ELEC-A7151: Object Oriented Programming with C++

Project Plan

Julius Järvilinna 596174
Tousif Zaman 980285
Tomi Mikkola 793155
Konsta Kemppainen 1008250

Project scope and game description

We are making a conventional Tower Defense (TD) game with around 6 unique tower types and a decent amount of different enemies, maps, and enemy rounds. Our group tries to target grade 5 from the project, so our aim is to implement enough extra features and maintain a good quality to get there. In our TD game, towers can be bought, sold and upgraded during or in-between rounds. We discussed how we want to implement the tower upgradability and decided that will do at least simple upgradability. There was also willingness in the group to implement more advanced upgradability (for example tower looks or changing functionality) if we can get enough extra points for that.
Different enemies have different health and movement speed values and they can also have some abilities like splitting into other enemies upon death. We will have different damage types, meaning that some towers are more effective against certain enemy types. We're also planning on including some kind of boss enemy that can interact with the player's towers in some way. Player actions, towers and maybe some enemies will have sound effects. Our level map would consist of a background image, where the enemies path would consist of predetermined points on the map. Towers can't be placed on the path of the enemies.
Maps could also have obstacles such as trees and rocks that towers can't be placed on. We thought also of making the map to consist of little squares, but decided we want rather to make it a background image with an array of 2D points to describe the enemy path, because we thought it would look cooler.

The in-game user interface allows players to buy and place towers, hover over towers in the shop to gain more information on them and start and fast forward rounds. We might add some keyboard shortcuts for these actions. Fast forwarding could allow for example 2x and 4x speeds and it could be implemented so that enemy moving speeds, tower fire rates and projectile moving speeds are multiplied with either 2 or 4 (with selected fast forward speed). The UI shows how much money and lives the player has and how much towers cost. Towers placed on the map can be clicked to show their range and open up a menu that lets the player upgrade and sell the tower. We might also add targeting options such as those in newer Bloons TD games. Our project will also include a list of high scores. Each defeated enemy will give a certain amount of points. Enemies with more HP drop more money. If an enemy gets past the player's defenses, the player will lose lives depending on how much HP the enemy has left. Maps and enemy rounds are read from files. We might add another game mode where enemy rounds are randomly generated. It will be possible to pause the game and also save and quit to the main menu. The game state is saved in a file that can be loaded again from the menu. If we have the time, we'll include a level editor that allows the user to create the path the enemies follow as well as place obstacles that towers can't be placed on.

High level structure of the code

Our design philosophy is to plan everything out carefully beforehand so that we can focus on coding our game logic instead of worrying about organizing our code.

File hierarchy

In order to get a good idea of the high level structure of the code, it might be a good idea to plan out the folder hierarchy. Most header files are omitted due to their trivial nature.

```
include/
        |__logic/
                |__ …
        |__gui.hpp
src/
```

```
|__main.cpp
|__logic/
        |__application.cpp
        |__map.cpp
        |__object.cpp
        |__game.cpp
        |__enemies/
                |_enemy.cpp
                |_exampleEnemy1.cpp
                |_exampleEnemy2.cpp
        |__towers/
                |_tower.cpp
                |_exampleTower1.cpp
                |_exampleTower2.cpp
        |__projectiles/
                |__projectile.cpp
                |__exampleProjectile1.cpp
                |__exampleProjectile2.cpp
|__gui/
        |__main-menu.cpp
        |__game.cpp
        |__level-editor.cpp
        |__pause.cpp
tests/
lib/
```

File documentation (subject to minor changes)

main.cpp - The entry point for our application. Constructs an Application.

logic/application.cpp - The central unit that handles events, rendering, etc. It has access to a std::optional<Game> (which contains a Game instance if there is a game in progress) and a GUI instance. It keeps track of the program's state using a private state_ variable, which can be changed using special methods namely launchGame, launchMainMenu, launchLevelEditor, etc. It has a TGUI member variable that defines the current GUI.

logic/map.cpp - A Map class contains a static method that attempts to read a map from a json file. It returns a Map type that contains all the necessary information about a map (background image, enemy path, blocked regions).

logic/game.cpp - Game has an update method that is called by the Application class once every tick and it takes care of the game logic. In order to do that it stores the enemies, towers and projectiles in separate vectors. It also has a private paused_ member variable that stops all action from happening.

logic/object.cpp - An Object is a non-GUI element that is to be rendered on the screen. It's an abstract class that is meant to be inherited by enemies, towers and projectiles. An Object has a hitbox, a position and a sprite.

logic/enemies/enemy.cpp - An abstract class that defines an enemy. An enemy has a move speed and a certain amount of hitpoints. Its movement is handled by the Game class. An enemy inherits the Object class, which means it has a hitbox, a position and a sprite.

logic/towers/tower.cpp - An abstract class that defines a tower. A tower has an attack method that gets a reference to a vector of projectiles where the tower is free to add its own projectile. The attack method is called by Game. A tower inherits the Object class, which means it has a hitbox, a position and a sprite.

logic/projectiles/projectile.cpp - An abstract class that defines a projectile. A projectile inherits the Object class, which means it has a hitbox, a position and a sprite.

