AALTO-YLIOPISTO
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

Laine Mikael, mikael.2.laine@aalto.fi
Pigg Niklas, niklas.pigg@aalto.fi
Savola Joonas, joonas.savola@aalto.fi

ELEC-A7151 Object oriented programming with C++, Course Project: Tower defense 16

Espoo, 12.12.2021

Table of contents

1 Overview

Our topic is the tower defense game. The main objective in this game is to defend against enemy attacks that happen in waves. Enemies move along a single predetermined non-branched path and their objective is to reach the end of it. By placing different defensive towers, the player tries to stop the enemies from reaching for their goal at the end of their path. If enemies reach the end of their path, the game is over, and the player loses.

The game begins with a map that has a path running through it. The enemies will spawn at the beginning of the path on the upper or left side of the map. In the beginning, the player has some money to build their first towers and upgrade them. By destroying enemies, the player will gain more money that can be used to build more towers and upgrade existing towers.

Each tower has a fixed range that they can shoot. If the enemy is in the range of the tower, the tower will shoot the enemy. The towers will prioritize enemies that are the farthest along the path. There are four types of towers that have different attacks and specialties. The basic tower shoots one projectile at a time to one enemy. The cannon tower is like the basic tower but does more damage and its projectile can hit multiple enemies along its path. The pulsing tower will shoot projectiles all around it with a fixed time interval. The freezing tower will shoot projectiles at enemies doing damage and slowing them. Each tower costs a different amount of money. The player can place towers on the map and change their position between enemy waves. Additionally, the player can move towers on the map during a wave as well, but cannot upgrade, delete, or build more.

There are four types of enemies the player will encounter. Infantries are the basic fodder enemies with low health and an average speed. Tank enemies have a higher health but are slower. Scout enemies have low health, but they are faster. Container enemies split into several Infantry enemies upon their destruction. Each enemy has an associated colour, as well as an outline indicating how much health they have. Enemies make a distinct popping sound when they are destroyed.

The game includes five maps of increasing difficulty with each map having a specific number of enemy waves in it. Additionally, each wave is unique in its composition, i.e., the first wave of Map 1 will be different from the first wave of Map 2.

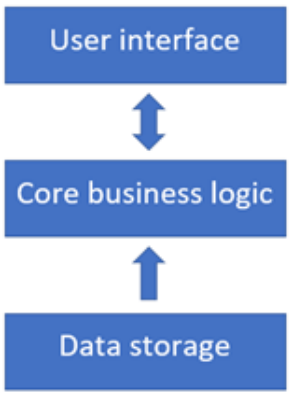
The game can be played with a mouse by selecting the various buttons on the graphical user interface, such as selecting the map, starting, pausing, and ending the game as well as controls related to building, moving, and removing towers. Some of the GUI functions also include sound effects to indicate successful and unsuccessful actions.

The maps of the game are loaded from specific text files, where there is a standardized form for a map. Furthermore, the enemy waves are also stored in said file and can be customized as well.

