Project plan: Tower Defense

ELEC-A7151 Object oriented programming with C++

Group #3:
Oskar Tainio 590138
Katariina Tuovinen 561507
Arsi Ikäheimonen 47130M
Esa Palosaari 618573

Introduction and general outline

Our group's object is to create a simple Tower defense game based on C++ programming language and SFML graphics framework. Due to the abundance of possible features provided, the project is divided into three different phases. In the first phase we will implement the alpha version of the final game with simple grid based map and limited range of features. This will provide us a core working base for the beta version (second phase), where additional advanced features will be implemented. In the final phase the game will be finished and refined to match the requirements found in the evaluation matrix.

Description or use case of the game

When the program is executed, main function will open up a title screen, which has the game's title in it. User can then close the window simply just by clicking it or pressing any button. After the title window is closed, a same sized window will open in place of title window. This is done by class Engine's start() function. Engine will create a map, which is loaded from user's own database. In the game screen user can see the playable map and user interface screen, where the stock of user input happens. UI-screen contains buttons to buy/sell towers, infographics screen and menu options.

Towers can be bought by clicking the tower icon and then dragging and dropping the object. Towers can't be built on roads. Tower's statistics (such as range and damage) can be seen when user clicks the placed tower. Tower's range is also drawn as a circle around the tower. Towers fire different kinds of projectiles based on their type. Projectiles include types like bullets, rockets and lasers. Enemies will lose health points based on each projectile's statistics.

Enemies spawn only when a new level starts. There can only be one type of enemies in a single wave. User cannot know the type of the wave beforehand. Special levels can be added when the core game is done and refined. There is always a limited time between each wave, but the user can start a new level by pressing the start button. Player can also save the game before a new level starts. Saved game contains current map information and user's own statistics such as money, score and level.

If players homebase reaches zero healthpoints, the game will end. After the homebase explodes, the endgame title will popup and ask the user whether he or she wishes to quit the game or load the latest saved game.
A rough sketch of user interface is demonstrated at the end of this document (appendix 1.1)

Major architectural decisions

We plan to use something like the Model-View-Presenter (MVP) programming model which is a common graphical user interface architectural pattern (https://martinfowler.com/eaaDev/uiArchs.html; https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter). We separate the data model of the game from the graphical view by building them in separate classes. This has at least two advantages: (1) the data model and the graphics of the game can be built more or less separately which speeds up the development when there are several

people working on the project, and (2) the separation of the data model and the graphics makes it possible to change, debug or upgrade one part of the game without it affecting the other which is useful when we plan to build several iterations of the game. Separating the data from the view may help also for example in saving and loading the data on and from the disk. If there would be plans to make the game a multiplayer or an online one, it would be helpful to have just one data which could provide different views to multiple users.

