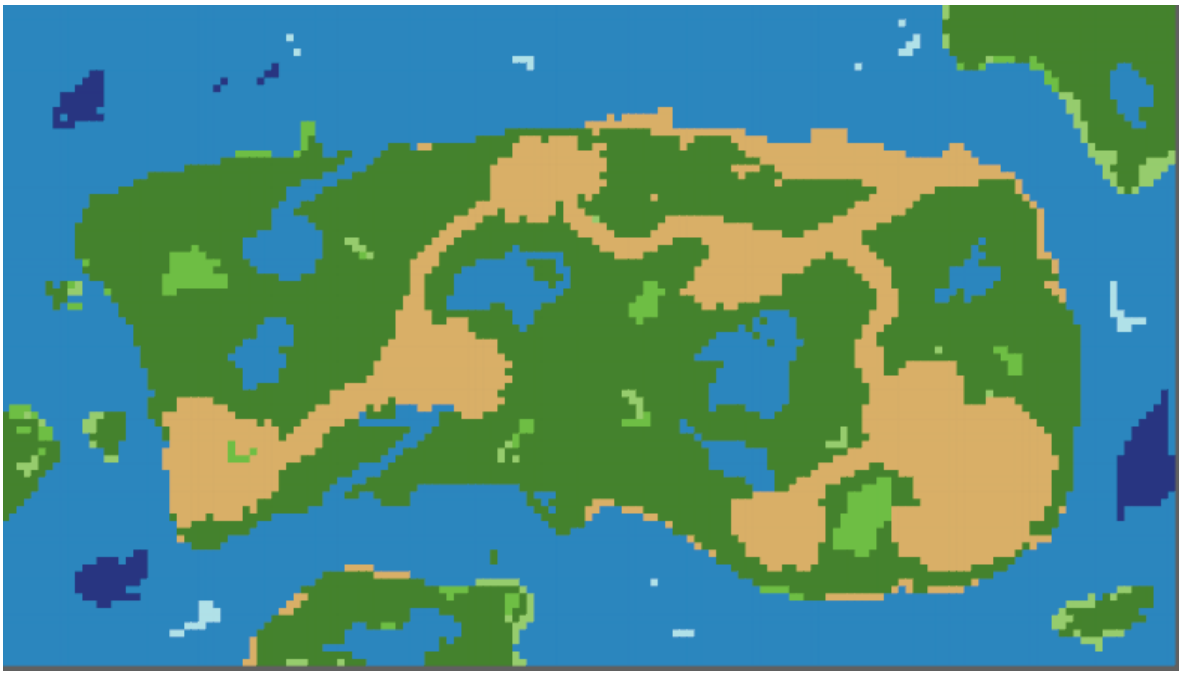
Scope of the work: what features and functionalities will be implemented, how is the program used, and how does it work

Our project is a tower defense game with waves of enemies attacking the player's base. Turrets can only be placed in designated spots on the map. There are multiple types of enemies and turrets with different attributes. Each turret has a price that it costs to build, and a selling price which is 70% of the building price. Money is earned by turrets destroying enemies.

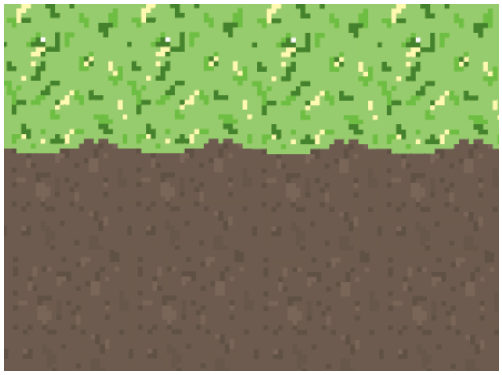
The game will feature multiple maps (levels) the player can play. The goal of a level is to complete the level so that your base's health points will not get to zero. Should that happen, you lose the game. Each enemy that reaches your base will deal damage to it.

