1. Scope of the work

The tower defence genre is one that most people have come across at some point and which was really popular at the peak of Flash games up until Flash was eventually retired. There are a few games which have received critical acclaim and remained relevant even after Flash. One of such games is Bloons Tower Defence, which is currently at its sixth mainline release with several spinoffs across all platforms. This has been inspiration also to how we have designed some aspects of our game, but truth be told, most TD games are very similar at the core level:

In the beginning of the game, the player is given a certain amount of money to spend on towers and take on the first enemies. We have decided to award more money to the player every time they kill an enemy. One level goes on for five rounds/waves and we can add additional towers in between each round. When we reach the end of a round, the player will be paid an extra amount of money and they are able to place more towers.

The project will include at least three different types of towers and monsters. Towers can differ in the damage they inflict, the range they have, etc. And there are multiple enemy types that have different speeds and amount of hits they can take before dying.

Since we don’t know the workload and complexity of the mandatory features, we cannot say for sure which additional features we will be able to implement yet. However, the ability to place towers in the middle of a round is a feature we hope to add after the mandatory requirements are fulfilled. In addition, we hope to be able to offer upgradable towers, non-hardcoded maps, possibly a level editor and sound effects. Furthermore, additional enemy- and tower types are considered.

2. High-level structure

Game object

The game object is the universal object that handles the overall logic of the game that is not specific to a certain class. It will contain the update function that contains the game loop. Update function will call render function that will update what is being shown to the user. The game object will contain logic of the main menu and initialize a world when a level is selected. Game object will also track user’s inputs and call class’s functions based on them. The game object will contain all the universal variables about the game such as window size and game status.

Game object will have member functions such as:

- UpdateMousePositions (relative to the view)
- InitializeWorld
- InitializeEnemies
- UpdateEnemies
- UpdateTowers

- SpawnEnemy
- Render

Using these functions game objects creates the game loop that updates the status of the game based on user actions and rules of the game. We are not yet sure how the number of enemies and towers will be handled, but game object will track that the maximum number of enemies at given time is not exceeded. Initialization of world and enemies will call necessary member functions of those classes to properly start a level. Finally, render function will update the visuals being shown in the GUI.

Level class

We planned the level map to be a grid-based system. Level class is a representation of a single game level. Instance of a level class will include a list of all the tiles in the grid. Level class will also include level-specific information that it takes as parameters about the world that includes:

- Height (tiles)
- Width (tiles)
- Number of enemies to be spawned
- Type of enemies to be spawned
- Tiles in the world and their order
- Length of a round (timer)

Height and width are taken as type Int, and they indicate how many tiles a level has. The number of enemies should indicate how many enemies should be spawned in a level for each round. This can be taken as a vector of Ints and each Int can indicate how many enemies a certain round will spawn. Type of enemies should contain the information about what type of enemies should be spawned. By taking all the information above as a parameter for constructor allows us to create new levels from a file that has the level details in correct format.

Abstract tile class

Each individual tile that the world is consisting of is implemented as an instance of a tile class. Tile class itself is an abstract class that contains logic that universal to all tiles. Tile may be occupied by a tower or an enemy. Tile also knows it’s position in the world. Some planned methods for tile class are:

- EmptyTile
- ReturnStatus

Abstract tile class will have many child classes that represent different types of tiles a world can have. Each of these will have some unique methods that they require to function as intended. Some of these child classes include:

- Road
- Grass
- Entry point
- Exit point

Road will represent the path for enemies to move along. Road class will have methods to implement simple path finding. Road class will have two constructor, one that takes exit point as parameter and one that takes another road as parameter (along with its position in the world). Road class will then store a value of what's the "next" road tile from itself, so it "knows" where to move the enemies. The road will be constructed from end to beginning in this way. Respectively entry point will take the first road tile as a parameter for its constructor and exit point takes only its position as parameter. Entry point will also have method for spawning enemies and exit point will have method for making the player lose the game if enemy reaches this point.

Abstract tower class

We concluded that the most logical way implement towers is to have an abstract class which all the specific towers inherit. The common attributes that every tower has include:

- Damage
- Range
- Attack speed
- Value
- Upgrade price
- Level

The damage, range and attack speed should all be relatively straightforward. Damage could be the amount of damage points dealt per attack. Therefore the damage output would be a function of damage and attack speed. Every tower would also have a range (how far they can detect enemies and attack). We are not completely sure yet how we are going to implement this but some preliminary ideas are discussed later on. We are also planning to implement upgradeable towers so for that the towers will have to have a level attribute (to cap the upgrades to a for example level 5) and also a upgrade price which should increase with each level. A possible attribute to also add is a value of the tower, which should reflect the price and level of the tower. Therefore, we could implement a selling function where the player can sell the tower and get at least some of the money refunded to their wallet/bank.