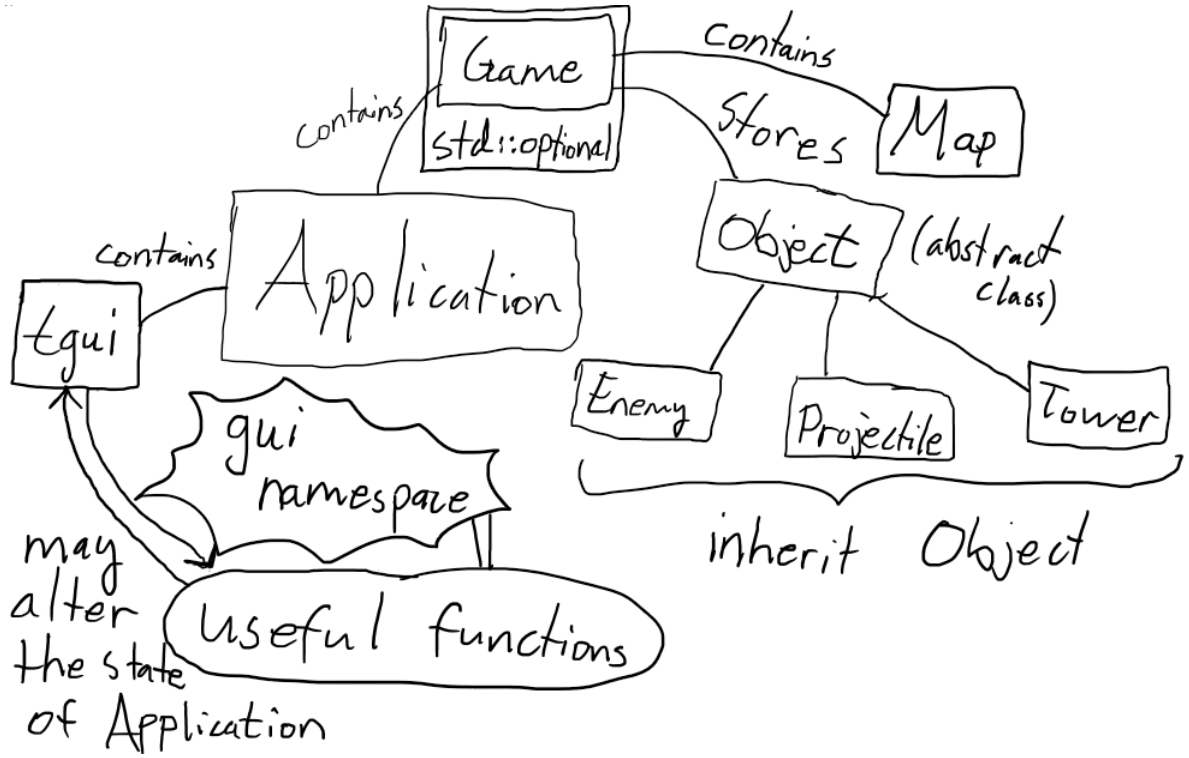
gui/main-menu.cpp - Contains a function called createMainMenu(Application&) that returns a tgui instance. The function is a part of a gui namespace.

