Project plan - Tower Defense

ELEC-A7151

Tower Defense Group 2:

Mimi Mokka, 652911

Anders Alho, 585075

Henry Pietilä, 597296

Jonna Mikkonen, 656289

Date July 17, 2020

Tower defense is a classic strategy video game, in which a player places towers on a game field so that enemies traversing through the field are eliminated before they reach the other end of their path. If the enemies reach the end of the map, the game is over. This plan describes our tower defence game which we will implement during 2020 summer course of Object Oriented Programming with C++ as a course project. We aim for a fully functioning tower defence game with a graphical user interface (GUI), and some additional features as discussed later in this plan.

Scope of the work

We will implement the basic features, including basic graphics, at least three different types of towers, and enemies, and a simple user interface.

As required, there will be at least three different types of towers: e.g. a basic tower which shoots enemies within its range, an ice tower which slows enemies down inside its area of effect and a bomb tower which shoots a bomb that damages several enemies inside bomb's explosion area.

There will also be different kinds of enemies whose properties are different. Some enemies will be stronger than others and require more hits to kill. Some enemy types will move faster and some will split into several smaller enemies when killed. In the future we might also implement enemies with different kinds of weaknesses and strengths against different kinds of towers.

The game is played through simple graphic user interface which shows information about the state of the game such as money, lives, score and the current game field with towers and enemies. User interface is also used to place towers, start rounds and show more specific information about enemies, towers and their upgrades and description about the game and how it is played.

Gameplay description

First the game is implemented so that the player buys and places towers on the game field. Then an enemy wave is initiated, and during the wave towers shoot enemies inside their range. Changes to the towers are not allowed during this phase. In the future we plan to make it possible to modify towers both in between and during waves of enemies.

The enemies will follow a single, non-branched path. If an enemy reaches the end of the path, a life is lost. If the player runs out of lives, the game is lost.

The game follows the simple logic of a regular tower defense game. Upon starting the game, the interface asks the player to start playing. In the beginning the player is given a fixed amount of money which can be used to buy towers along the path of the enemies. When the player is ready, they will start the first round, causing a wave of enemies to spawn at the beginning of the path. After the wave the same logic repeats with increasing difficulty until game is lost or all rounds are cleared. The player gets money for each enemy killed and loses lives if an enemy gets through the whole path. As mentioned before, the game is lost if the player has no lives left.

Additional features

From the start, the basic features will be implemented in such a way that make additional features straight-forward to add, requiring as little code refactoring as possible. When all the aforementioned basic features of the game are implemented, and when we have a working game, we plan to implement some additional features. We aim to implement the features listed below, and possibly more if possible.

- Non-hardcoded maps
- Upgradeable towers
- More different kinds of enemies and towers
- A list of high scores that is saved locally per map, with a username
- Different attack and defense types for both the towers and the enemies

Non-hardcoded maps can be implemented by implementing the basic, hardcoded maps and game field so that new maps can be loaded or generated pseudo-randomly. We also aim to implement upgradeable towers by giving upgraded towers some extra features and boost, for example by decreasing tower's reload speed and increasing range.

At first we plan to implement three different kinds of towers and enemies suggested in the project description. We will implement abstract base classes for both towers and enemies, which should make it easy to implement new derived classes with different attributes and features.

We also plan to implement a bullet class which describes the projectile a given tower shoots towards enemies. We will implement enemies in such way that they respond differently for different types of bullets. Finally we aim to implement a high score list and point calculation system for the game. Points are given by defeating enemies and by winning rounds.