Obviously, the tower should have also some member functions and methods. Among those could be:

- Constructor
- Destructor
- Upgrade
- Refund
- GetDamage
- GetRange
- GetAttackSpeed
- GetLevel

Constuctor is an obvious must for all towers as it will instantiate the object and give predefined values to the attributes such as damage, range, attack speed etc. Destructor is also a good idea to have (although we're not sure if necessary in case the object doesn't have any dynamic memory that needs to be freed). Upgrade could be a void function which is called when the player purchases an upgrade to the tower. It increases the value of the tower and then the damage, range or attack speed or whatever we decide the upgrade to do.

Since we are not yet entirely sure how the interaction between towers and enemies will be handled, we proposed functions GetDamage, GetAttackSpeed and GetRange. This way some sort of external code, such as the aforementioned game object could pull the relevant attributes from the tower object and communicate to the enemy object when and how much damage should be dealt.

Different tower types could be developed by playing with the range, damage and attack speed attributes. The towers could include:

- A regular machine gun tower, dealing medium damage to a single enemy in quick succession over a medium range and could switch from target to target quickly for it is relatively fast fire rate.
- A sniper tower which can deal huge damage to a single enemy from a great distance but which would be punished/balanced with a low fire rate.
- A grenade or flame tower, which would be an “area of effect” (AOE) tower, being able to damage several enemies at once but will be balanced with a low range.

All of those towers should be upgradeable which should introduce a new dynamic to the game, making players choose between either buying more towers or upgrading the current ones for better stats.

Abstract Enemy class

We’ll have an abstract enemy class to implement different kinds of enemies. All enemies have hit points and speed that depend on the type of enemy in question. These work as one would think; Hit points tell the toughness/health of the enemy and speed tells how fast the enemy moves through the map. Enemies could also have value, which tells how much money the player gets when this enemy is destroyed, (but this info can also be stored in the game object).

Methods for Enemy class:

- Constructor
- Destructor
- TakeDamage
- GetHP
- GetSpeed
- GetValue

Constructor and destructor might not necessarily even need explicit implementation. TakeDamage is called when a tower hits an enemy and it shall take damage as an integer and lower enemy’s hp by the amount. GetHP and GetSpeed are just a way to access the HP and speed values from outside the class. GetValue works the same way but returns the value of the enemy (which should be awarded to the player’s funds upon the enemy’s death.)

Ideas for types of enemies:

- A “basic” enemy which has normal speed, medium to low HP. The easiest type of enemy who will populate the first waves.

- A fast moving enemy who moves significantly faster than the basic enemy but has at the same time much lower HP. Can be tricky to tackle when spawning in masses/hordes.
- A slow enemy with a lot of HP. A sort of “tank” type enemy who might slip through defences when turrets are busy engaging other faster moving targets.
- Enemy that splits into multiple others. This would be a nice feature to implement if we can figure out how to.
- Camouflage enemy. Could be visible to for example only a certain type of tower as long as it is at full HP. After that other towers could see and engage it as well. This would incentivize the player to use different towers not just one type with upgrades.
- Some enemy that heals other enemies. This could be interesting to implement.

Common attributes of all the enemies:

- Hit points
- Speed
- Value

Shop class

(Shop class is utilized by game class or integrated inside of it without a separate class.)

Each level has a shop, where the player gets their towers from. Shop class contains all the defined tower types with their corresponding prices and maybe some info on what each of the towers does. If you have enough money you can click on the tower and drag them to the playing area creating a new tower. Shop class will keep track of all the tower types available in the shop.