2 Software structure

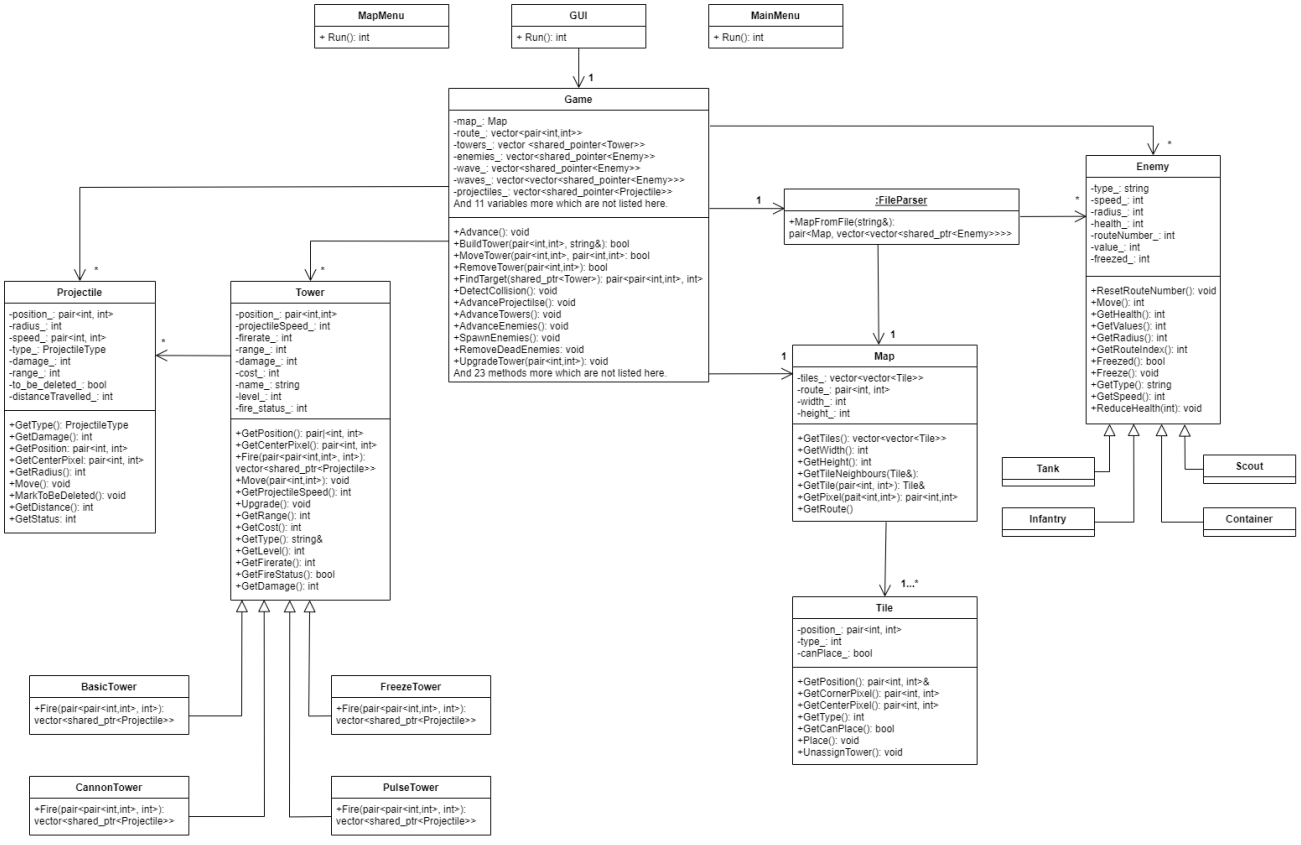
Software's overall architecture can be presented as a three-layer architecture where top layer is user interface, middle layer is core business logic and bottom layer is data storage, see picture 1. User interface layer's purpose is to provide interface between user and program. SFML external library were used to created graphical user interface and track user's mouse events. To keep the separation and independence between the layers SFML graphical interface were only used in user interface layer. This makes possible to change the user interface layer in the future more easily if needed.

User interface and core business logic uses bidirectional communication between each other. Core business logic layer is responsible for creating the game instance and simulating how the game behaves, for example how enemies move and when towers start to shoot enemies. Core business logic communicates game situations to the user interface layer which provide graphical feedback to the user. User interface layer provides any user input to the core business layer. SFML audio interface was used in both user interface layer and in core business layer, but only 5 rows of code in core business logic layer are related to SFML audio interface. Data storage layer represents text-based map -files which are saved to the local computer. Core business logic can load map files from the data storage but cannot save map files there.



Picture 1: Layered arhitecture of the software.

See picture 2 for more detailed class structure. Main classes are Game -class and GUI -class. There are totally 17 different classes and one object. More detailed class descriptions are on next subchapters.



Picture 2: Class diagram of Tower defense -game.

2.1 GUI, MapMenu, MainMenu, AboutMenu -classes

GUI, MapMenu and MainMenu classes represent different views of game's graphical interfaces and listens for user input events. SFML library's graphic interface is used to create the graphical user interface. All three classes have only one method called Run() each. MapMenu and MainMenu instances create static menu views where user can choose to start a new game, close the program, and choose between different maps. AboutMenu is also a static that shows the "about" section of the main menu. A GUI instance communicates with Game -class instance. The GUI instance creates a new Game instance and gets the information about current game situation from the Game instance and uses that information to draw a dynamic graphical interface to the user. GUI instance uses the methods of the Game -instance to update the game situation.