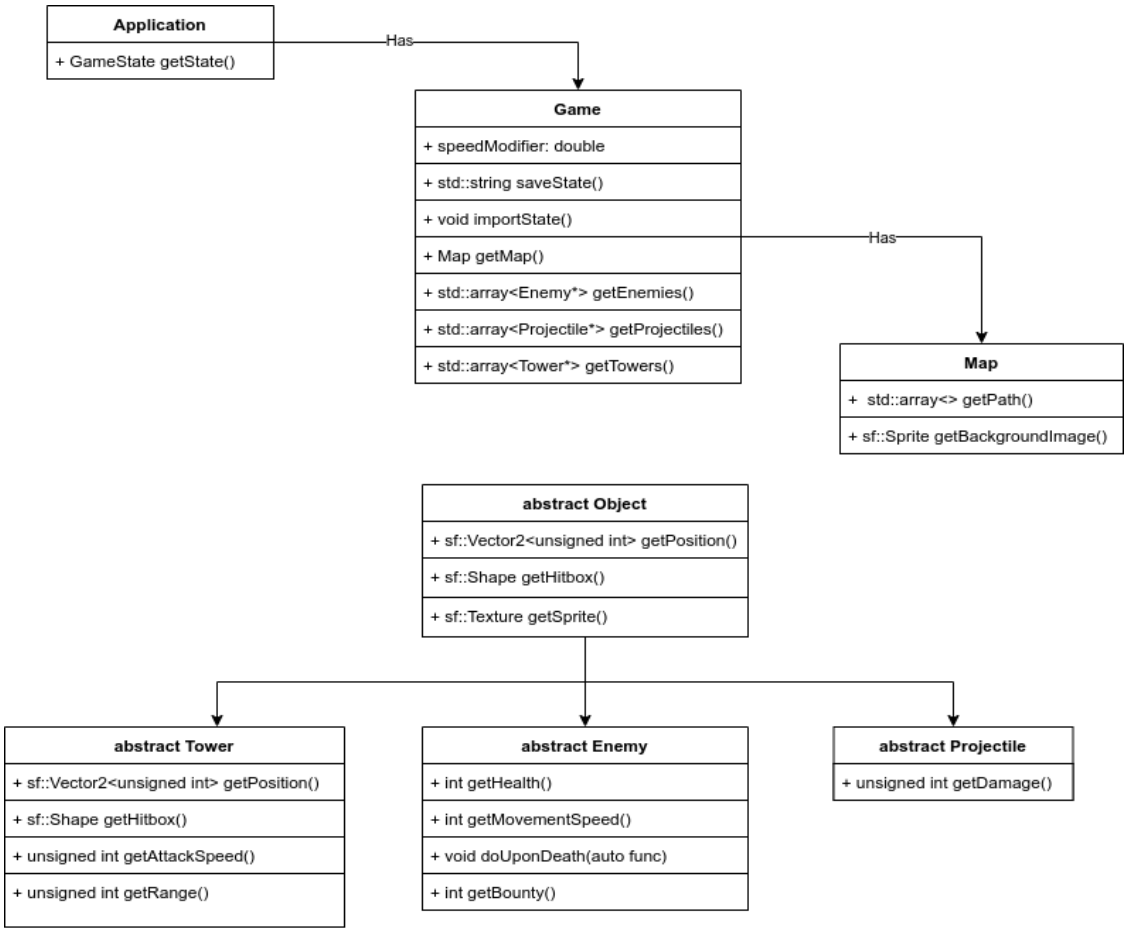
gui/game.cpp - Contains a function called createGame(Application&) that returns a tgui instance. The function is a part of a gui namespace.

gui/level-editor.cpp - Contains a function createLevelEditor(Application&) that returns a tgui instance. The function is a part of a gui namespace.

gui/pause.cpp - Contains a function called pause(Application&, tgui&) that changes the GUI so that it displays a pause menu. The function is a part of a gui namespace.

Class diagrams





We decided on some more formal aspects of the game and formalized the class relationship diagram into a UML diagram above.

The planned use of external libraries

SFML

SFML is a library that allows us to play audio and render graphics. It also lets us handle keyboard and mouse input. Why this is useful should be quite self-evident. One good thing about SFML is that it's cross-platform, which means we won't have to put any extra effort into porting our code.
The main benefit of SFML compared to other alternatives (like writing our own graphics engine with OpenGL) is that it's convenient. We have deadlines to meet and for a 2D tower defense game performance isn't absolutely crucial.

Graphical User Interface (GUI)

The hardest part of any project is always the GUI. Since we have limited time to work on the project, it would be nice to have an easy solution to the GUI problem. Our game is going to require a lot of different menus: a pause menu, the main menu and the tower selection view, just to name a few. A level editor would be the final boss of menu design. Programming all of these from the ground up would be a very tedious task.
TGUI (Texus' Graphical User Interface) is a cross-platform GUI library for SFML that is easy to use and looks decent. It'll hopefully come in handy to solve our GUI problem.

JSON parser

Our game will have to save to and load from the hard drive, and one easy solution is to simply store the needed data in JSON files. Another alternative could have been to use comma separated lists, but JSON files will offer greater flexibility.
This flexibility will be particularly important when we want to store the internal structure of a map. A map might have regions where towers can not be placed or it might need to store other information about itself that requires complexity way beyond what a comma separated list can reasonably offer.
The library we are planning to use is Niel Lohmann's JSON parser because it's simple and well-tested. It is not the fastest nor the most memory-efficient library on the market, but we can be quite certain that for our purposes it is more than sufficient.