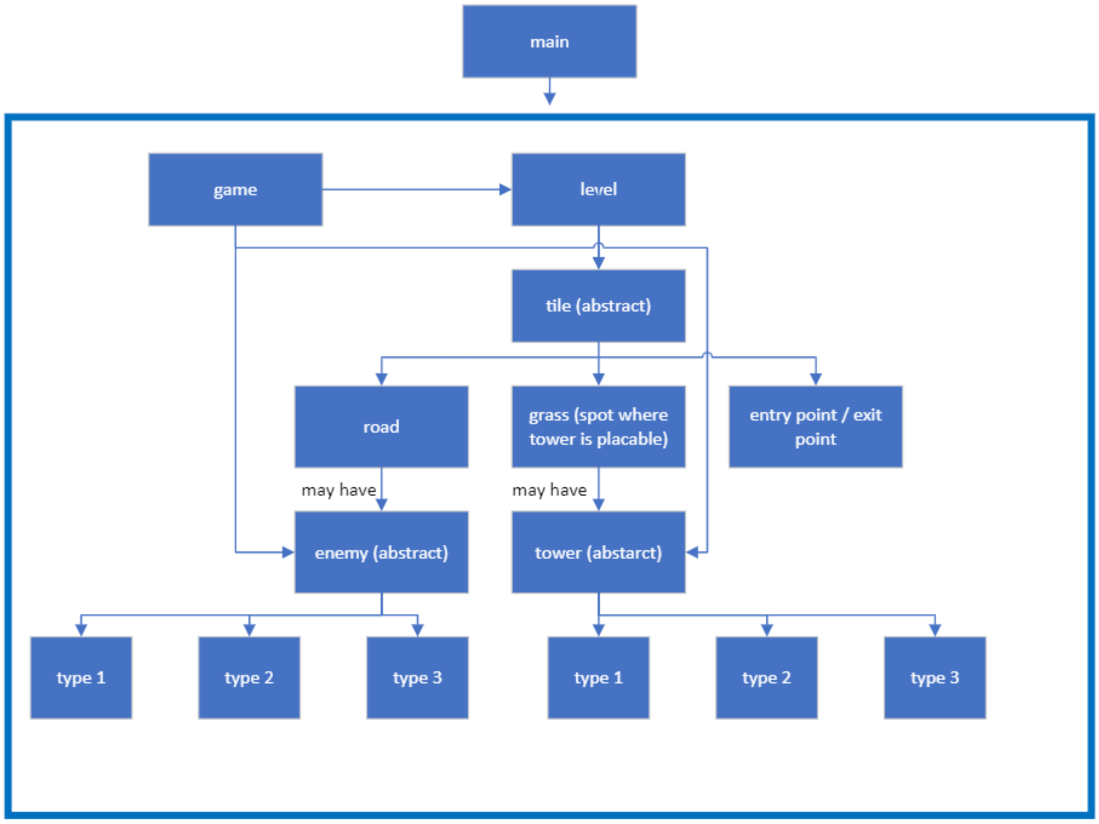

Figure 1. Class diagram

3. Libraries

We are planning on using the SFML library to handle game graphics and GUI. SMFL also has tools to track user inputs and other events (like closing the window). In addition, SFML should support the addition of sound effects and music which we intend to implement as a possible extra feature. We will also obviously use the C++ standard library while creating classes and member functions.

(Other libraries like box2D/Qt can be considered if it seems that certain actions or mechanics cannot be implemented purely with standard libraries and SFML.)

4. Division of work

We realize that at this stage in the development it is very hard to divide strict roles. The reason is that none of us are domain experts in the field of game development and it is hard to predict how much time and effort any specific task will take. As a result of this it is difficult to divide the roles between people now while also ensuring an equal workload for everyone.

How we have tried to approach this problem is identifying the core tasks that have to be done to get the product working and then tried to assign responsible people for those. Now again, we realize that this might not reflect the full extent of work that needs to be done and also that the workload for different tasks can be wildly different. Therefore, we will try to work as a team, share responsibilities dynamically and assist each other in our problems.

• Towers	Karl
• Enemies	Lauri
• Game class:	
○ GUI/Render view	Roosa
○ Game loop	Severi
• Level and tile class	Severi
• Level design/balancing	Karl
• Testing (unit testing/integration testing)	Everyone

Testing is something that everyone will probably take part in to some extent (for example unit testing their own code). Then the code will also have to be integrated and tested for how the modules work together.

5. Schedule

The schedule is rough and flexible so if things take more time than envisioned it can be pushed back.

- Plan submission deadline 11.11.
- Some sort of interface or interactable UI and tower/enemy prototypes ready during next week (wk46)
- Get some sort of a basic game loop running (wk47)
- Start integrating all of the individual units (wk48)
- The basic game ready with the mandatory features and without major bugs ready by 1.12.
- Do the demo to advisor as soon as possible (wk49) and reserve extra time for bug fixes and extra features
- Final commit deadline 9.12.