Project plan

Real time strategy game

Group info

Justus Ojala 770084
Otto Simola 653800
Zakhar Burda 791801
Joona Repo 656881

Overview

The goal of the project is to make a real time strategy game in which the player controls different units and buildings and uses those to either destroy the enemy or to control strategic points on the battlefield.

The army consists of different types of player controlled units. These units include builders, soldiers and vehicles. An optional hero-unit can be enabled in the settings of the game.

Builders are the base unit in the game. The game starts with these units which are used to collect resources like stone or uranium from the map. These resources are then used to build structures, which enable the player to purchase more advanced units. There are also defensive structures like turrets and anti-air guns. The units are purchased with money, which is obtained by destroying enemy units and structures. Money also accumulates over time. The units have different abilities according to their type.

The game's battlefield consists of different kinds of obstacles; broken buildings, mountains, rivers, forests, walls etc. In the beginning of the game the player's units spawn on the edges of the map in the designated spawn points.

The end-goal of the game is to destroy all of the opponent's units or to destroy their command center (Total annihilation). There are also other optional game modes like conquest or time battle where the players fight for different objectives or with a time limit. Customisable games with variable parameters like time, points needed to win and maximum army size are also an option.

Use case description

The player starts the game, which opens up the main menu. In the options tab the rules of the upcoming match are chosen — for example whether the match has a time limit or not — after which the player starts a match.

In the beginning of a new match the player gets some resource collecting units (Builders) and a command center. The builder units must be sent somewhere to collect resources, so the player selects them and clicks the gather button to order them to collect resources from a nearby resource pile.

After ordering the builders to gather resources the player is free to buy more builders from the command center by clicking the center and selecting the unit from the building menu. With more builders the player decides to build more structures, for example defence towers and a more advanced unit structure.

With the new building the player buys his first attacking units, which can be send to battle by selecting them and right-clicking the target. When the attacking units face enemy units, the player again selects their units and attacks the enemy by right-clicking the enemy units. After a while the players units have destroyed the enemy units.

The player continues this cycle of obtaining more resources and units for a few minutes while defending against the enemy's armies. When the player has a large army, they decide to make a final assault on the enemy. After a long, hard battle, the player destroys the enemy command center and wins. A victory menu pops in front of them, which tells them that they won and asks what they want to do. They select to go back to the menu and are returned back to the main menu.

User interface

The program will have a graphical user interface. A user must click on a unit they own to control it, then the user will see different action options to choose from. The chosen action and its result will be seen in an output window. Unit movement from location to location will be animated. With the graphical user interface, the player will be able to play without writing any commands directly but the game will include hot-keys for some actions.

The in-game interface has a battlefield and the players info bars. The info bars show the player data about the current situation of the game, for example the player's current resources (stone / uranium / money) and unit amounts. In the UI the player can also see a minimized version of the battlefield with the area the player is currently looking at highlighted (Minimap).

When selecting a unit the player is given multiple options to control the unit for example select target to attack, select point to move etc. When a builder unit is selected the player can order the builder to build structures by pressing the structure button, after which a preview model of

the structure appears, and choosing the build location by clicking a point on the map. If the structure is too large and can't fit to the chosen area the preview will turn red.

Different kinds of user interfaces in RTS games (Company of heroes 2 left, Command & Conquer right)

http://nyerguds.arsaneus-design.com/cncimg/img_archive/beta/desert01.jpg
https://www.coh2.org/file/12118/20160715204210-1.jpg

