range_ object. If so, the tower fires a projectile (Bullet-class, QGraphicsPixmapItem) at the current location of the enemy.

Killed enemies grant a certain amount of money, are removed from the scene and their memory freed. Enemies that reach the end of the path do not grant money, but instead lives are also deducted from the player.

Quick info about towers:

Basic - Average attack speed, 1 damage per hit, doesn't pierce armour, costs 400 gold

Quicky - Fast attack speed, 1 damage per hit, doesn't pierce armour, costs 1300 gold

Piercer - Slow attack speed, 2 damage per hit, pierces armour, costs 600 gold

Sniper - Very slow attack speed, 3 damage per hit, pierces armour, vast range, costs 1000 gold.

Quick info about enemies:

Red - 1 hp, grants 40 gold when killed, no armour

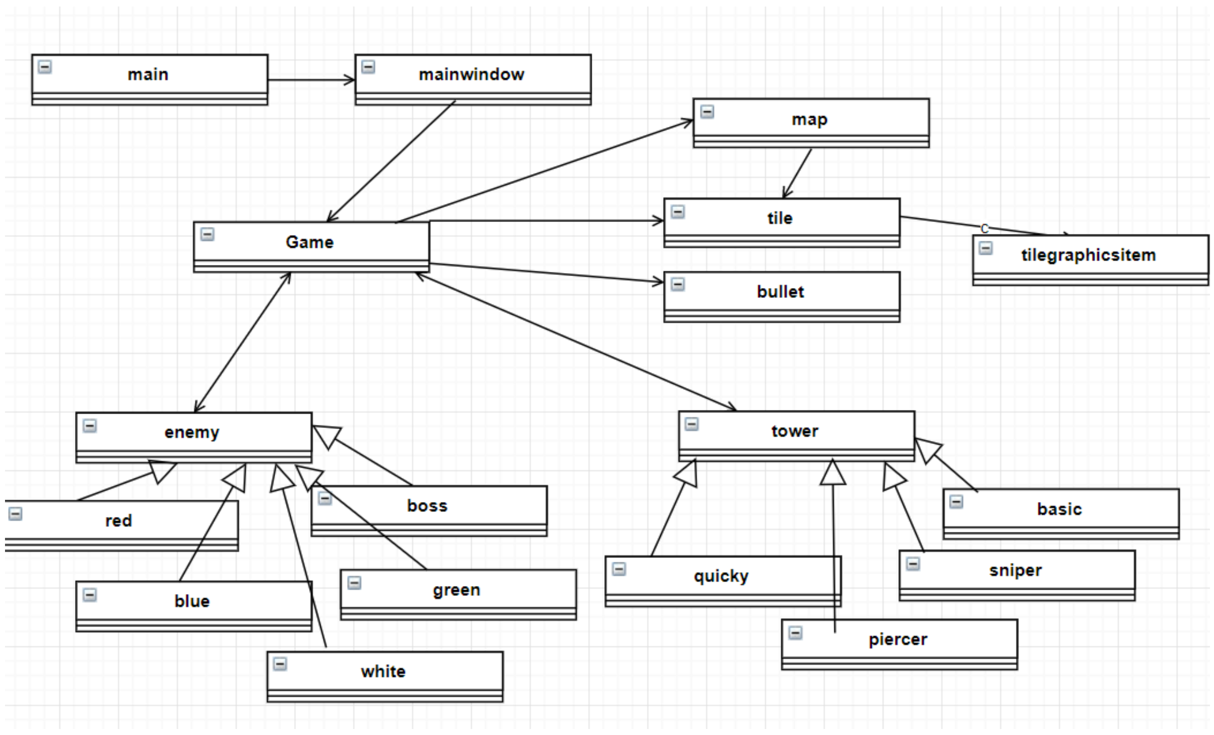
Blue - 2 hp, grants 60 gold when killed, no armour

Green - 3 hp, grants 70 gold when killed, armoured (takes damage only from piercer and sniper type towers)

White - 4 hp, grants 80 gold when killed, not armoured

Boss - 10 hp, grants 200 gold when killed, not armoured

2. **Software structure**: overall architecture, class relationships diagrams, interfaces to external libraries



No external libraries are needed. The software solely uses Qt.

