Snowmen VS Rabbits – Project Documentation

Roosa Ahlroos, Karl Rass, Lauri Karanko, Severi Koivumaa

Overview

"what the software does, what it doesn't do? (this can be taken/updated from the project plan)"

First, when opening the program a menu screen pops up. If you choose to play you have five different level options with increasing difficulties to choose from. Clicking one of these level buttons opens the game window and starts the game.

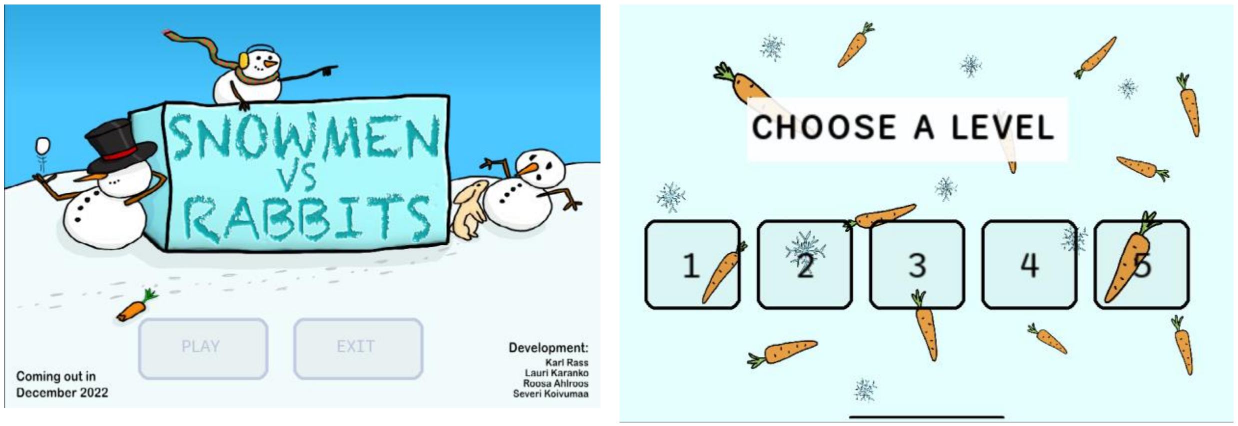

SNOWMEN VS RABBITS
PLAY
EXIT
Coming out in December 2022
Development: Karl Rass Lauri Karanko Roosa Ahlroos Severi Koivumaa
CHOOSE A LEVEL
1
2
3
4
5

Figure 1: Game menu and level select

In the beginning of the game, the player is given 200 $ of money to spend on towers and take on the first enemies. The player is awarded more money each time they kill an enemy. The amount of money received depends on the enemy defeated. One level goes on for ten rounds/waves and the player can add additional towers at any point during the game. When the player reaches the end of a round there is a small grace period, where he/she can take a breath and prepare for the next round. Successfully defeating a wave also grants the player a small monetary bonus.