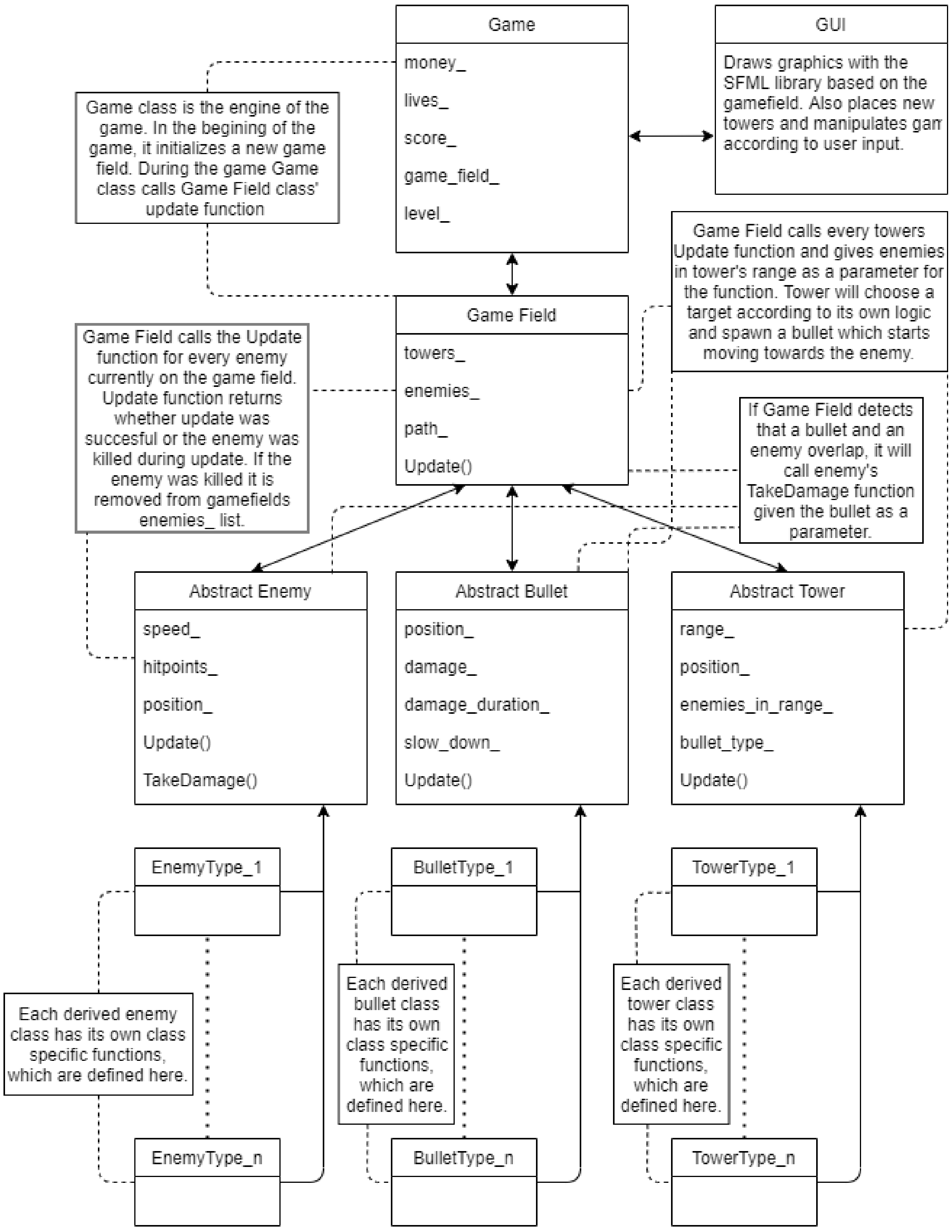
Game
money_
lives_
score_
game_field_
level_
GUI
Draws graphics with the SFML library based on the gamefield. Also places new towers and manipulates gam according to user input.
Game class is the engine of the game. In the begining of the game, it initializes a new game field. During the game Game class calls Game Field class' update function
Game Field
towers_
enemies_
path_
Update()
Game Field calls every towers Update function and gives enemies in tower's range as a parameter for the function. Tower will choose a target according to its own logic and spawn a bullet which starts moving towards the enemy.
Game Field calls the Update function for every enemy currently on the game field. Update function returns whether update was succesful or the enemy was killed during update. If the enemy was killed it is removed from gamefields enemies_ list.
If Game Field detects that a bullet and an enemy overlap, it will call enemy's TakeDamage function given the bullet as a parameter.
Abstract Enemy
speed_
hitpoints_
position_
Update()
TakeDamage()
Abstract Bullet
position_
damage_
damage_duration_
slow_down_
Update()
Abstract Tower
range_
position_
enemies_in_range_
bullet_type_
Update()
EnemyType_1
EnemyType_n
Each derived enemy class has its own class specific functions, which are defined here.
BulletType_1
BulletType_n
Each derived bullet class has its own class specific functions, which are defined here.
TowerType_1
TowerType_n
Each derived tower class has its own class specific functions, which are defined here.