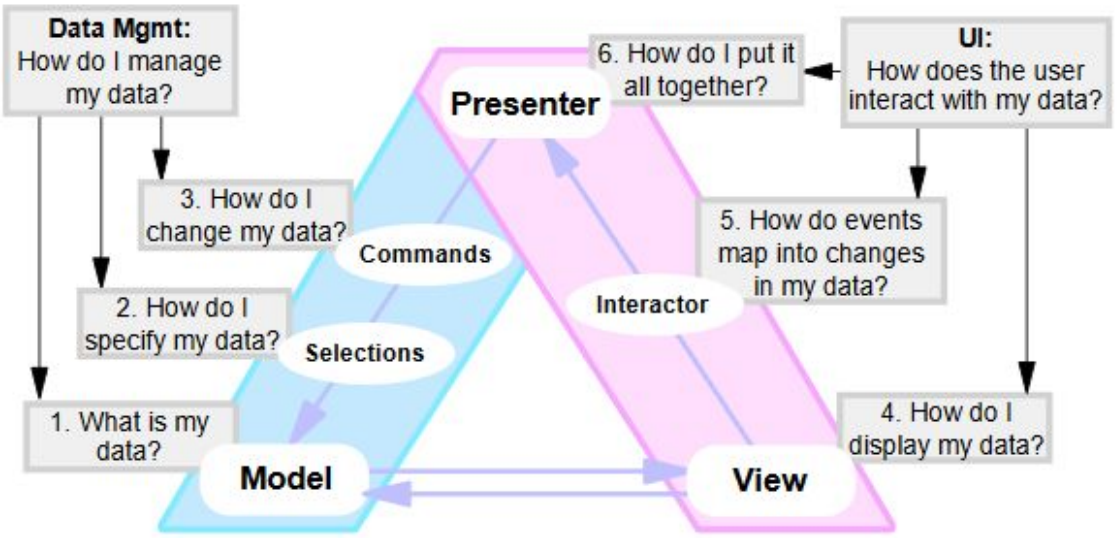


Figure X. The separation of data and user interface in MVP (http://www.wildcrest.com/Potel/Portfolio/mvp.pdf)

In contrast to the MVP architecture, our plan does not have a presenter class which would handle all of the business logic of the game. Instead, the Engine class has most of the tasks of a presenter and acts on both the data model and the view which it updates, but the main function, TitleWindow, and UserGraphics also take some presenter roles. Especially the UserGraphics class has the task of handling user inputs. This has the advantage of keeping the control logic close to the data (data model in the Engine when updating the data, the screen coordinates in the UserGraphics when listening and responding to mouse inputs). There could be problems with having all of the control logic working well together when it is separated in different classes. Special care should probably be given to how the data model and the view are updated when many separate classes with their own control logic can simultaneously affect the updating process.

Design rationale

Program's functional architecture is based on collaboration of eight different classes; TitleWindow, Engine, UserGraphics, TileMap, Enemy, Tower, Projectile and HomeBase. The UML diagram showed in the end of the document represents the relations between these classes. Closely cooperating classes are linked with arrows (such as Engine manipulates TileMap, which can alter UserGraphics). The TitleWindow-class is a separate body, which basically contains a very small amount of functions that do not collaborate with the rest of the program (start() to open window, manageEvents() to manage user input and

render() to handle frames). Given time possible main menu options will be implemented inside the TitleWindow-class.

The Engine class is the main powerhouse of the program. It starts the game, processes frames with update() function and contains all the information of existing towers, enemies, healthpoints etc. Main features, such as saving and loading the game, are also implemented in the Engine class. The class has the following private attributes: enemies, towers, projectiles and homebase. For example, the number of towers are stored in a vector. Changes in private information are transferred to TileMap class.
The TileMap class is called from the Engine class and updated at every certain interval of time (frames). The TileMap class holds the information about the map in vertex arrays which create the 2D game grid. Different values (such as 1 for grass tile, 0 for road tiles and 3 for enemies) in the vertex arrays indicate different objects in the grid which are then later drawn in the UserGraphics class. This principle is presented in the SFML's tutorial for drawing tile maps.

The UserGraphics class is called from the TileMap class and it draws the TileMap according to the values in the vertex arrays. It also takes input from the user and depending on the input it handles the different events and calls the Engine class to update game's state again. Engine class then modifies the TileMap according to player's input.
The UserGraphics class has the following functions: Start, drawMovingTower, placeTower, handleEvents, handlePause, handleEnd and render. The class also has the following private parameters: frameTime, currentTime, Textures, Sprites, Texts and Shapes which are all part of the SFML's Graphic module. The main usage of the SFML's Graphic module is located here.

Classes Enemy, Tower, Projectile and HomeBase contain specific functions and information needed for Engine's update() function to compute properly. The Enemy class represents the enemies attacking the home base. Enemies are designed to appear at the start of the level and their type is chosen randomly. Alpha-stage enemies can only move within the path-tiles and can only damage homebase.The class has following functions: move, getHit, EShoot, checkDistance and hit. The Enemy class also has the following private attributes: NPC, HP, moveRate, DMG and Placement.

The Tower class represents immovable tower objects in the game. The towers are designed to be stationary and shoot towards the enemies, each tower in its own firerate and projectile. The Tower class is also called from the Engine class and has the following functions: checkRange, shoot, getHit and checkAngle. The class has the following private attributes: NPC, range, firerate, Placement.