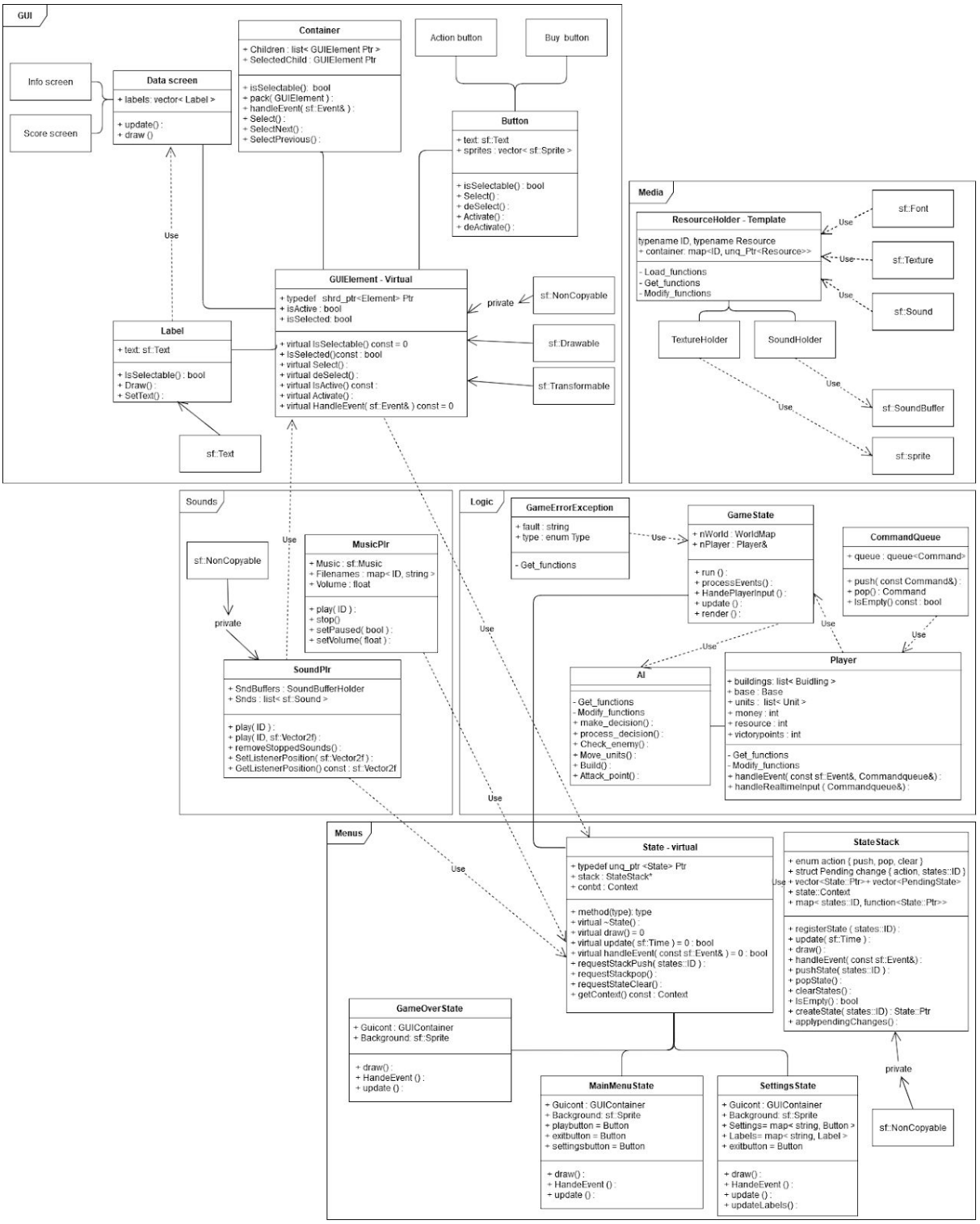
The game also has main menus for settings where the player can choose the game mode and change different options to manage the game rules.