2.2 Game -class

The Game class's main purpose is to simulate the game situation and behavior. It has 35 methods and 18 variables, and it refers to Projectile, Tower, Enemy and Map -classes. Key variables are shared pointer vectors where for example information about enemies, projectiles and towers are stored and a variable where the game's Map instance is saved. The main methods are related how to manipulate game situation on every game tick and when the GUI instance calls the Game instance's methods. The main methods related how to manipulate Enemy instance are SpawnEnemies() which creates enemies at the beginning of each enemy wave and RemoveDeadEnemies() which removes that have zero or less health. BuildTower(), MoveTower() and RemoveTower() methods are used to add, remove or update Tower -instances of the game. AdvanceProjectiles(), AdvanceEnemies(), AdvanceTower() are used to perform basic actions per every game tick. For example, AdvanceEnemies() and AdvanceProjectiles() updates the positions of all Enemy and Projectile instances to move one tick. AdvanceTowers() checks if Tower instances should use their internal methods. FindTarget() and DetectCollision() methods are used to manipulate Tower, Enemy and Projectile instances in relation to each other. FindTarget() checks if there are Enemy instances within a Tower instance's range and DetectCollision() checks if any Projectile instances and Enemy instances collide. Game instance uses the FileParser object's MapFromFile() method to create a Map instance and the game's enemy waves from a text file.

2.3 Map and Tile classes and FileParser -object

An instance of Map -class represents the game's map. The Map -class refers to the Tile -class. The Map -class has tiles_ and route_ key variables. The Tiles_ variable is two-dimensional vector where Tile instances are saved. The Tiles_ vector is basically a vector format presentation of the map. When creating a new Map instance, the path that Enemy instances follow during the game is created and saved to the route_ variable as a vector of pair of integers where pair of integers are x, y coordinates. The main methods of the Map -class are getters to return route_ and tiles_ variables. The Tile -class has three variables. The position_ variable is pair of integers and represents x, y coordinate of the tile, the type_ variable saves a Tile's type. The possible Tile types are Grass, Route, Start, End and Wall. Each Tile has a status to represent whether a tower can be placed in this tile or not. This information is saved to the Tile's canPlace_ variable as a Boolean. As mentioned briefly in the Game -class FileParser object is used to create a Map instance and enemy waves from text file. The following is the correct structure of the text file:

td_game savefile
map1
1.0
w15
h15
structure
442444444444444
401000000000004
401001110000004
401001010000004
401111010000004
400000010000004
400000011111004
400000000001004
400000000001004
400000011111004
400001110000004
400001000000004
400001110000004
400000010000004
444444434444444
waves
1I2S3T

The first row is the identifier that the text file is a save file for the tower defense game. The second row has the name of the map, the third has the version. The fourth and fifth rows tell the width and height of the map (in tiles) respectively. The sixth row is the tag for the map structure. The structure in this example is 15x15, where a number

represents the type of the tile; 0 for Grass, 1 for Route, 2 for Start, 3 for End, and 4 for Wall.

What follows is “waves”, which is the tag for the enemy wave content that follows. The parser reads the number (which can only be a single digit due to parser limitations) and the following letter, which represents the number and type of the enemy, I for infantry, S for Scout, T for Tank and C for Container. The waves are in a reverse order, so the topmost row is the last wave, and the bottom most row is the first wave.

2.4 Enemy, Infantry, Scout, Tank and Container -classes

The Enemy-class represents all the enemies moving across the game map. Each enemy has a route number index, type, speed, radius, health, value, and freeze duration. There are 4 subclasses of Enemy: Infantry, which is a basic enemy type; Tank, which is slower but has more health; Scout, which has the same amount of health as infantry, but it is faster; And container, which splits into several infantry enemies upon its defeat. The main method of the Enemy -class is Move() which is used to update the Enemy instance’s routeNumber_ variable based on the old value and the Enemy’s speed attribute. Enemy instances are created in Game -instance.