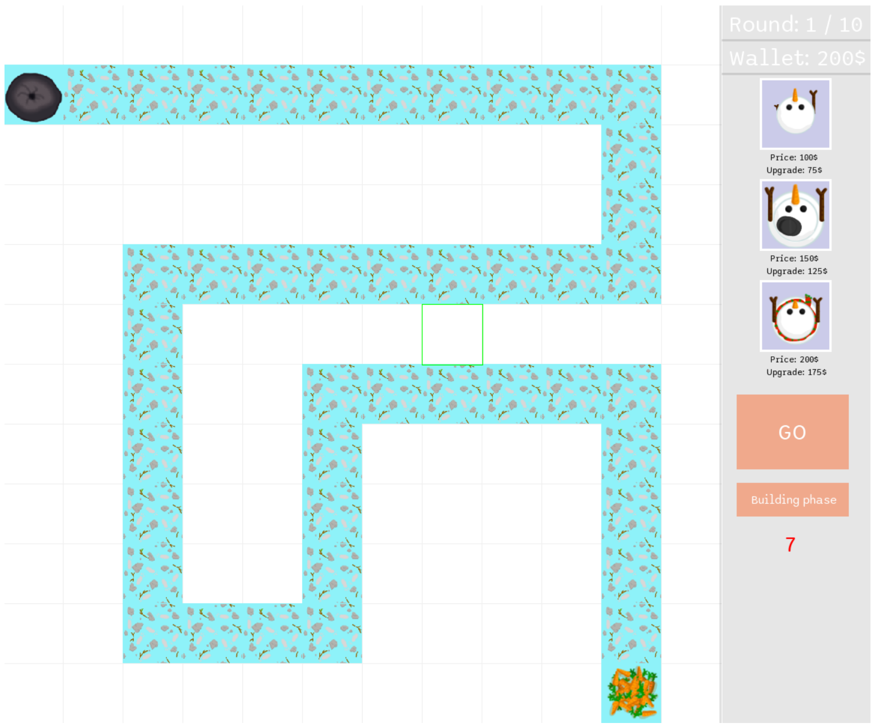


Figure 2: Game window before first wave

The different tower types (three in total) can be chosen from the sidebar on the right side of the map. The selected tower is placed on the map where the player clicked, if he/she has enough money. The towers can also be updated by clicking them, again, if one has enough money for it. The cost of tower placing and upgrading is told below each tower in the sidebar. The sidebar also tells the ongoing round number as well as the current amount of money.

The rules of the game are simple: Defeat all the enemies and you will win the game. Alternatively, if any enemy was to reach the end, the game will be lost.

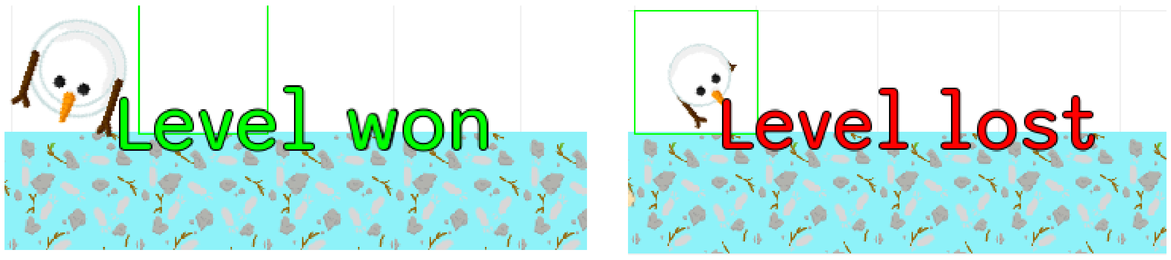


Figure 3: Level won and level lost messages

The additional features we implemented:

- Non-hardcoded maps
- Upgradable towers
- Tower placement that can be altered during an enemy wave
- Sound effects (music in our case)

The maps are loaded in from text files in the project folder, which are essentially CSV files. Note that the level files can be edited by changing the field values but that requires certain understanding of the file structure and therefore we do not think it qualifies as a “level editor”. More about that later.

Edited levels also support a version of branched paths. By adding multiple starting points for enemies and connecting paths at some point before the ending point it’s possible to have enemies coming from two different paths at the same time. However, this feature was not used in our level designs because there’s some visual glitches caused by multiple enemies not being able to occupy the same tile at once. It’s also possible to add multiple ending points and have two or more separate roads for enemies in the same level.