Structural plan

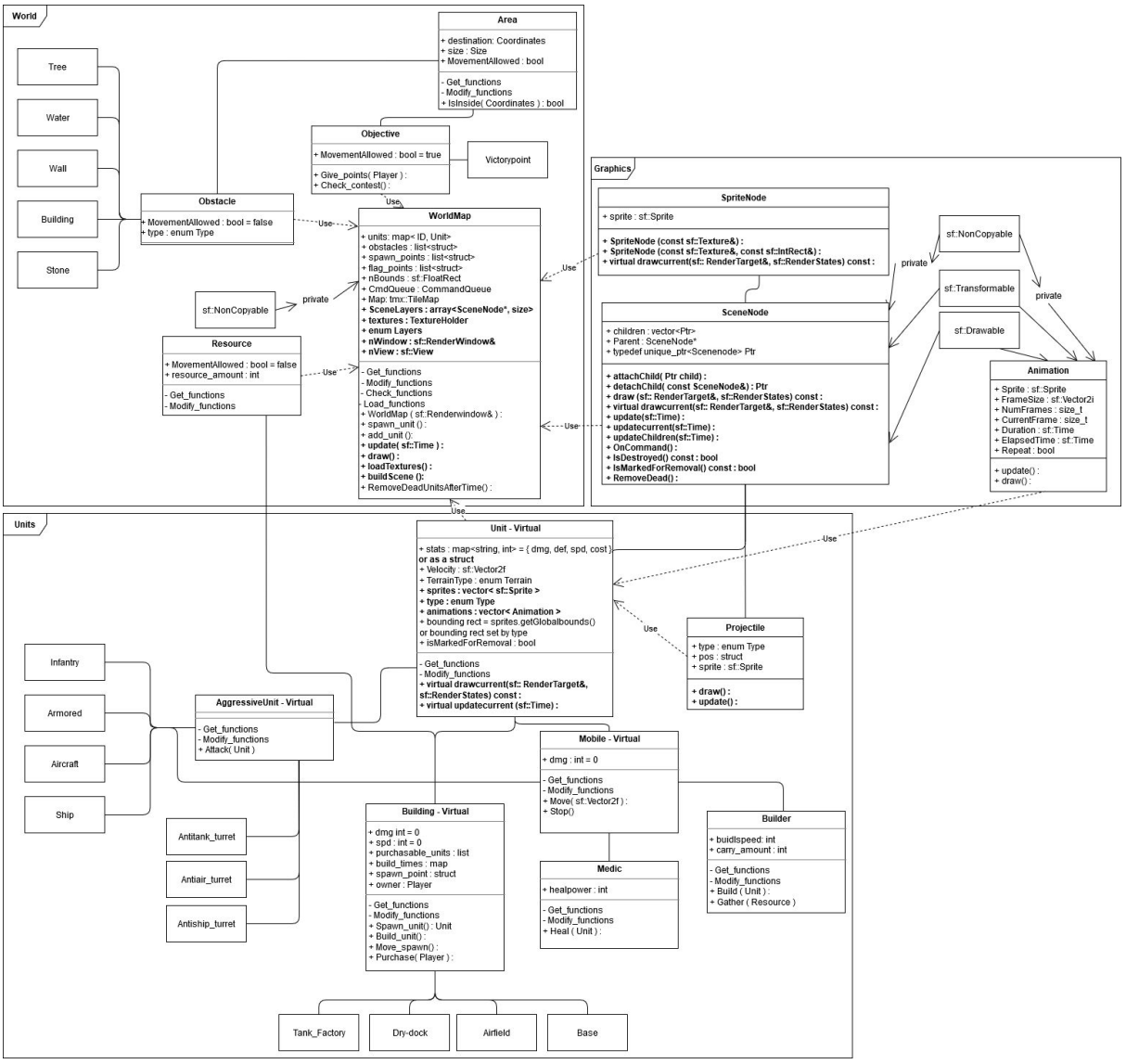
The structure of the program will use some concepts introduced in the book: **SFML Game Development: Learn how to use SFML 2.0 to develop your own feature-packed game; Haller Jan**, in addition a lot of the structure will be designed by the project team. We will design the structure to be as general as possible so that we have leeway when we start to design the theme and graphics of the game.

UML-Graph

The UML-Graph is divided into halves so that it could fit the document with decent resolution. The whole graph will be in the plan folder as a PNG.



The first part of the UML



The second part of the UML

Units

There is one parent class for all units, the abstract class Unit, which contains most of the functions and values needed by all units. It defines the unit's statistics (HP, damage etc.), as well as its location, velocity and bounding box. It also contains information about the unit's sprite and, for shooting units, the projectiles.

The Unit class is extended by two abstract classes describing certain traits of the unit; Aggressive and Mobile. The four general unit types (still abstract) then extend these based on what they need. Aggressive units are both aggressive and mobile, so they inherit both. Defensive buildings and passive units only fulfill one of these conditions, and therefore inherit aggressive and mobile respectively. Building is neither mobile nor aggressive, so they inherit the Unit class directly. Each of these adds information material to this type of class;

for example an Attack function for Aggressive and mobility information for Mobile. This way we avoid defining mobility and aggression multiple times.