3. **Instructions**

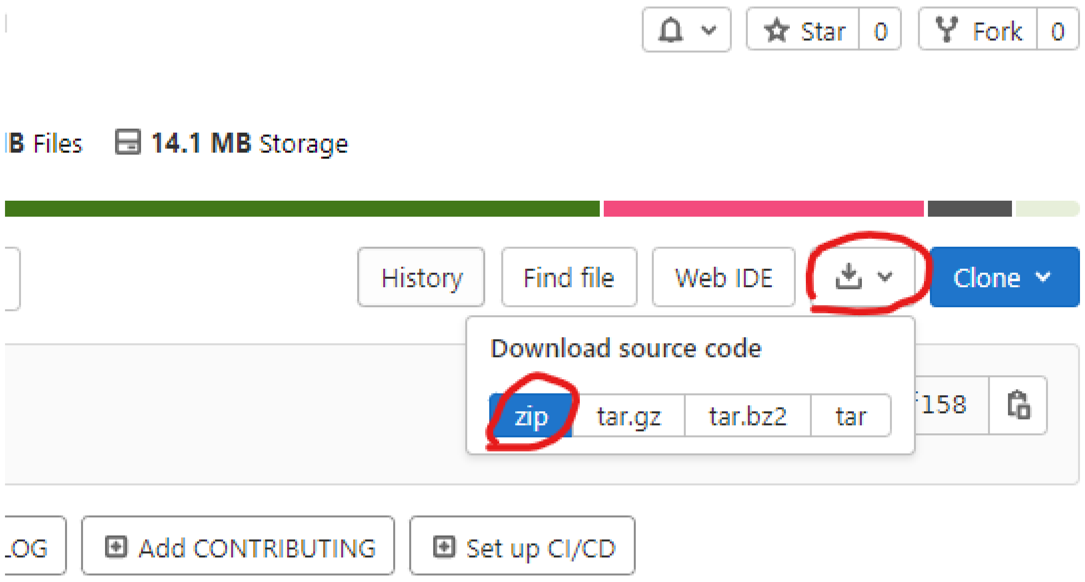
In order to run this program you will need the following programs installed on your computer:

For Windows: MinGW, CMake, Qt Creator (Open source).

Note that if you haven't installed Qt Creator before, you will have to create a free account to use the IDE

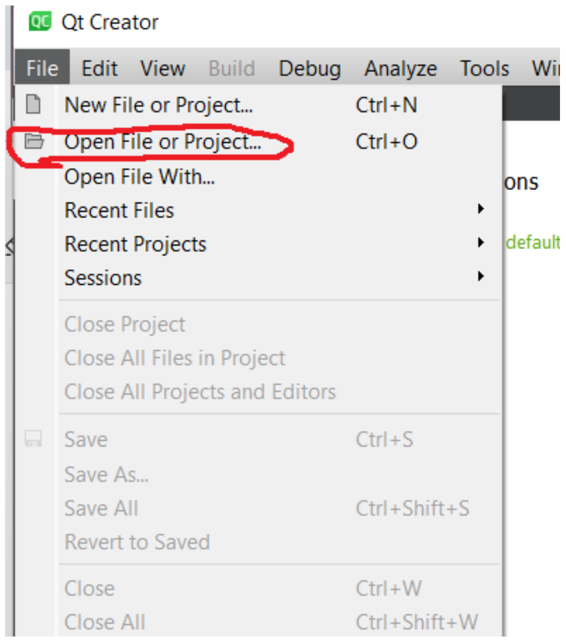
For Mac: Qt Creator (Open source)

From the Git repository, click on the download button in the top right corner of the browser and choose the .zip option.