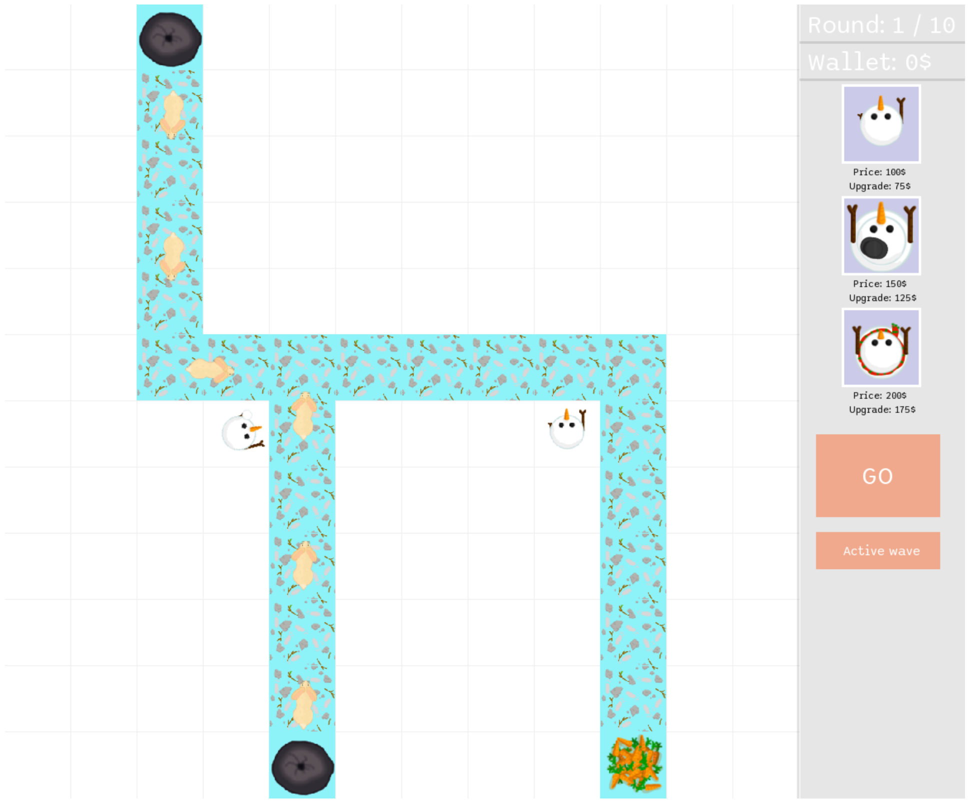


Figure 4: Edited level with multiple paths

Software Structure

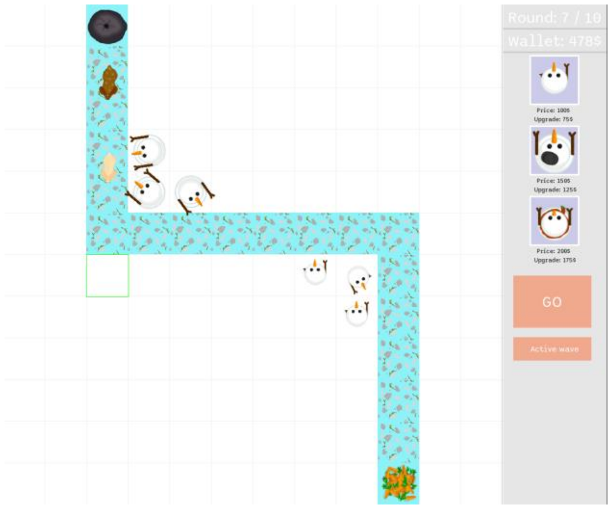
“overall architecture, class relationships diagrams, interfaces to external libraries”

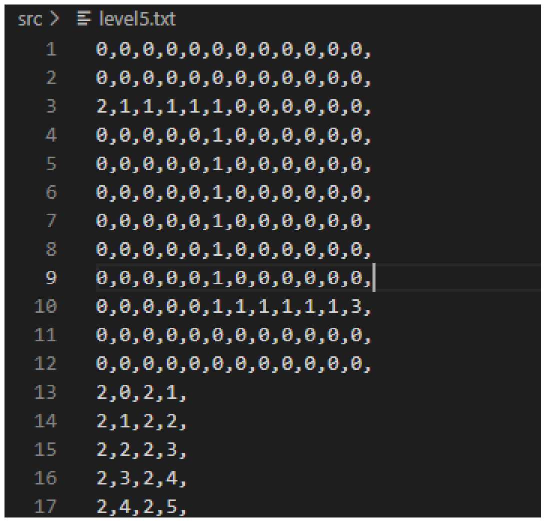
(**Bolded** words represent a class in the program)

