Project documentation

Real time strategy game (Jatkosota)

Group info

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

Overview

The Jatkosota video game is a real time strategy game in which the player controls the Finnish defence forces to battle the Soviet army. The game takes place during the continuation war in Tali-Ihantala.

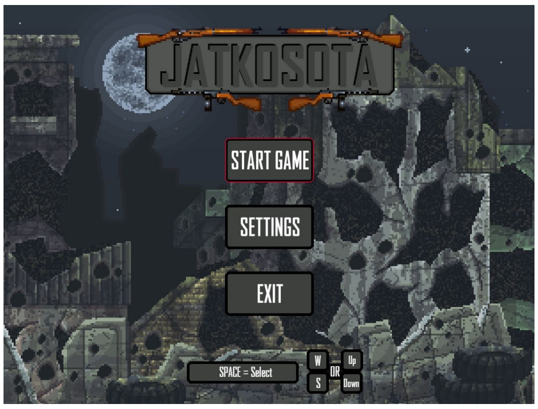
The purpose of the game is to either control the flagpoints or to destroy the enemy's forces. The player has a variety of units available to complete this task; Infantry, tanks and planes.

The most basic units are engineers, which are used to gather iron from iron piles and to build barracks, factories and airfields. In addition they can build defensive structures, like anti-tank or anti-air turrets.

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 (the Base and an Engineer) spawn on the edges of the map in the designated spawn points.

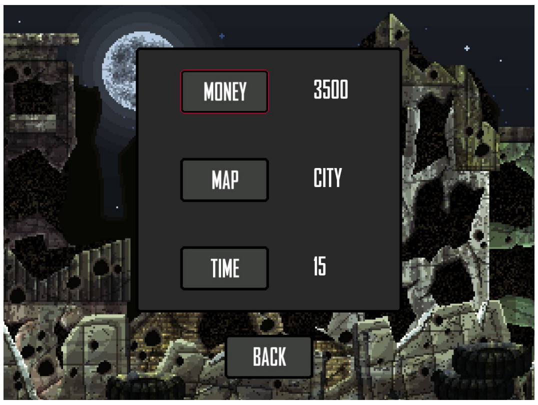
The end-goal of the game is to destroy all of the opponent's units or to capture the flagpoints for long enough. The player also has an option to change some of the game settings like the timer length or starting money amount.

Main menu



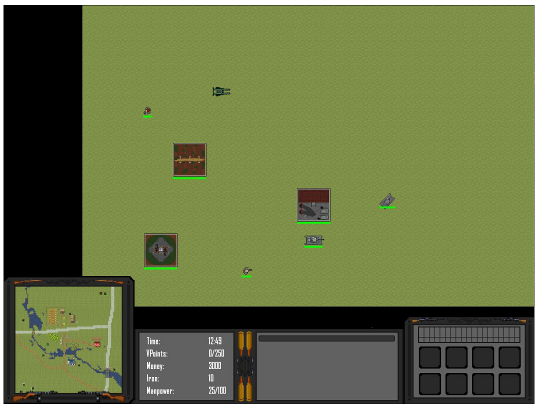
In the main menu the player can access the settings, start a new game or exit the game. The menu controls instructions can be seen on the bottom of the screen.

Settings


MONEY 3500
MAP CITY
TIME 15

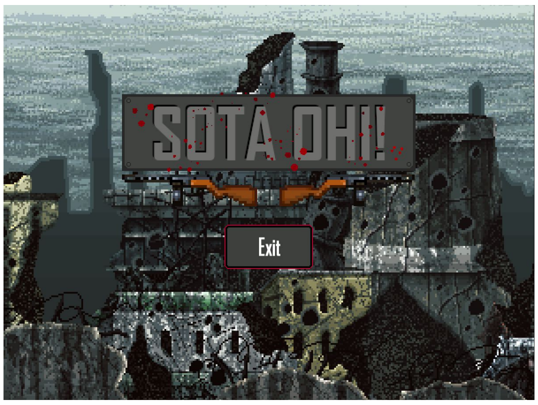
In the settings menu the player can change the map (currently all options are the same), the starting money amount and timer length.

Game


```
Time:        12:49
VPoints:     0/250
Money:       3000
Iron:        10
Manpower:    25/100
```

The actual gameplay happens in the match. Here the player can control units and state of the match. In addition important info about the current state of the game are shown to the player.

Game over



The game over screen is shown when the game ends. The player is given the option to exit the game.