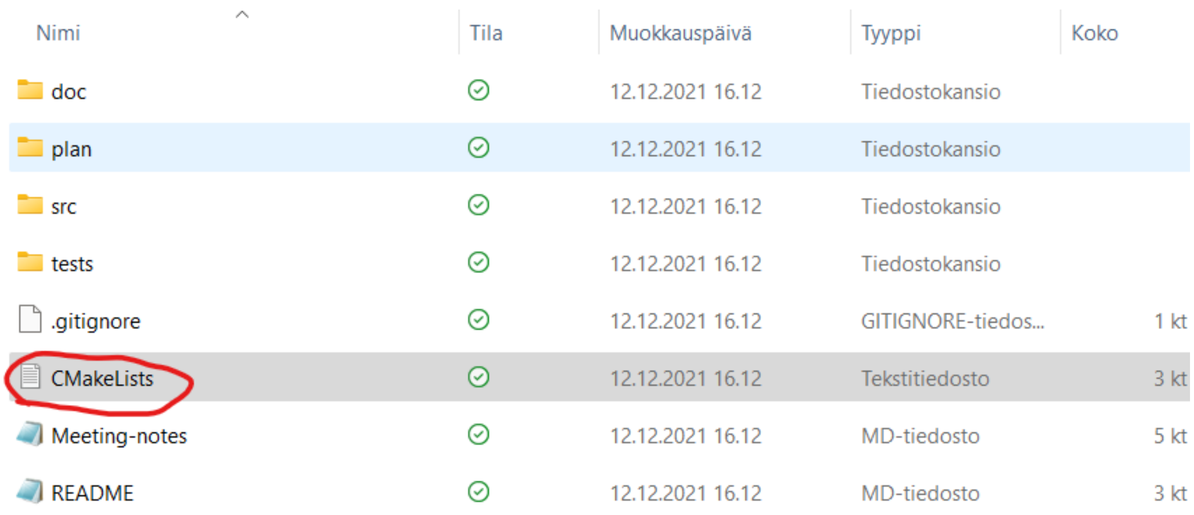


Unzip the project to a folder of your choosing (your desktop, for example). The project folder should be called “tower-defense-8-master”.

Next, launch Qt Creator and navigate to the top left corner, click File -> Open File or Project

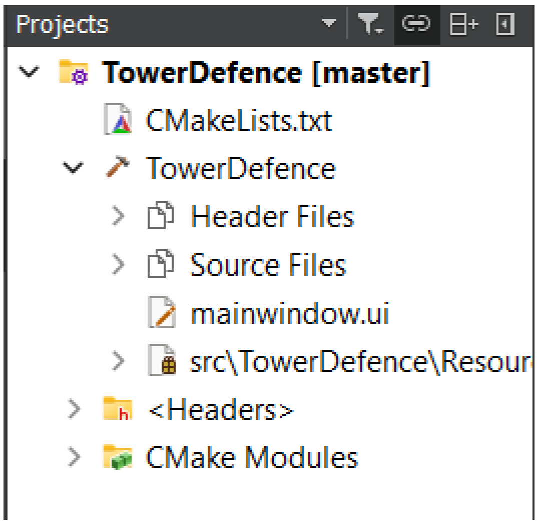


Navigate to the project folder that you just downloaded, open it and select the file named “CMakeLists.txt” located in the root of the folder.



Nimi	Tila	Muokkauspäivä	Tyyppi	Koko
doc	✓	12.12.2021 16.12	Tiedostokansio	
plan	✓	12.12.2021 16.12	Tiedostokansio	
src	✓	12.12.2021 16.12	Tiedostokansio	
tests	✓	12.12.2021 16.12	Tiedostokansio	
.gitignore	✓	12.12.2021 16.12	GITIGNORE-tiedos...	1 kt
CMakeLists	✓	12.12.2021 16.12	Tekstitiedosto	3 kt
Meeting-notes	✓	12.12.2021 16.12	MD-tiedosto	5 kt
README	✓	12.12.2021 16.12	MD-tiedosto	3 kt