Main.cpp runs the menu and from there the whole program. Navigation in the menu is made possible with buttons, which we have our own **button** class for. Using the buttons in the menu, you can start game loops with **game** objects that create a different **level** based on which button was pressed. The game class uses the SFML-library to create an interactable graphical user interface. Game class together with level class and its functions manage the overall game loop. Game class also updates the game’s state and renders a frame. **Enemies** and **towers** to be rendered are stored within vectors inside the game class.

Because of this, some functions that need to access these vectors directly are implemented in game class whereas others are level specific functions and are therefore implemented withing level class. An individual level is loaded from a separate txt file with necessary information in correct comma-separated values format.

Each level consists of **tiles**, which is an abstract class that has four different derived classes under it. These are **road**, **grass**, **entrypoint** and **exitpoint**. Roads represent the path for the enemies to walk on and the grass tiles are for the towers to be placed on. Entrypoint is the tile where the enemies appear from and the exitpoint is the tile where the game is lost if an enemy reaches it. A level forms a grid-like matrix consisting of tiles to keep track of their neighbors and positions. It also has a list of created enemies and towers with their locations. Both enemies and towers are abstract classes with three derived classes that have different basic statistics. The game class communicates with a separate **sidebar** class that is mainly used for creating new towers. The game uses an enemy-adding sequence that creates enemies, adds them to the level and makes them appear on the map. Neighbour values are used to create pathfinding system for enemies. Every tile where an enemy can appear has to have a neighbor value stored to its "next_" parameter pointing to the next tile from itself in order for the tile to know where to move an enemy next (generating a level without roads and entrypoints having neighboring values results in game crashing.). Game class also constantly tracks enemy positions and compares them to towers' positions and range attributes to deduce when a tower is going to shoot an enemy. When correct requirements are met a **snowball** is spawned on the tower and moved quickly towards the enemy. When a snowball hits an enemy, it takes damage. When an enemies hp value reaches 0 it is deleted.




src > level5.txt

```
0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,
2,1,1,1,1,1,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,0,0,1,1,1,1,1,1,3,
0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,
2,0,2,1,
2,1,2,2,
2,2,2,3,
2,3,2,4,
2,4,2,5,
```

Figure 5: Level layout and level files

On the above figure it is possible to see the layout of the level, with the entry point of the path in the top-right corner and the exit point in the bottom-left corner. The enemies can be seen on the blue path and the towers the grass (snowy grass) tiles. Additionally, on the right side of the figure is an example of the file corresponding to the same level. The levels are stored as a matrix of comma separated values with 0, 1, 2 and 3 representing different types of tiles. These files can in theory be edited in any text editor, provided that the format remains the same and the grid size is 12x12 tiles (a design choice and technical limitation). The only reason however why we do not consider this to be a "level editor" is that the road tiles need to be manually linked to each other, which can be seen done under the 12x12 matrix. The procedure is simple as every road tile simply needs to point to the next road tile in the matrix. However this requires understanding of the file system, is not intuitive and is therefore not considered a level editor.

On the below figure is our class diagram for all the used classes in the game:

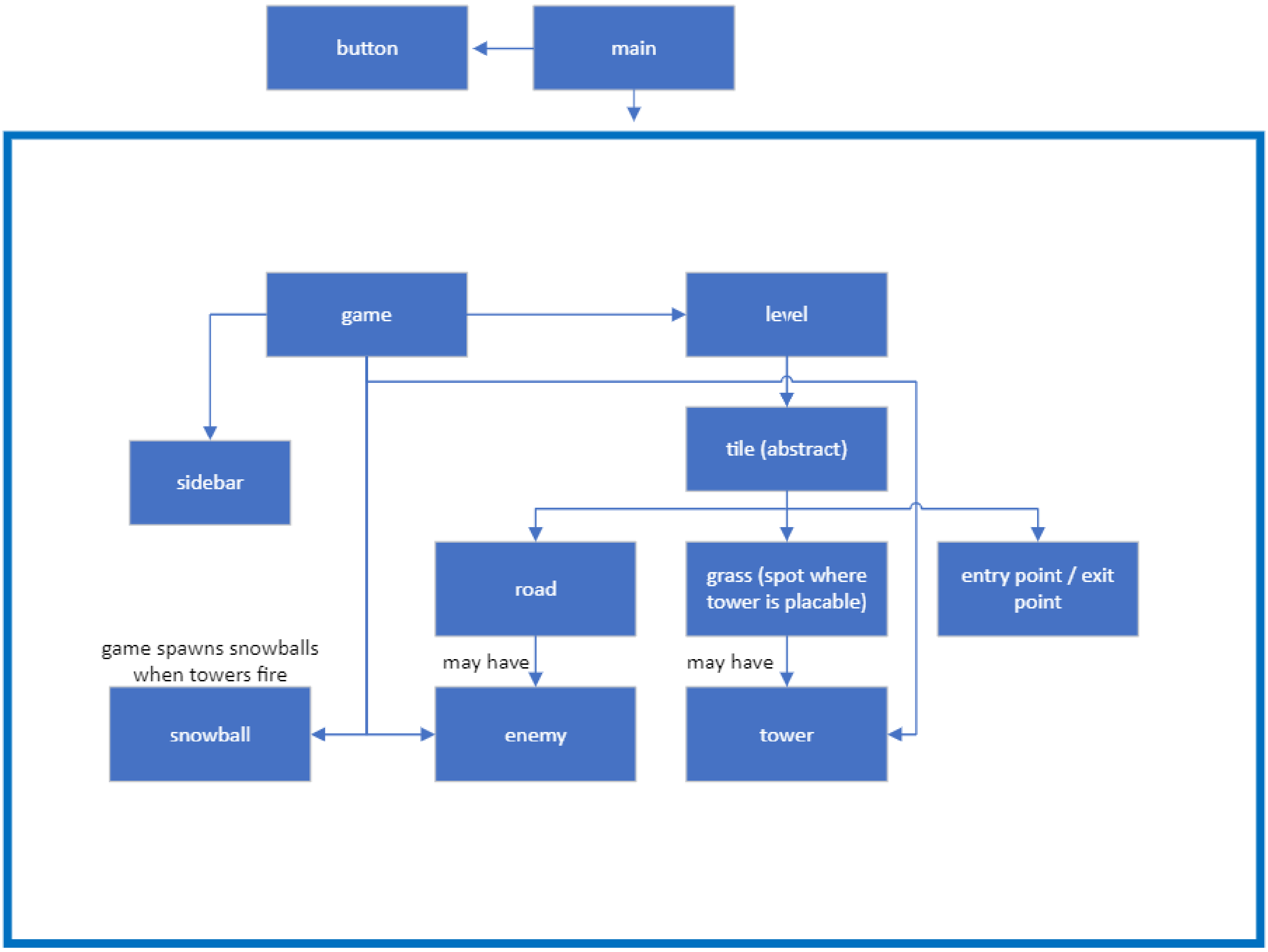


Figure 6: Class diagram

Instructions for building and using the software

"How to compile the program ('make' should be sufficient), as taken from the git repository. If external libraries are needed, describe the requirements here. How to use the software: a basic user guide"

Official way:

For CMake to work properly you must go to the CMakeLists.txt file and change the "link_directories" version according to your operating system. Then you can compile the source files and build/run the file "main.cpp" which has the main-function and the basic game loop.

The game uses SFML-library for graphical user interface, which is an universal dependency regardless of the operating system. However, this library is included in the project's lib-folder and should be handled by CMake, so one does not need to have SFML downloaded to their personal computer to run the code.