<u>Units</u>

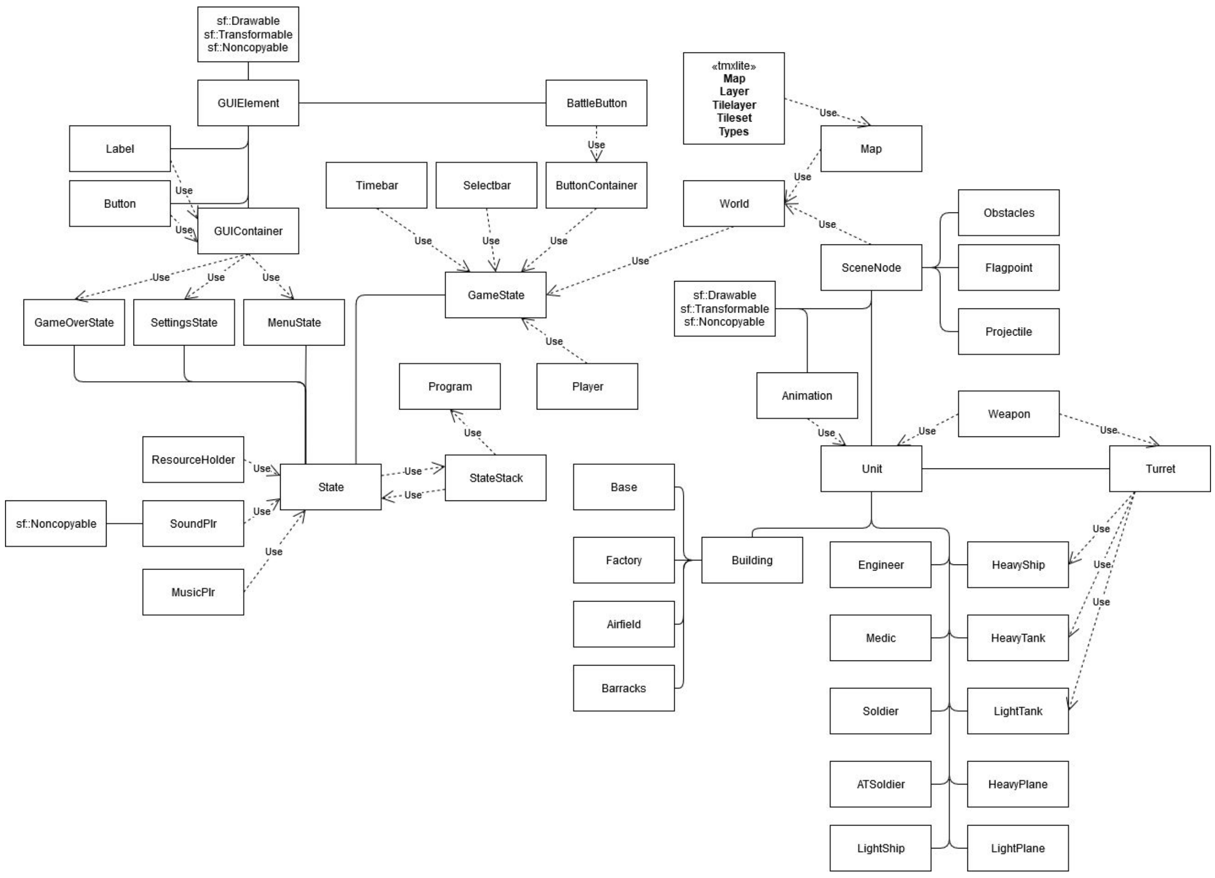
<u>Current units which have implementation at some level:</u>

- Light and heavy tanks
- Light and heavy planes
- Light and heavy ships (inaccessible in game, currently on land)
- Soldier, engineer, AT soldier, medic
- Factory, base, barracks, airfield

<u>Mechanics left out</u>:

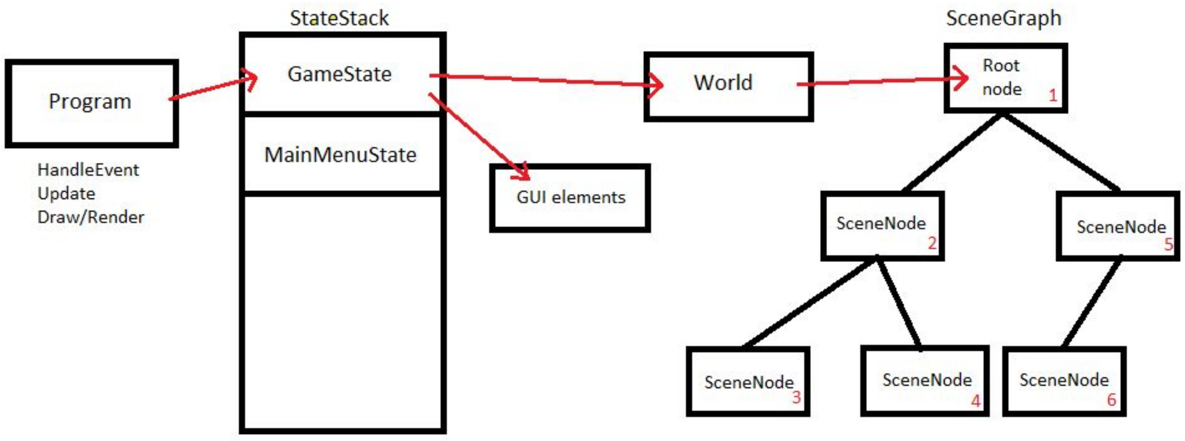
- Heroes
- Multiple game modes / maps
- Naval units and water
- Proper plane mechanics