Each map is read from a text file. This allows the creation of custom levels. Certain characters in the file describe either a road, grass, or a tile to place a turret in. A turret spot is the size of 2x2 tiles.

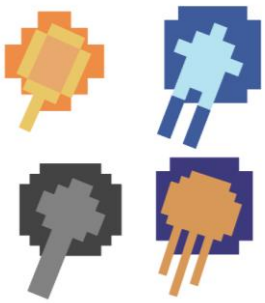
The dimensions of the map (image ratio) are 4x3 and the map is tile-based. There are pre-set turret spots and the path for monsters.



Game's map with about 5 levels. Current level will be marked by the flag. Unreachable levels will be hidden.



Smooth transition between grass and path

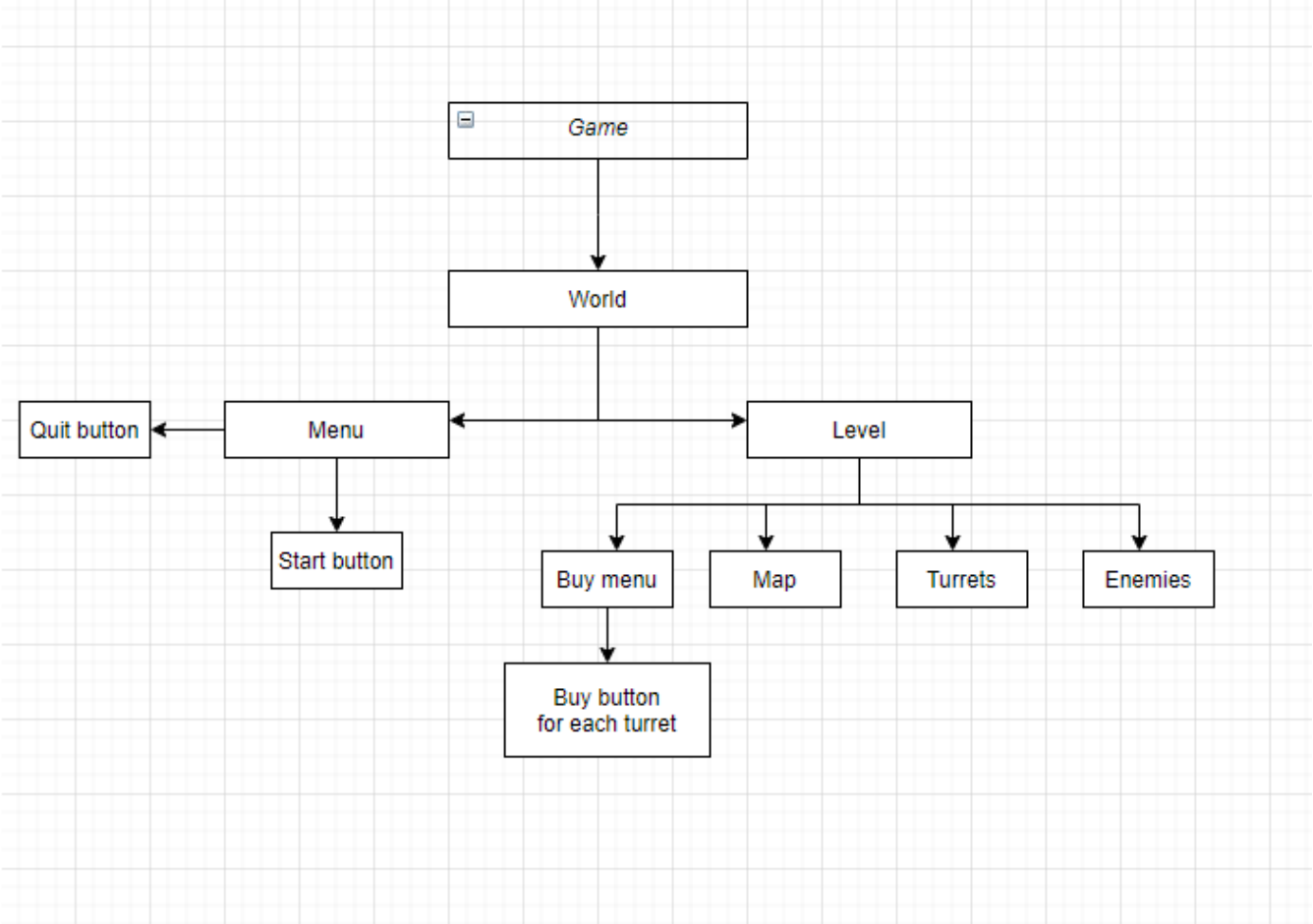


Turrets concept

The high-level structure of the software: main modules, main classes (according to current understanding)

Main classes (for most classes, we have both .cpp and .hpp files):

- The most top-level class is Game, which contains everything
- Menu class, from which SideMenu and MainMenu inherit from
- MainMenu: offers the chance to choose the specific level you wish to play. You can't play a certain level without having completed the previous one.
- SideMenu: has list of turrets that the player can buy
- Map class: 4x3 dimensions, tile-based, has path for enemies, and specific spots on which to place the turrets, we will read it from a text file that has dots denoting non-turret and non-path area, hashtags denoting path area, and O (letter) denoting turret area
- ResourceManager, which manages memory-heavy files (images) dynamically
- Level class, containing enemies, turrets, SideMenu, and map
- Abstract Enemy class, from which specific enemy types inherit from
- Abstract Turret class, from which specific turret types inherit from