Other possible options:

Ubuntu (Linux)

On Ubuntu, the game can be compiled and launched by the following procedure:

1. Open the project folder in Terminal
2. Run the following commands:
    a. g++ -c src/*.cpp
    b. g++ *.o -o Game -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
3. Run the game with the following command
    a. ./Game

Mac

From terminal, navigate to the project folder and compile the .cpp files with the SFML libraries, for example:

/usr/bin/clang++ -lsfml-system -lsfml-audio -lsfml-network -lsfml-window -lsfml-graphics -L/libs/SFML-2.5.1-macos-clang/lib -fcolor-diagnostics -fansi-escape-codes -g src/entrypoint.cpp src/exitpoint.cpp src/game.cpp src/grass.cpp src/level.cpp src/main.cpp src/road.cpp src/tile.cpp src/tower.cpp src/tower_sniper.cpp src/snowball.cpp -o src/main -std=c++17

Then run the main with command: src/main

Windows

On windows follow these steps to compile and launch the game:

1. Install MinGW and SFML on your computer (Here's a good video tutorial for installation: https://www.youtube.com/watch?v=M3zYZTdlqyg&t=302s).
2. Install Git Bash if you haven't already
3. Download project repository and copy files from src folder to a separate folder outside copied repository.
4. Copy all the .dll files from your SMFL installation bin folder to the same folder where you copied src files.
5. Delete lauri_hello_test.C file from your copied files
6. Copy "misc" folder from project repository and paste it inside the folder where you copied all the files in step 3.
7. Inside the folder of your copied files create a folder named "src".
8. Move level.txt, level2.txt, level3.txt, level4.txt, level5.txt files inside src folder you just created.
9. Open copied folder in Git Bash Terminal and run following commands:
    a. g++ -c *.cpp
    b. g++ *.o -o main -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio

10. Run the game by double clicking main.exe file that was generated in your copied folder

Basic user guide

The game starts with a window that allows you to either “play” or “exit”. Exit, of course, closes the game and play moves you to choose the level you want to play. The level numbers correspond to the difficulty of the game; one being the easiest and five being the most difficult. After clicking on a level, the game window will open.

You start out the game with enough money to place one basic tower. This can be done by first clicking on the image of the tower (or snowman in our case) and then clicking on the part of the map you want to place the tower on. Once you press “Go” or the timer has run out, the first game round will start. During the round, you will be able to place more towers and upgrade them when you have enough money to do so. Upgrading works by clicking on the tower on the map with the left mouse button. You gain money by destroying the enemies and selling towers. Selling the tower gives you a set amount of money and it is done by right clicking on the tower on the map.

You will win the round by destroying all the enemies before they reach the carrots. The rounds will repeat ten times and get more difficult as each round goes by. The game will be won when all ten rounds have been completed. By closing the game window, you will return to the menu to either stop the game or to choose another level to play.

Testing

“how the different modules in software were tested, description of the methods and outcomes.”

In the beginning of the project, we created classes for the towers and enemies. For those classes we also created their corresponding test files which test their functions and uses. However, we soon got the graphical user interface up and running and from that point on, it was easier and faster to test the code by running it through the GUI, than by creating a separate test file for everything.

From very early on, we decided that we would not have a separate person to test the code, but everyone should test their own code. This sped up the coding process because no-one had to wait around for someone else to test their code before fixing it. On the other hand, everyone was everyone’s tester, and we commented and changed each other's code quite freely. (Always of course reporting our actions and intentions as we did.)

We are also quite proud of the fact that our game was simultaneously developed and tested on three of the most popular operating systems: Windows, Ubuntu (Linux) and MacOS (4 if one was to consider WSL separately). Therefore, we are relatively confident in our solution working somewhat reliably on all of the aforementioned platforms. We discovered that there are differences in how those platforms operate, for example opening files through the program in Linux wants the relative path to be included, while on Windows works the exact opposite way. Therefore, we tried to program our game in a way that it would retain its core functionality regardless of the operating system. The only somewhat unsupported platform is the Windows Subsystem for Linux, which does not natively support neither graphical interface or audio. If one has their WSL set up to support both, the game should also work there, however since we verified our solution on native installations of all three platforms, we decided to not focus on WSL support specifically.