2.5 Projectile -class

The Projectile-class represents all the types of projectiles shot by different types of towers. There are three types of projectiles: basic, which is destroyed when it hits an enemy and does damage; cannon, which can go through several enemies while doing damage; and freeze which slows the movement of an enemy that it hits. Each projectile has the variables of: position_, radius_, speed_, type_, damage_, range_, to_be_deleted_ and distanceTravelled_. The position variable stores the x and y coordinates of the projectile. Radius is an integer representing the radius of circle that is the projectile, as all projectiles are represented as circles in the game. Speed is a pair which stores both the speed in the x direction as well as the speed in the y direction. Damage represents the damage the projectile does to enemies. Range is the range of the tower that shoots the projectile. Distance travelled is the distance the projectile has travelled since being fire and when it becomes greater than the range or a basic or freeze projectile hits an enemy, the projectile’s to_be_deleted_ is set to true and the projectile is deleted at the next game tick. The projectile also has several getter methods to get its

private variables and the main method is the Move() method which moves the projectile to the next location, according to its location and speed and it updates distanceTravelled_ variable.

2.6 Tower, BasicTower, FreezeTower, CannonTower and PulseTower -classes

The Tower -class is the base class for the towers. A Tower instance represents a tower which can shoot enemies when enemies are within its range. It has four subclasses: BasicTower, FreezeTower, CannonTower and PulseTower. BasicTower shoots standard projectiles with medium fire rate, its projectiles cause medium damage, it has long range, and it is cheap to build. Freeze tower shoots projectiles which slow enemies' movement. FreezeTower has slow fire rate, it has medium range, and it is very cheap to build. PulseTower shoots eight projectiles (on every 45-degree starting from 0 degrees). PulseTower has slow firerate, its projectiles cause medium damage, it has medium range, and it is very expensive to build. CannonTower shoots big size projectiles. It has very a slow fire rate, but its projectiles cause extreme damage, it has long range, and it is expensive to build. The main method of the Tower-class and its subclasses is Fire() which is used to shoot projectiles towards enemies. Key variables are position_ which represents the x and y coordinates of the tower, projectileSpeed_ which is the tower type specific projectile speed, range_ which is the tower type specific firing range, firerate_ which is the tower type specific fire rate and damage_ which is the tower type specific damage. Tower -instances are created in a Game -instance.

3 Instructions for building and using the software

The program can be compiled with Make. The /src folder has the Makefile that is used to compile with Make. The Makefile has multiple commands. "Make main" will compile the program to an executable file called "main.out". "Make clean" removes all objective files and the executable file from the source code folder. "Make run" executes the executable file "main.out". "Make valgrind" runs the Valgrind for the executable "main.out". "Make all" combines commands "main", "run" and "clean". It compiles the program, runs it, and cleans all objective files and the executable after the program terminates.

When it comes to external libraries, the program uses SFML for graphics and audio implementation. The libs folder includes SFML headers and precompiled SFML libraries for Linux (GCC - 64-bit) and Windows (GCC 7.3.0 MinGW (SEH) - 64-bit). The SFML headers are included in compiling in Makefile. We also tried to link the precompiled libraries in Makefile but we could not get that working correctly. One reason could be that the libraries are compiled with a different compiler or with a different compiler version. We did not try to build SFML ourselves since we had no time to learn how to do that correctly.

Due to things discussed above, the end-user needs the SFML installed for example through some package manager. We listed some examples of installation commands for a few package managers below:

apt (Debian based Linux): *sudo apt-get install libsfml-dev*
pacman: *pacman -S mingw-w64-x86_64-sfml*

We have made sure that the program works with Aalto virtual workstations (Linux) without any installations needed.

If the program fails to load data from the map text files throwing CorruptedSaveFile exception, there is a problem with the text file line-endings. The map text files are saved with UNIX text line-endings, and we have tested that those work in Linux and Windows with MinGW. But if for some reason those line-endings have changed spontaneously causing issues on Linux, all you need to do is save the map text files again with UNIX line-endings.