World map

The WorldMap class is responsible for doing most of the management map elements and units. The map will be loaded from a premade file which tells the program which texture to use where. After loading the map and its textures the WorldMap class will also create the SceneNode tree that holds the units.

The map will be constructed with an external program “Tiled” and loaded onto the game with a pre-made C++ framework listed on the “Tiled” documentation. Some time may be spent on porting the map for our purposes. “Tiled” uses its own filetype called TMX which we can load into our game using the additional framework.

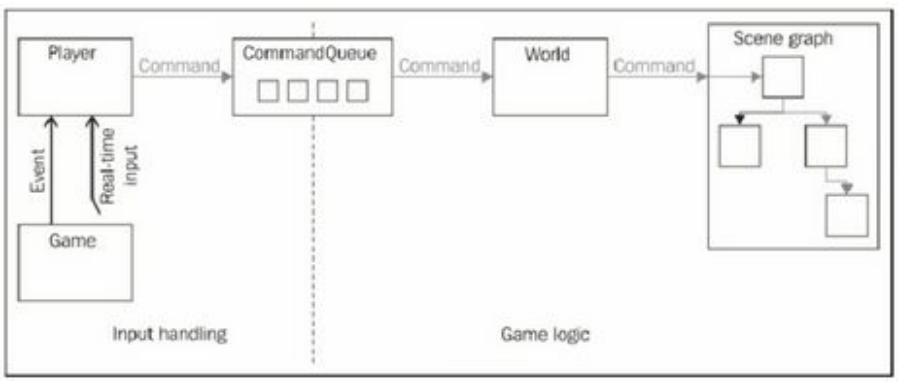
The WorldMap class also goes through and executes the commands that have been queued into the CommandQueue. In addition it is also responsible for updating the units and map elements when an interaction has occurred. This updating also includes the destruction of the units and elements.

Resource Management

We will be doing resource management with the use of “ResourceHolder” class which is a template class that will be used to create holder classes for different use purposes (mainly for textures and sounds). These holder classes will have all the heavy files like textures and sounds loaded and stored as a map. Because the files will be paired with an ID, fetching the right texture or sound when we need them in the different parts of the program will be quick and simple. The fetched files will be sf::SoundBuffer and sf::Sprite classes as the original sf::Texture or sf::Sound classes are really heavy to use. Using the lighter Sound Buffer and Sprite classes will make the program take less memory and run smoother.

Gameflow

The GameState is the main component of the game which processes the ticks (run cycles) of the program. During a tick the GameState first processes and handles the player's inputs in the processEvents/handleEvents functions (Key presses and mouse presses). If there are any changes to the game logic’s objects after handling the inputs, the changes will be turned into commands and will be sent to the commandqueue for later processing.



The picture shows the flow of the command system. Pictures and the idea from the book: **SFML Game Development: Learn how to use SFML 2.0 to develop your own feature-packed game by Haller Jan**

After this, the GameState will use the update function to update the game logic since the last tick happened, things will move, take damage and do all kinds of interactions. Here the WorldMap class will use the Scene graph to force all the objects to update.

After updating the WorldMap class will also take care of the cleaning (when it's needed). This is also the point when the CommandQueue is gone through and all the effects of the commands will be updated to the game logic. After updating it is time to show the player the changes we have made (aka. render the view). Here the GameState calls the Scene graph (through the WorldMap class) to draw the sprites of the SceneNodes. We will load the heavy resources before the start of the match, because we want the game to use the lighter resource classes through the handlers.

AI - Artificial intelligence

AI in the game will be implemented in this way, that each AI's unit will continuously do some tasks. For example Builder robots will continuously search for a good place for a facility and build it. With getting more resources, AI will build facilities and create new units. Each attacking robot will defend one building. When AI runs out of buildings, its attacking robots will explore a map in groups in order to find opponent's units and facilities to destroy. When they find some, they will attack the nearest target until it is destroyed. When AI's unit is destroyed itself, other attacking units will start exploring a map from the place, where their comrade was destroyed.