Work log

"a detailed description of the division of work and everyone's responsibilities. For each week, a description of what was done and roughly how many hours were used for the project by each project member."

Main areas of responsibilities (+ estimate of the time spent on the project):

Roosa (54h): Menu, level choosing (with its graphics), menu and game communication, buttons, level's sidebar, removing towers, adding towers by clicking on their image in the sidebar (time estimate very general since I worked sporadically)

Karl (50h): Tower classes, level design (layout and balancing), game audio, reading non-hardcoded maps from files and creating the files, Ubuntu testing and documentation.

Lauri (45h): Enemy classes: abstract and three derived, Graphics for towers, enemies, tiles and background for menu, compiling process on mac and CMake in general, project documentation.

Severi (80h): Game map generation based on vector input. Tile abstract classes and different variations. Enabling tiles to hold either enemies or towers as pointers with their class methods functioning. Enemy movement with pointers transferring from tile to tile and tiles handling path for enemies. Towers detecting enemies, turning towards them and shooting at them. Snowball animation with enemies taking damage as the ball hits. Game loop with repeating rounds with specific enemies spawning.

(~17h was spent in collaborative meetings: planning, giving feedback, presenting results and contemplating solutions to problems.)

Weekly accomplishments

Week 44

We arranged our first meeting one day after the groups were officially finalized. We started by getting to know each other and going through everyone's ideas and understandings over the project. Research over the topic and SFML in general. Trying to get an idea of how big the project will be and what challenges we'd face. Trying to contact the advisor.

Week 45

We created the plan for the project and went over different classes and attributes that we think would be useful in addition to the general rules of the game. We divided some of the first tasks from which to start. At this point we had a general idea of the project, although some details were left to be clarified on the go. The SFML-library was quite new to everyone still, so its optimal use was not clear right from the beginning and it was difficult to estimate the workload.

Week 46

The first real week of programming. The basics of the enemy and tower classes were constructed. A few derived enemy types were also made. At the same time the first windows were created with the help of SFML tutorials. Menu screen and basic game level took their first steps with the ability to track user input. Button class for easier button usage was made along with some basic textures to test enemies and towers "on the field". Our advisor also dropped by to our weekly meeting and briefed us on the project technicalities. We reached an agreement that we would reach out in case of difficulties or problems but otherwise no extra sync sessions would be necessary until the demo in the end of the course.

Week 47

Different level options were made to the menu screen. You could now start the game object from the menu. The enemies and towers got pretty much their final form at this point. The main game part was at a point where you could click the road to create visible enemies and click them again to kill them. If you clicked the grass tiles, the game created towers and upon clicking again they got upgraded. At the end of the week the towers were able to detect enemies moving near them and shoot them (without visuals). Also if an enemy reaches the end, the game stops and does not respond to user input anymore. There was also plenty of testing regarding CMake and compiling. Also more graphics were developed.

Week 48

Background image added to the menu screen. Waves got introduced to the game with enemies spawning in predetermined groups and the next patch spawning after the previous has been defeated. When enough waves have been killed, the game is won and no more enemies spawn. The idea of non-hardcoded levels was presented and implemented. The levels would now be read in from a text file (essentially csv) with a certain format. That allows editing the levels on the go, provided that the user knows the file structure. This meant also multiple levels that could be chosen from the menu. In addition, snowballs were created to visualize the towers shooting at the enemies. The enemies also now face the enemy they are attacking, turning dynamically. A sidebar to the level was made so you can choose which tower to create, and a button on the sidebar to skip the grace-period between waves. You can't buy towers if you don't have the money for it (money <= starting bonus + killing enemies). The sidebar also has a wave counter.