Division of work and responsibilities

We aim to divide our project workload between our group members so that at the end of the project, the amount of time each group member spent on the project is roughly equal. It is okay for group members to occasionally do a little less work than is expected as it can be compensated for by working harder on other weeks. We have also tentatively divided our main responsibility areas between our group members.
This division is as follows:

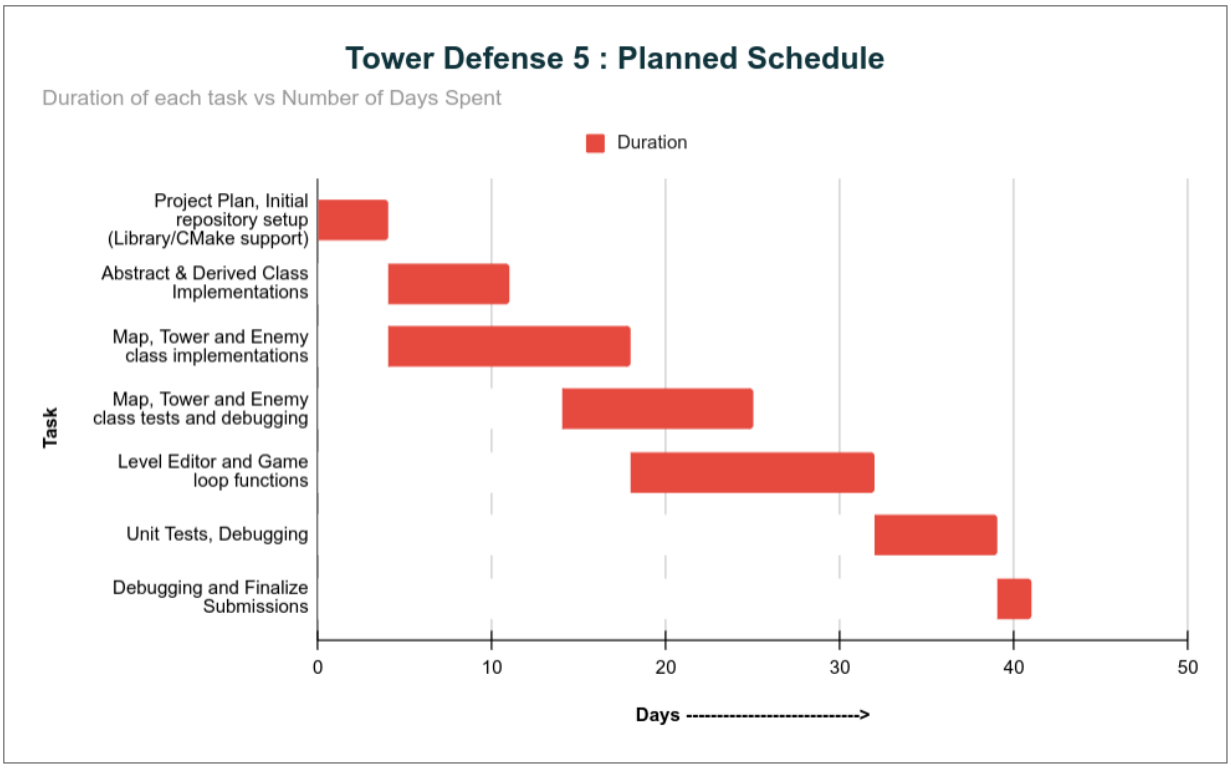
- Tomi : artistic side of our project, like designing how the towers and enemies will look like. Tomi also had interest in doing at least one tower himself.
- Konsta : takes care of map and GUI implementation
- Julius : responsible for towers and enemies implementation.
- Tousif : responsible for CMake, unit tests and the main abstract classes.

Even though each member has a certain responsibility area we will each help and support each other during the project. We estimate that the project work should take on average 10 hours or more per group member in a week. We will also use Trello to get a clearer view on what tasks are not yet started, what are currently worked on, or what tasks have already finished.

Planned project schedule

The project task split and tentative schedule for the project are added in the table and the Gantt chart below.

Task	Start	End	Duration
Project Plan, Initial repository setup (Library/CMake support)	11/1/2021	11/5/2021	4
Abstract & Derived Class Implementations	11/5/2021	11/12/2021	7
Map, Tower and Enemy class implementations	11/5/2021	11/19/2021	14
Map, Tower and Enemy class tests and debugging	11/15/2021	11/26/2021	11
Level Editor and Game loop functions	11/19/2021	12/3/2021	14
Unit Tests, Debugging	12/3/2021	12/10/2021	7
Debugging and Finalize Submissions	12/10/2021	12/12/2021	2



We plan to achieve these milestones and sync on a weekly basis. The team prefers to meet and discuss the progress and next steps Tuesday evenings.

Mock UI