The Projectile class represents the projectiles shot from the towers. The projectiles move to the direction where the tower has ordered them. The Projectile class is called from the Engine class and has the following functions: move, hit and checkGoal. It also has the following private attributes: NPC, DMG and Placement.

The last class is the HomeBase class which represents the home base of the game. The home base is to be protected, but doesn't have any own functions or features to do so. The HomeBase class has only the following functions: getHit and explode, the latter of which is called from the Engine class when the game is over. The HomeBase has following private attributes: HP and placement.
A rough sketch of class hierarchy can be found at the end of this document (appendix 1.2)

Databases and algorithms

Program does not rely heavily on databases. Data needed for creating maps are stored as text-files. These files contain information about each TileMap as number indicating it's type. Also, Player high scores are stored in text file. Information about enemies, towers, projectiles are stored in corresponding objects and updated as game progress. Objects are stored in vectors

Algorithms needed to run the program contains PathFinding, Projectile, ChooseTarget and NewLevel. Algorithms are implemented as functions in classes.
PathFinding is used by enemy objects and it utilizes information saved in Path list. Path list is a linked list, which contains information about path leading to the Homebase.
Algorithm selects enemy path by simply checking the next coordinate values and corresponding MapTile in Path list.

Projectile algorithm calculates angle between Tower and Enemy objects. Angle utillizes coordinate information and calculates a vector starting from tower and ending at enemy. This vector if used to get angle of fire for projectiles.

ChooseTarget algorithm determines nearest Enemy object for every tower. Algorithm utilizes coordinate information and calculates Euclidean distance between tower and enemy. Distances are stored in vector and enemy with lowest value is selected as target. Tower shoots enemy if minimum range is smaller than Tower range.
NewLevel algorithm is used to generate increasingly harder enemy waves for each new level. Number of Enemy objects, Enemy HP and moveRate increase as game progress.

Goals and requirements

Due to the fact that we divide our project to three different phases, our goals are also robustly divided to three according difficulty levels. The first stage contains all the requisites needed to complete the project implemented in the evaluation matrix and TIM-homepage of the course. This consists following basic features:

User interface:

- Mouse movement and interaction with map (clicking and dragging objects)
- Three game statuses: Menu, Ingame and pause
- Drawing map, Tower shop-panel and Information panel, Menu- and options panel
- Readable values, such as money, game level, score, object statistics.
- Title-state of the game, which opens when program starts and closes as user clicks it (starting the actual game)

Graphics:

- SFML-based textures for different classes
- SFML TileMap-class for map manipulation
- basic visual textures, such as simple shapes for each drawable object

Towers:

- Three major type towers with different damage, fire rate and range
- Towers can be destroyed and bought (latter based on user´s money)

Enemies:

- Three different types of enemies
- Only one type of enemies will spawn in each wave
- Road-based pathfinding algorithm

Non-hardcoded map:

- Map is read from user's own documents and passed into working TileMap-class
  Ingame map can also be saved during paused game and loaded if user wishes so

After these features have been implemented, the program is at the Alpha-stage of the game. This is also the first major goal of our project. The second phase consist of refining the program and adding as many additional features as possible to the base game (based on our skill level and time). At least the following features will be implemented:

Advanced graphics:

- Refined textures for each drawable object
- Different animations for moving objects within map
- Event-based sounds are implemented

Towers:

- Towers can be upgraded and repaired. Upgraded towers receive boosts to their statistics.
- Towers can be attacked and destroyed
- Towers are given special abilities if upgraded to max level.

Enemies:

- Enemies have two major types: Ground and Air. Ground-based enemies follow the road and Air-based enemies can fly over towers straight to the homebase.
- Enemies can damage towers
- Boss waves are implemented. Bosses have significantly more HP and deal more damage.

Gameplay:

- Main menu is created with options such as new game, load game and highscores
- Locally saved highscores are created and they will be saved in the same way as maps
- Homebase has also some upgrade-options, which will give the game some more flavor.