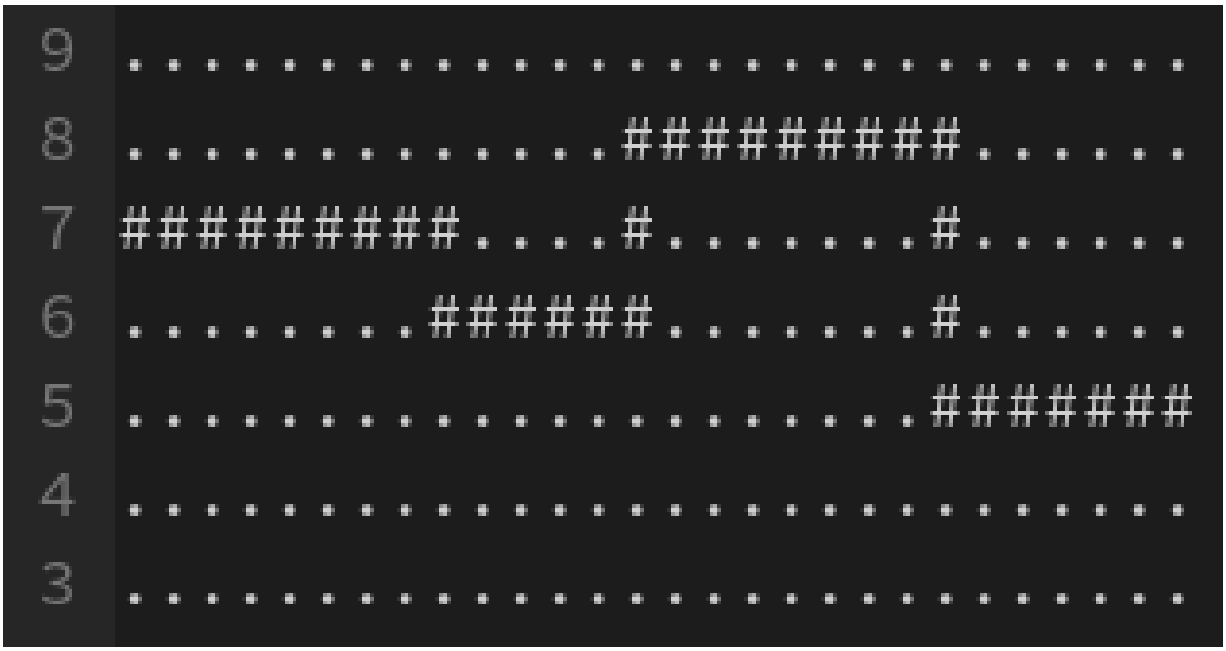


Above is the initial layout of the project, which is subject to change in the future. Game is the class that implements and runs the whole tower defense game. It is the highest class in the work. It creates the window in which the game takes place. It also sets the environment for a steady FPS display (we will set it around 60-200). The World class will layer out the Menu and Level, respectively (so that the background would stay in the lowest layer and buttons etc on higher grounds). Additionally, it governs Menu and Level, as the game switches between these states. Level is the “game” itself in the sense that then the level itself starts with all the turrets and enemies. You can add turrets by buying them with money and then dragging them to a certain position. The main menu consists of “Start” and “Quit” buttons.

Implementation-wise, it’s also worth noting that the graphical resources are allocated on the heap, so as to prevent stack overflow. There is a ResourceHolder class that manages the resources. It is a class template that takes in an identifier and a resource and then loads the

resource from a given file. There's also an option to load shapes (for buttons), which doesn't need filenames. That way, we don't need to bother with using *new* or *delete* in our project. The game should be fairly lightweight and most computers, even slow ones, should be able to run it without problems (it's an offline game, so no Internet connection is required either).

Below is a basic model of a map file where dot is the space for non-turrets and hashtag the path for enemies. This also means that the map design is tile-based: each tile has a texture segment of its own.



There are also some constant files storing the constant values for the project.

Turret class is implemented as an abstract class that contains the common properties and actions of different types of turrets. The specific turret types inherit from the abstract base class and add the type specific features. Turrets can be placed in certain locations and removed by selling them. Each turret type has its own price, attack power, bullet speed etc.

Enemy class is also implemented as an abstract class and the specific enemy types inherit from it. The classes handle characters that try to attack the base and can be destroyed by turrets. Each enemy has its own health points, attack power, speed and price money. Some enemies