Figure 1: Diagram of our main classes and how they communicate with each other

High-level structure of the software

Our current understanding of the main classes and their connections is depicted in Figure 1. The main classes are Game, Game Field, Abstract Tower, Abstract Enemy and Abstract Bullet.

Game is the interface that the player uses in order to play. It stores the user's money, lives and score. It is the engine which drives the game by updating the game field, and by keeping track of the current level. Game class also handles the the graphical user interface (GUI), which is generated with Simple and Fast Multimedia Library SFML, by giving it the needed parameters.

The Game Field stores the game field. It has lists of towers and enemies currently on the game field and regularly updates them. Game Field also stores the path of the enemies through the map.

We have an abstract enemy class that has the basic enemy attributes and functions. Having an abstract class also helps storing all different kinds of enemies in one list. Abstract enemy class has protected attributes speed_, hitpoints_, and position_, and their getter functions. The speed attribute tells how much an enemy moves on the game field on each tick. Hitpoints are the amount of damage an enemy can take before it dies. Position defines enemy's current location on the map.

Enemies also have functions for updating its position and taking damage. Enemies get the path by parameter when they are initialized. Update function checks that an enemy is still alive, meaning its hitpoints are greater than 0, and then moves the enemy along the path according to its speed. TakeDamage function takes a bullet as a parameter and decreases the hitpoints of the enemy or has some other effect on the enemy, for example slows down the speed of the enemy.

Different kinds of enemies are implemented in their own derived classes, which are presented in the picture 1 as EnemyType_1 and EnemyType_n. This means that we have n different derived enemy classes for different enemy types, which will be named according to the theme of the game and class's traits.

Different types of towers and ammunition are implemented similarly in their own derived classes of abstract Bullet and Tower classes.

Towers
different types:
1. A basic tower, shoots enemies within its range,
2. A slowing tower, slows down enemies inside its range,
3. A bomb tower, shoot a bomb when enemies in range, can kill multiple enemies

Also we will implement helper classes when the need for one occurs. At least we are going to implement a Position class that represents coordinates on the map. The class has for example function that calculates distance between two positions.

Libraries

We will use SFML graphics library because they seem beginner friendly yet sufficient for our needs. We believe we will not need any other libraries since according to our understanding SFML has a sound library as well. We will implement the sounds if we have time.

Division of work

We will use Trello to divide the work load and tasks between group members. We also have a Telegram chat for general discussion. We will meet regularly either in person or via video call (Google Hangouts) in order to discuss our progress and problems we face during the project. We will also meet during the summer in person to simply code together, to keep up motivation and lower the bar to ask for help.

We can roughly divide the work load in four categories: graphics and use of libraries, game and game field classes, abstract tower and its derived classes and abstract enemy and its derived classes. Once we have some rough drafts for these, we can work with the common features.

For the time being, we will divide the work load so that Anders will focus on the graphics, Henry will begin with the game and game field classes and Mimi and Jonna will start developing the tower and enemy classes respectively. We will keep tabs on each others' progress (e.g. Trello cards), and of course help each other in any way we can.

Initial schedule

Our first goal is to build a very basic graphical user interface to play the game, since then testing becomes easier than without one.

Our next goal is to have some basic, initial drafts for the enemy and tower classes. Then we will move on to implement the game field: paths where the enemies can move, and abstract enemy and multiple derived classes to test the basic graphics and enemy movement. Towers will be added when basic enemy functionality is achieved.

Once we have built the basic functionality in all aspects of the project, it will be easier to implement the so called 'missing pieces' and debug. Our aim is that all basic features have been implemented before the midterm meeting. Also we plan to write basic unit tests for the basic functionality at this point.

We will then reflect on our progress and feedback, and decide which and how many features we have time to implement. The initial weekly schedule is presented in the table 1. Week 1 means the week after project plan deadline (week 30).

Initial schedule	
Week 1	Make a blob on the screen with SFML
	Basic enemy and game field implementation
Week 2	Basic tower and bullet implementations
	Interaction between enemies, towers and game field
	Basic SMFL graphics (e.g. circles and squares) for tower, enemy and game field
Week 3	Advisory meeting with assistant
	Different types of towers and enemies
	Sprites for different towers and enemies
Week 4	Testing and finishing all the basic features
	Starting to implement additional features
Week 5	Implementing the additional features and choosing the final features to implement
	All additional features implemented at the end of the week
Week 6	Project demonstration and deadline
	Final bug fixes and documentation

Table 1: Schedule