After these features have been implemented, the program is at the Beta-stage of the game. This is the second goal of our project. The third and final phase consist of plugging possible memory leaks and reorganizing code to a more reasonable state. Third goal will be achieved

when the program is refined to the final stage and is ready to be submitted to evaluation. No new features will be implemented during this phase.

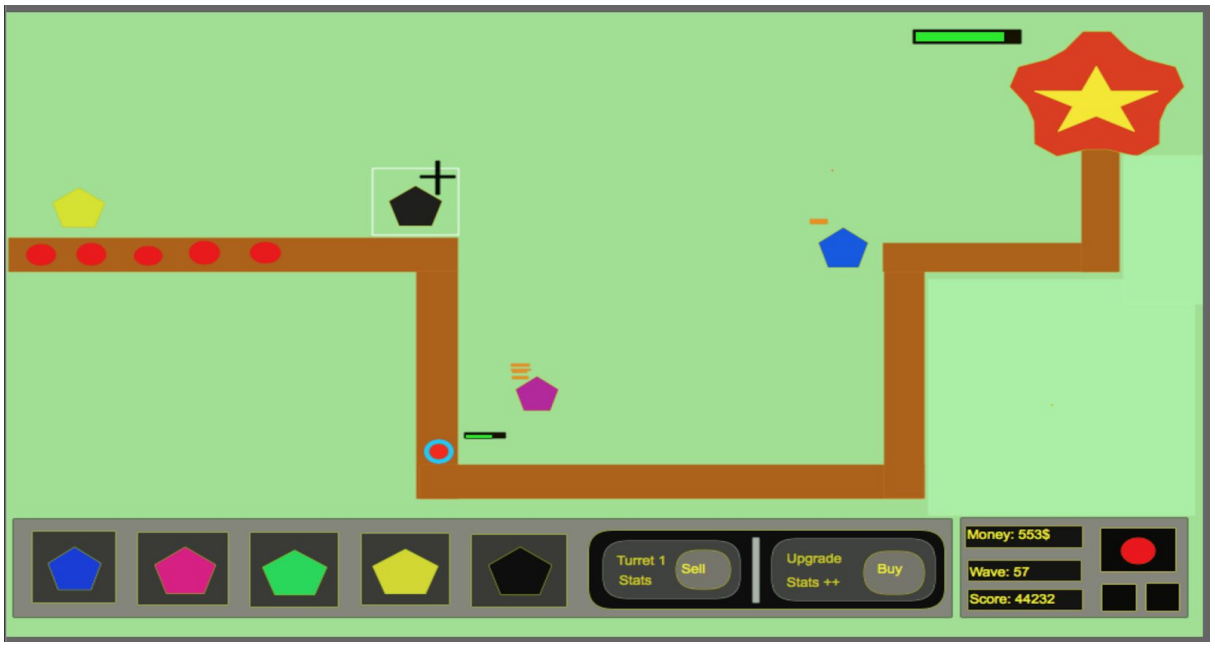
Preliminary schedule

Week 44	Project planning Getting acquainted with SFML-library
Week 45	Project plan finished and submitted to Git, Implementation of core features Getting acquainted with SFML
Week 46	Implementation of core features done, syncing class structures together. 18.11. game at Alpha-stage. Meeting with group advisor.
Week 47	Mid-term meeting. Implementation of additional features, syncing additional features to the base-game.
Week 48	Implementation of additional features, syncing additional features to the base-game. 2.12. game is at Beta-stage.
Week 49	Refining existing program, 9.12. game is at the final stage and ready to be submitted to evaluation.
Week 50	Demo-week and project documentation submitted.