have actions that only they can do (for example, multiplying to smaller enemies). By defeating enemies, players can gain money that they can spend to buy turrets.

The planned use of external libraries

The SFML (Simple and Fast Multimedia Library) will be used for creating graphics, sound and user input. No other external libraries will be used.

Division of work and responsibilities between the group

Kerkko: Game sound, class implementation, project management (schedule, organizing meetings etc.)

Mark: SFML implementation, graphics, designing the class hierarchy

Minh: Design of enemy characters, turrets

Niko: Implementation of basic classes (main, level, turrets, enemy, map, etc.), the core elements of the game

The responsibilities, however, don't limit to these. The project is a huge effort and requires maximum commitment from each member. Having completed your own part of the project does not mean that the project is complete, as all parts must be completed so that we have a playable game.

We are committed to a strict work ethic. It is crucial to keep up with the schedule as failure is not an option and the members of the group aim for a good grade. We have expertise in using SFML and that should suffice for the scope of this project. On the other hand, 75% of the project team are new SFML users and that might seriously slow down the project development process. Time has to be spent installing tools and reading documentation. That is time off from writing code.

We have set up a dedicated Discord server to address project-related issues and plan and have meetings in. The server includes numerous subchannels, e.g., one for TODO, one for general observations, one for error and problem handling and so on. A Telegram chat has also been set up, but it is the secondary communication channel in case Discord is down.

Planned schedule and milestones before the final deadline of the project

5.11. Project plan submission

6.11.-> Weekly meetings on Tuesdays to ensure we are on schedule and no problems have arisen

28.11. First completely functioning version of the game, from now on honing minor issues in the game

12.12. Final git commit

17.12. Project evaluation