The class AI will be responsible to execute the AI commands to the units. The AI class will have multiple algorithm functions to do its decision making. These decisions will be made for each group of AI units, so that one of the units in the group will act as a leader.

GUI - Graphical User interface

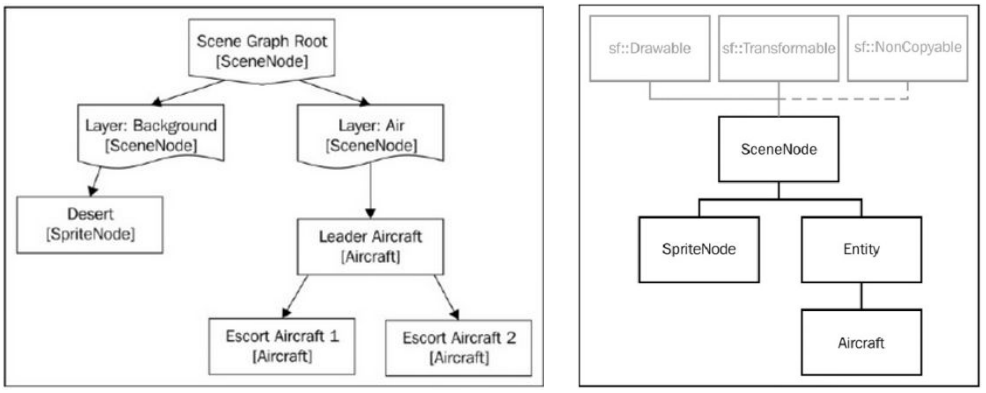
The base class of the GUI is the GUIElement class. It is used to make different kinds of GUI components like for instance buttons, data screens or labels. These components again are grouped together with the container class which holds and manages the elements.

States

To make various menus and screenviews we use a simple finite-state machine where screens like the main menu and the settings are defined as states of the program. The finite-state machine consists of the StateStack class which manages the States and the State class and its children which are the states of the machine as its name indicates. Using the StateStack we can switch between the states and make the user experience simple. The gameplay itself is implemented in the GameState.

Rendering

The rendering is done with the Scene graph which is stored inside the WorldMap. The Scene graph is a tree where empty SceneNodes start different layers like Background, Air, ground and water. This Tree data structure allows us to easily group the desired entities together and call them to update and render. After giving the call to update or render to a SceneNode it will call its children to update/render as well. This cycle will continue until we get to the leaves of the tree where we can stop and see that we have gone through all the nodes. The drawing and updating itself will be done inside the unit which is linked (with inheritance) to another unit. Removing these nodes and Units is up to the WorldMap class which removes The Unit classes after they have first marked themselves to be removed.



The left shows the Scene graphs structure and the right the SceneNode structure.
Pictures and the idea from the book: **SFML Game Development: Learn how to use SFML 2.0 to develop your own feature-packed game by Haller Jan**

Error handling

We will use one exception/error class which will have a changeable type and reason. This enables us to call the same class and just modify it to tell us why and where the exception/error happened.

Algorithms

General

Shooting

- When the units are attacking each other they should be able to do some micro prioritizing on which unit to attack. Also they shouldn't be all attacking the same target if the enemy is a group. There also is a randomness factor to make the units miss some of the shots they make.

Damage models

- When units for example attack or heal each other simple health calculations are made. Some units have different kinds of multipliers (eg. armor) that affect the damage amount. Also heavy units like tanks can angle their chassis so that they take

less damage from enemy shots.

AI-opponent

The AI needs multiple types of algorithms to work. The AI for example has to be able to prioritize which enemy units are most dangerous, make decisions based on the current situation in the game world and put them into action.

AI decision making

- AI decision making could be made with simple states. AI decides distinctly for each unit group if it should be in a certain state. For example if a unit is low on health, the AI should put it in a defensive state where the unit might be ordered to retreat or heal. Other states could be an offensive state, passive state, etc. depending on how many parts we want to divide your AI decision making.