Week 49

The final week had a lot of fine tuning and bug fixes. We got the music working on three of the most popular operating systems: The solution was tested and proven to work on native installations of Windows, MacOS and Ubuntu. The only unsupported solution is when one tries to run the game through WSL, as WSL does not have native support for GUI or Audio. This does not, however, affect the work of the rest of the game. The tower statistics were finally separated so that different towers have different damage, speed, cost etc. Also, the Project Documentation was made.

Overall challenges with game balance

In general, what we mean by the balance in the game is trying to figure out and design the appropriate level of perceived difficulty for the player. The goal is to have the game challenging enough so that it wouldn’t feel too easy and still rewarding to beat, but also easy enough that it can actually be beat without unnecessary levels of frustration. Also the game should be getting progressively more difficult while factoring in that the player skill will also be increasing with play time.

This has been a multilayer problem. The first layer is the levels themselves. How we would summarize difficulty in a level comes down to the distance of the path and the enemy exposure to towers. The longer the path is, the higher the exposure to towers, meaning the player has more time to damage the enemies and thus the game is easier. As opposed to this, the shorter paths mean less exposure, less time to do damage and are therefore more difficult, requiring more thought and strategic placement of towers by the player. Therefore, The earlier levels are designed to have a longer path and more curves, while the later levels are getting shorter and less curvy.

Second layer is the enemies. With our limited time, we did not have the opportunity to design overly complex enemies with special abilities. Therefore the different enemies are distinguished by three major attributes, which are their health pool, movement speed and the money they drop upon death. The balance is mainly determined by the health vs speed relationship. However, in addition to this, the way and order how enemies spawn to the map has been thought through. This turned out to be slightly tricky due to the current technical implementation where only one enemy can occupy one tile at any given time. Therefore we experienced a problem where our fast moving weak enemies would be blocked by slow moving strong enemies. In the presence of more time, we would like to get past this limitation and enable several enemies to occupy a single tile simultaneously.

The third layer is the towers themselves. The towers have three main distinguishing attributes, which are damage per shot, range and the rate of fire. Those three are all quite tricky to find a suitable balance between, while still making the towers feel unique. An additional level of complication is the upgrade system which we decided to implement for the towers. This introduces the topic of currency, which tightly also ties to the enemies and their money upon death drops. The upgrades are always designed to be cheaper than buying another tower because otherwise there would not be an incentive to upgrade the towers at all. Additionally, having the ability to upgrade towers brings in a new dynamic, allowing players to upgrade an existing tower for better stats in case of an emergency when they perhaps do not have enough money to buy a new tower. We believe that our game with the towers and their upgrades allows for various different approaches to play each level and hopefully will remain interesting.

The hardest thing to get right however is the balance between enemies and towers. We went through several phases with this and had to make many adjustments in order to get to the point where we are now. First of all we had a balance which was clearly unfair to the enemies. A single tower was capable of taking care of majority of the waves and the player got filthy rich immediately – clearly not a very interesting game. After discovering a bug which was affecting the tower cooldown between shots, we had a phase where the enemy waves were almost impossible to beat with the starter money and money upon death values. After a few more tweaks we got to a state where we are now. The game might still prove to be a challenge at times, especially on the later levels but we have proven it to be still beatable. Of course the balance could still be improved and we'd actually be interested to see if someone can find a clever exploit or a sort of "meta" on how to trick our balance system and play the game more efficiently.

Conclusions

All in all, we believe that the project has been mostly a success. At least we are quite proud of the product we have put together in just a few weeks and have even ourselves had fun playing our own game. As noted before, it was difficult from the start to estimate exact workloads from the get-go and we were all learning as the project went on. However, we always tried to help each other when anyone needed help and tried to solve the problems together. Perhaps a comment from our part is that while WSL was the suggested way of working for the exercises in the earlier part of the course, then at least in the project phase it turned out to cause the most issues, mainly because of WSL not supporting graphical user interfaces nor audio out of the box. All of those problems would have been avoided by working native installations (or virtual machines) of Windows, Ubuntu or MacOS from the get-go.