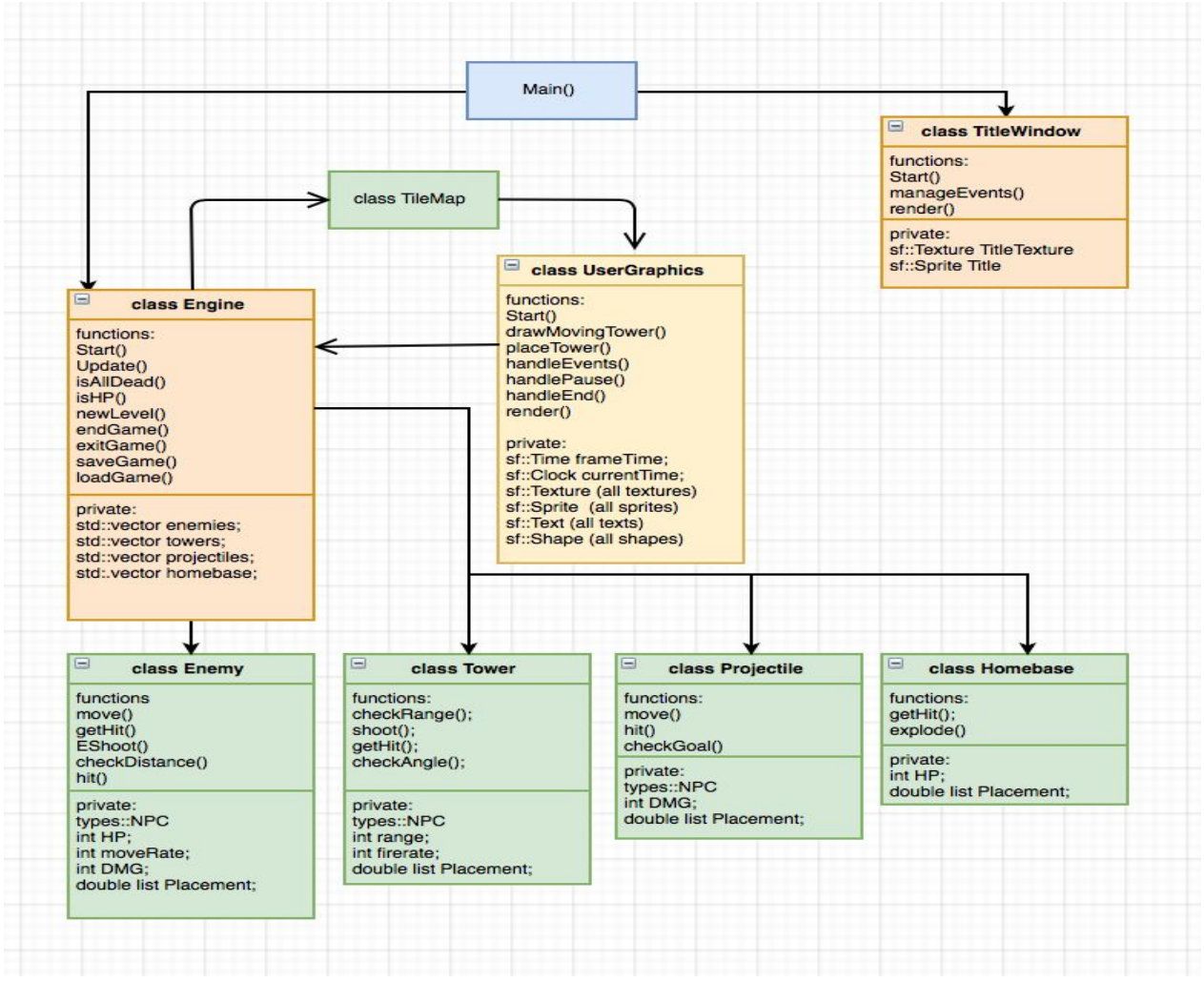
Distribution of roles and work

Workload is distributed among group members weekly in the following way: 12 hours of self-study and work, 4 hours of working in group in exercise-sessions, 1 hour of group meetings and 1 hour of experiment-based documenting. In addition there will be 5 hours of documentation reserved for the final program for each member. Total workload for each group member will therefore be a total of 113 hours for the duration of whole project. Distribution of roles is based on each programmers skills in the following way:

Oskar Tainio: Project management, Main, TitleWindow, Engine, UserGraphics
Katariina Tuovinen: Engine, Homebase, UserGraphics
Arsi Ikäheimonen: Towers, Enemies, Projectiles, UserGraphics
Esa Palosaari: TileMap, UserGraphics



1.1 Rough sketch of the game's user interface.



1.2 Rough sketch of class hierarchy and collaboration.