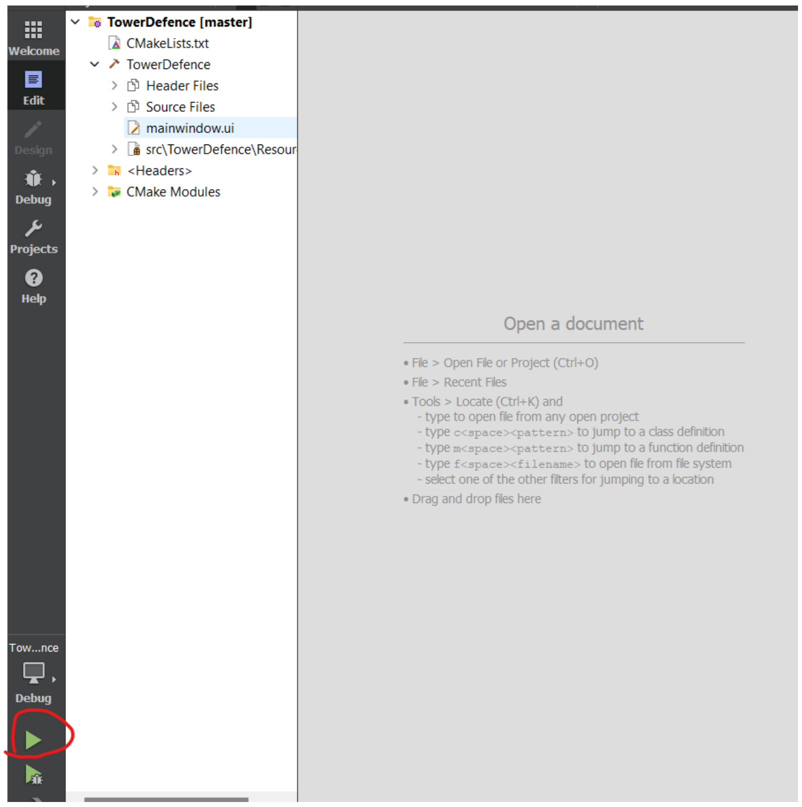
After configuring the project, you should be able to browse through the source code on the left hand side of the IDE.



```
Projects
TowerDefence [master]
    CMakeLists.txt
    TowerDefence
        Header Files
        Source Files
        mainwindow.ui
        src\TowerDefence\Resour
    <Headers>
    CMake Modules
```

4. **How to compile the program**

To compile and run the program, click the Green play button in the bottom left corner of the IDE.