AI prioritizing

- The AI should also know how to prioritize targets. This could be achieved for example with calculating a ‘danger value’ for each enemy unit based on unit stats and other info (eg. current health, position or group size).

AI actions

- After making a decision the AI should also be able to put these decisions to action.

Movement

Path finding

- Units need to know how to navigate the world map and avoid obstacles like buildings and walls. This is where we need to use a pathfinding algorithm.

Resource collecting

- When giving an order to your builder unit to collect resources from a resource pile, the builder unit needs to know how to bring the resources to the base and go back to the pile for more resources. The resource collecting algorithm uses the pathfinding algorithm to achieve this.

Information structures

Enumeration - Types etc.

We will use an enumeration system to easily differentiate different types of units like air, land and sea units or defensive and passive buildings.

Struct - Stats etc.

Structs can be used as a help struct when we need to store data in a systematic way. For example we can store our units stats and important data in struct so we have them all packed together and that we can call them in a descriptive way which makes the code easier to read and understand

Map - ResourceHolders etc.

The resource management for example is based on a map where the heavy textures and sounds have been loaded and paired with enumeration ID's. This will give as simple and quick access to the resources. With the same ID we can easily use the same texture in multiple places.

Tree - Scene Graph

For simple iteration through our Units we will be using the Tree data structure. This enables us to implement a simple layering and grouping system. In the Tree there is a root node which will have empty nodes (The starter nodes of the different layers) as children. After these empty nodes come the units which are attached to the layers. A group can be formed simply linking the child nodes to the parent as a sub tree. When the Tree is done it can be easily and quickly traversed just by calling a function to the root of the tree which will call the same function to its children. This cycle continues until we reach the leaves of the tree where we can stop the iteration.

Stack - StateStack

When managing the games states we can use the stack data structure. We can easily manage the states when the current state is on top and the older states are under it (**LIFO - Last one in First one out**). Then when we want to go to a new state we can place it on top of the stack and when we want to return to the earlier state we just pop the top state and delete it.

Queue - CommandQueue

We use a queue data structure when handling commands coming from user input handling to the SceneNodes/Units. This allows us to handle the commands in the order they have been made (**FIFO - First one in First one out**) so that the upcoming changes will be in the desired order.

TMX - WorldMap

This is a file created by an external program called **Tiled**. It's a special formatted XML-file that can be parsed with an external framework called **TPS**.

Timetable

Week 1 (13. - 19.7.) **Planning**

- Research, UML, structure planning, design

Week 2 (20. - 26.7.) **Base structure of the game**

- Base virtual classes like Worldmap, Units, Resource classes

Week 3 (27.7. - 2.8.) **Game logic and Graphics**

- Scenenodes, commands, game logic

Week 4 (3. - 9.8.) **AI and UI (Mid-term meeting)**

- AI logic, UI

Week 5 (10. - 16.8.) **Game flow**

- AI integration, Game states, Compiling with CMake

Week 6 (17. - 23.8.) **Finishing touches**

- Sounds, Texture designing, final testing, final bug fixing

Week 7 (24. - 27.8.) **Documentation**

- Final documentation

Division of work and responsibilities

Every group member will have a dedicated focus area. However we will be flexible when needed. There will be some big picture decisions and difficult topics which need group effort which will be tackled together.

Topic assignments

Otto	GUI, Graphics, Gameflow
Joona	Map, Graphics
Justus	Units, Units AI, Units GUI
Zakhar	AI, Units AI

Testing

We will have a great focus on testing during development and implementation of new functionality. We will cross-check each others' implementations and test them when working on them and in addition write automated tests. We have also reserved some time for thorough tests at the end of the project development.

Literacy references and links

SFML Game Development: Learn how to use SFML 2.0 to develop your own feature-packed game; **Haller Jan ; Hansson Henrik Vogelius ; Moreira Artur - Packt Pub 2013**

SFML Documentation: https://www.sfml-dev.org/documentation/2.5.1/

Tiled Documentation: https://doc.mapeditor.org/en/latest/reference/support-for-tmx-maps/

Tiled map Editor: https://thorbjorn.itch.io/tiled

SFML tilemap parser: https://edoren.me/STP/

Attachments

The UML graph PNG is in the plan folder of the project