3.1 How to use the software

The user starts the game and the first thing shown is the start menu. The start menu has “play”, “about” and “quit” buttons. The “about” button shows some information about the game. The “quit” button closes the program. The “play” button leads to the map selection menu. The user clicks the "play” button, and a list of playable maps is shown to the user. There are five different maps to choose from. After choosing a map, the user presses the chosen map. Every map name acts as a button in the map menu. In the beginning, the map is empty containing only the enemy attack path.

At the beginning of the game, the user has some gold that can be used to buy new towers as well as upgrade them. The user can place a new tower by first selecting a tile from the map and then clicking one of the towers in the sidebar. The tile that is selected is

highlighted in blue. If the user has not selected a tile and tries to buy a tower by clicking one of those tower icons, nothing will happen. The tower is always placed at the centre of a tile. Tile can have only one tower at once. The tower cannot be placed on the enemy path. The sidebar shows the money user has left and the price for each tower. By destroying enemies user gains more money.

The user can upgrade, move, and delete towers. The tower can be selected by clicking it. The selected tower will be highlighted with red borders. If the user wants to upgrade a tower, it should be selected first, and then by pressing the up-arrow icon in the sidebar the tower can be upgraded. The upgrade will improve the tower. Upgrading costs 50% of the tower's original price. The selected tower can be deleted by pressing the trash can icon in the sidebar. Deleting the tower will return money to the user. The selected tower can be also moved by pressing the blue arrows icon in the sidebar. The tower will be moved to the tile highlighted in blue. The user cannot buy, delete or upgrade towers during enemy waves, but towers can be moved both in edit mode and during enemy waves.

After the user has placed towers on the map, the game can be started by pressing the play button in the sidebar. That will start the first enemy wave. After the wave is completed, the user will be returned to edit mode. While in the edit mode, the user can move, build, upgrade and remove towers like in the beginning. When the user is ready for the next round, the new wave can be started by pressing the play button again. The sidebar shows what wave is currently ongoing and how many enemy waves there are in total.

If the user wants to pause the game, that can be done by pressing the pause icon on the sidebar. If the game is paused, it can be continued by pressing the play icon. The sidebar has also buttons for quitting and restarting the game. The restart button will start the same map again from the first wave with no towers placed. The quit icon leads the user back to the map menu.

The user will win the game if all waves are completed. If one of the enemies reaches the end of the enemy path, the game is over, and the user loses. After the game has ended, the user can restart it by pressing the restart button or return to the map menu by pressing the quit button.

From the map menu, the user can get back to the main menu by pressing the "back" button. The program can be closed from the main menu by pressing the "quit" button or clicking the cross on the top-right corner in the program window.

4 Testing

Classes and features were tested with unit tests and manually. Unit tests were executables which tested that implemented functionality worked as intended or tracked down possible errors. Unit tests were used when for example Enemy, Tower, and Game -classes first implementations were created and tested. Unit test executables also ran the though Valgrind to check if memory leaks occurred. When unit tests were passed so that no known error were left, and no memory leaks occurred feature was merged to the master branch. After GUI was implemented the emphasis of testing shifted more to manual testing. Program was tested manually by testing different use cases via GUI and see how the program responded. Also, if there were no errors found that needed more investigation during manual testing, printing information to the command line was used to track down the source of the error.

5 Work Log

5.1 Week 1

During the first week, most of the time was spent on ideation and planning the project. Each member also set up the proper project development environment on their personal machines.

Mikael Laine: Focused on writing the project plan and took part in creating the UML-diagram for the program in collaboration with Joonas. Roughly 5 hours of work.

Niklas Pigg: Niklas also wrote a significant part of the project plan documentation. He also implemented barebones versions of the Tile and Map classes, and a simple file parser. Roughly 5 hours of work.

Joonas Savola: Joonas took part in creating the UML-diagram and writing a section of the project plan. He also looked up tutorials on the SFML library that is used on the project. 5 hours of work.

Lotta Ketoja: Lotta wrote a portion of the project plan as well as implemented a basic version of the Projectile-class Roughly 4 hours of work.