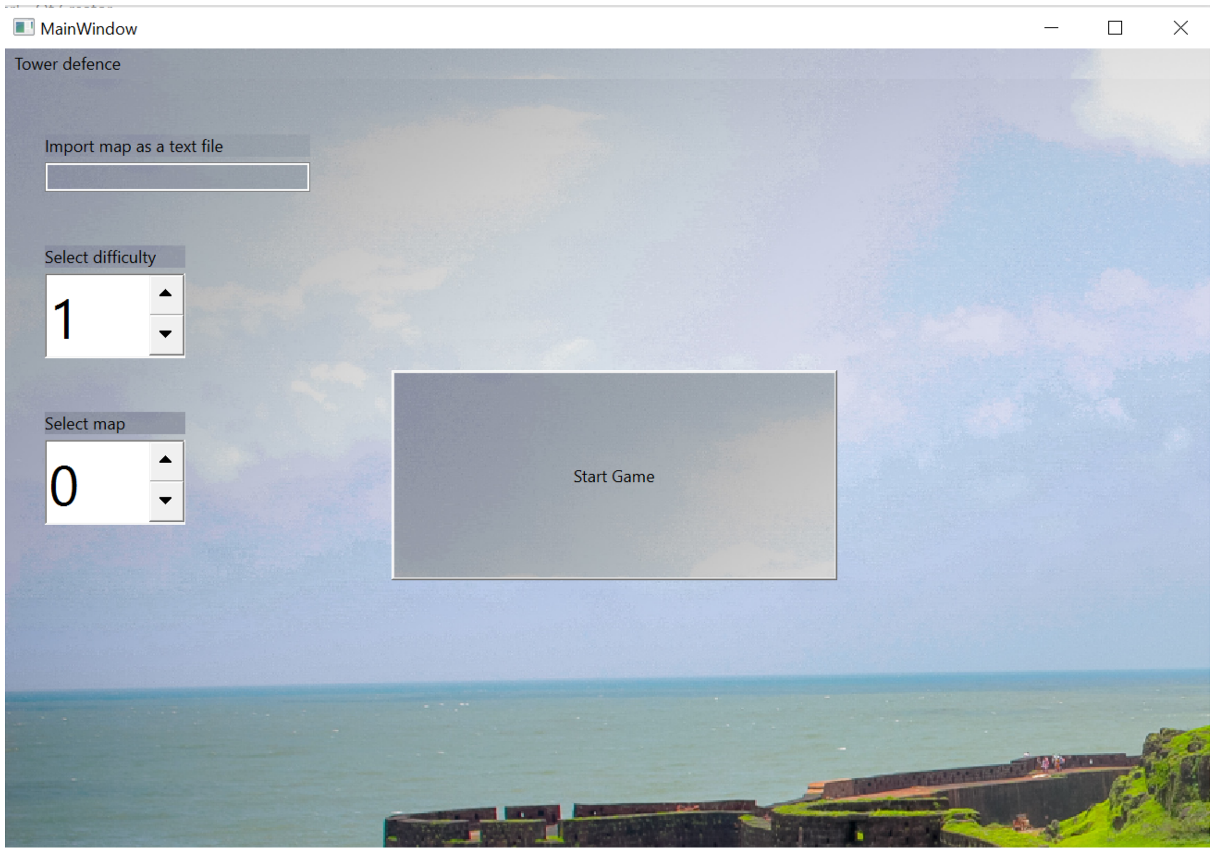


Open a document

- File > Open File or Project (Ctrl+O)
- File > Recent Files
- Tools > Locate (Ctrl+K) and
  - type to open file from any open project
  - type `c<space><pattern>` to jump to a class definition
  - type `m<space><pattern>` to jump to a function definition
  - type `f<space><filename>` to open file from file system
  - select one of the other filters for jumping to a location
- Drag and drop files here

5. **How to use the software**

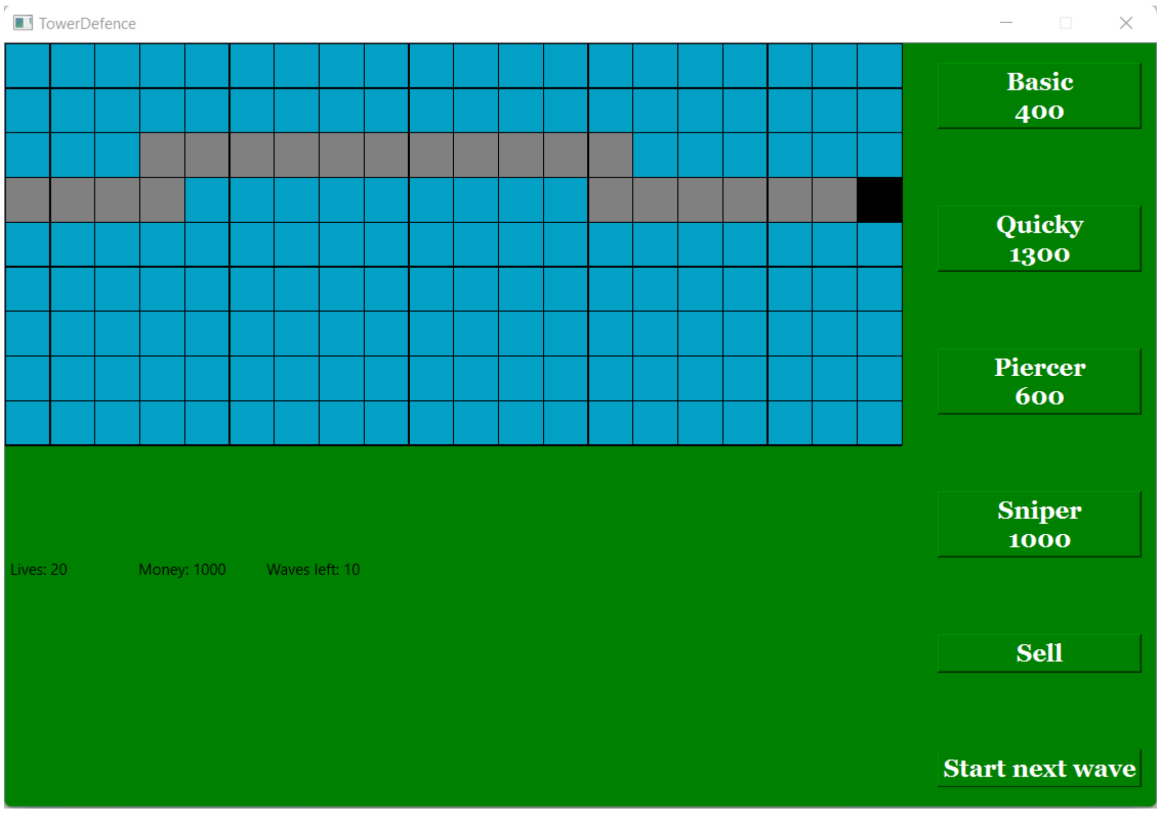
When running the project, a main window opens up. Interacting with the window allows the user to set a difficulty level for the game. There are five different difficulty levels (1 being the easiest and 5 being the hardest). The user also is asked to input the filename (without .txt postfix) of a map file or select the map from a spinbox. After the user clicks the push button “Start Game”, the program reads the file and renders a map based on the contents of the map file. Main window after running the program:



The text input field is used to open maps you have created by yourself. If you have created a playable map-file, added it to the folder ./src/TowerDefence and added it to the Resources.qrc file list, you can open it by typing the filename (without ".txt" postfix) in the text input field of the main window. Make sure that the select map count is 0 when importing your own map file.

To play a map we have created for you, set "Select map"-counter to 1, 2 or 3 and leave the text input field empty. Difficulties range from 1 to 5 with 5 being the hardest difficulty setting. Click "Start Game" to begin.

If a game can be created, a game object is initialised and a new window pops up:



Clicking “Start next wave” starts the next wave. A wave cannot be started before the previous wave is completed. Towers can be placed on the map by clicking the wanted tower button on the right side of the window and then clicking a free tile on the map. Clicking a placed tower will show it’s effective range and re-clicking will hide the range. Selling a tower is done by clicking the button “Sell” and then clicking a placed tower. Selling a tower returns 80% of its value to the user. Towers attack enemies automatically when they enter their effective range.

When an enemy reaches the black tile at the end of the path, lives are reduced based on the enemy’s remaining hp. You will gain money for each enemy killed, but won’t get anything for an enemy that reaches the end of the path. Losing all lives means you lose the game! Clearing all waves means you win the game!

6. **Testing**: how the different modules in software were tested, description of the methods and outcomes.

In order to test the program we used a variety of simplified test files along with well-placed print commands (qInfo() in Qt syntax). To test little parts of the program such as individual for loops and tower placement, we used the method qInfo() to print the wanted information into the output stream.

To test bigger functionalities such as map or wave reading methods we used simplified test files. For example, when testing the enemy wave spawner in the Game-object, we used a file that only contained one enemy wave, consisting of three Red enemies instead of a full 10-wave containing file.

7, 8 & 9. Work log: a detailed description of the division of work and everyone's responsibilities.

Week 1:

All of us tried to install the SFML-libraries in order to get started with the project. After hours of trying, we decided to set up a meeting with our project assistant so he can help us out. Even with the help of our assistant we could not make it work so we decided to write the project using Qt. Each project member spent approximately 10 hours this week.

Week 2:

We installed Qt Creator and the needed programs to run the project. We created classes, main and a graphical interface. Kim will start writing enemy and tower classes. Kim and Matias created Map-class and wrote a function that reads a .txt-file from which it then saves to a 2D vector (private variable) that represents the game map. The function also saves the coordinates of the enemies' path into a private member of the class Map. Sanni familiarized herself with the graphical interface and started creating a main window for the game. Meeri wrote the Tile-class which represents game tiles within the map. Each project member spent approximately 20 hours this week.

Week 3:

Learned more about Qt. Finished enemy classes (will add more methods later on), finished some graphicsItem-classes for map tiles. Running the project now opens up a menu. Kim wrote tileGraphicsItems and some enemy classes. Created a resource.qrc file to store resources. Matias wrote enemy classes and started working or tower classes. Meeri started working on projectile classes. Sanni created a start game button for the manu and a text field to import a map from. She also imported a picture for the menu. Each project member spent approximately 15 hours this week.

Week 4:

Sanni has begun working on a wavespawner in Game.cpp and methods for enemy class. Kim and Matias created pushButtons for buying towers and linked them to a handleClick-function. Meeri made it so towers can be bought and sold (and they appear on the map).Meeri and Kim created a range for the tower and made it shoot. Each project member spent approximately 15 hours this week.

Week 5:

Meeri made the enemies move along the path. We all fixed problems in the game, for example selling enemies and the game terminating when the last enemy was shot. We tested and fixed different scenarios where the program crashed. Each project member spent approximately 15 hours this week.

Week 6:

Added five different difficulty levels and more maps. Added armoured enemy type. Finalized the git repository, moved CMakeLists to the root folder. Wrote the readme-files and project documentation as well as the source code documentation. Each project member spent approximately 15 hours this week.