5.2 Week 2

During the second week, each member focused on implementing their agreed on basic classes. Mikael did the Enemy-class, Joonas implemented the Tower-class, Niklas did the Tile and Map classes and Lotta finished the Projectile-class.

Mikael Laine: Implemented the basic functionality of the Enemy-class and its subclasses and added a game clock to the Game-class, however this feature was later done away with. Roughly 4 hours of work

Niklas Pigg: Niklas continued development on the Tile and Map classes. Additionally, Niklas implemented the basic functionality of the Game-class as well as continued work on the file parsing system. Roughly 4 hours of work

Joonas Savola: Joonas implemented basic versions of the Tower-class and all its subclasses as well as some minor changes to the Tile and Map classes. Roughly 4 hours of work

Lotta Ketoja: Finished the implementation of the Projectile-class. 3 hours of work. This week Lotta also left the project.

5.3 Week 3

During the third week, the focus was on implementing a basic GUI for the games as well as further developing the classes introduced in the previous week. Mikael would have the responsibility of the GUI, and Joonas and Niklas would focus on the advanced methods of the basic classes.

Mikael Laine: Implemented a barebones version of the GUI without any functionality but drawing many of the game objects on a screen. Roughly 3 hours of work

Niklas Pigg: Continued work on the advanced features of the Map and Tile classes as well as implemented a basic functionality for dynamically creating the route of the enemies based on the map. Roughly 3 hours of work

Joonas Savola: Joonas implemented several functions to the Game-class, namely functions related to building, moving, and removing towers from the map as well as a basic function for tower targeting. Roughly 5 hours of work

5.4 Week 4

During the fourth week, Mikael continued work on the GUI. Niklas and Joonas continued work on the advanced methods of classes with more complex algorithms.

Mikael Laine: Updated the game GUI, now drawing the game properly and some additional features, such as showing the tower levels, as well as adding a sidebar with buttons to the game to control it. Roughly 5 hours of work.

Niklas Pigg: Finished the Map and Tile classes and tested them and finished the dynamic enemy route generation. Roughly 4 hours of work

Joonas Savola: Joonas implemented many advanced functions, such as the collision detection of projectiles, as well as game tick advancing. Roughly 5 hours of work.

5.5 Week 5

During the fifth week, we aimed to iron out many of the features we have implemented and get the game to a working state. Mikael's responsibility was to finish the game GUI, however Niklas had the responsibility of creating the main menu and map selection GUI features, as well as implementing the wave system. Joonas' work was focused on fixing, refactoring and making changes to existing classes to enable better performance and fix bugs.

Mikael Laine: Mikael implemented the final features missing from the game GUI related to the building, moving, removing and upgrading of towers as well as changed some of the other features to make them more intuitive. Mikael also added sounds to the game. Roughly 6 hours of work.

Niklas Pigg: Niklas implemented the main menu and map menu for the game, as well as the wave progression system for each map. Niklas also spent time optimizing some of the games functions which had become quite cumbersome. Niklas also added the missing implementation for the Freeze tower as well as fixing the Container enemy type. Roughly 8 hours of work.

Joonas Savola: Joonas many of the containers of the game, such as moving to use smart pointers. Joonas also spent time optimizing the game performance and refactoring several classes. Furthermore, Joonas implemented the basics of the Container enemy type's functionality which had been missing so far. Roughly 6 hours of work.

5.6 Week 6

During this week, we were satisfied on the game's functionality and focused on writing the documentation for the game and finishing the project. Everyone commented the code they had written and wrote parts of the document. Mikael wrote the work log and overview of the project. Joonas wrote the software structure and testing parts. Niklas wrote the compilation, building and user instructions.

Mikael Laine: Commented all his own code, namely the GUI-class as well as the Enemy-class and portions of the Game-class. Wrote the overview and the project log parts of this document. Roughly 4 hours of work.

Niklas Pigg: Niklas commented his own code and wrote the user instructions, and compilation and building instructions parts of this project document. Niklas also added the external libraries to the project folders. Roughly 4 hours of work.

Joonas Savola: Joonas commented their own code and wrote the software structure, and testing parts of this project file and created the diagrams for the software structure parts. Roughly 8 hours of work.