Software structure



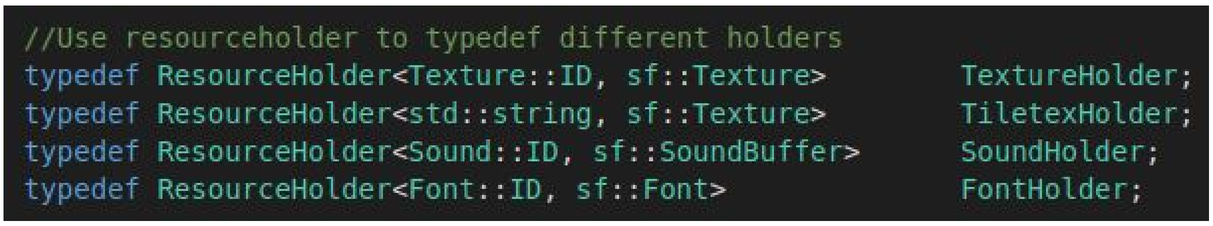
Here we can see the class relations of the program (excluding unused classes). The program uses SFML 2.5.1 and tmxlite libraries.

Run flow



The flow of the run cycle of the program starts from the program class which first polls for user input events, then calls the game to update and if enough time has passed calls the game to render. These signals are then passed on to the top most state on the statestack which directs the calls to gui elements and the world class. The world class then passes the signal to the SceneGraph (root of the scenenode tree) in which the signal traverses the tree in pre-order until all attached nodes have received the function call signal. After this all events have been handled in the proper places, the game has been updated accordingly and the changes have been rendered to the window.

Resources

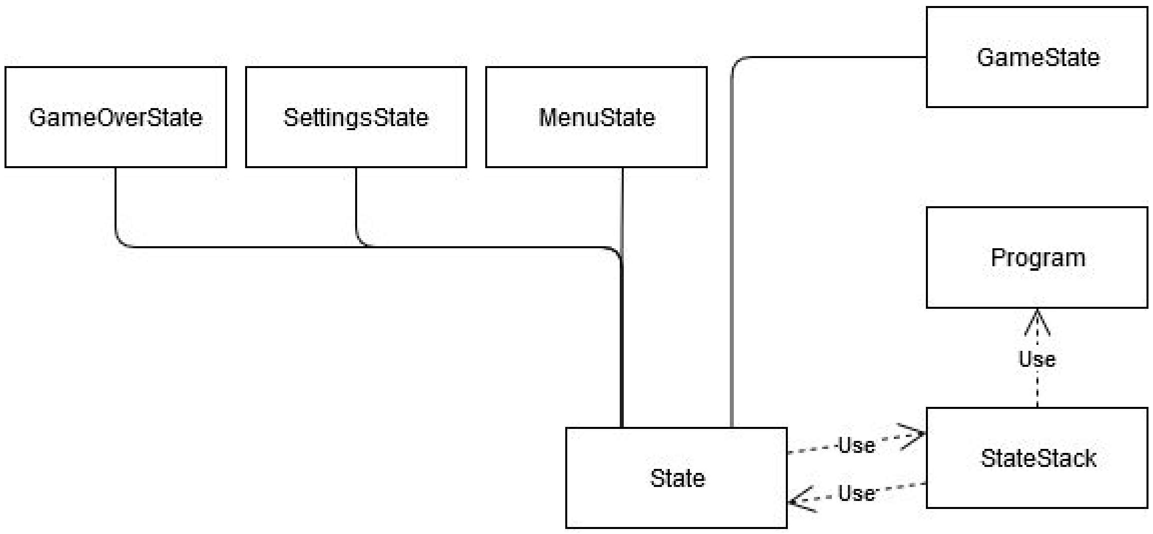


```
//Use resourceholder to typedef different holders
typedef ResourceHolder<Texture::ID, sf::Texture>        TextureHolder;
typedef ResourceHolder<std::string, sf::Texture>        TiletexHolder;
typedef ResourceHolder<Sound::ID, sf::SoundBuffer>      SoundHolder;
typedef ResourceHolder<Font::ID, sf::Font>              FontHolder;
```

The ‘ResourceHolder’ class template frame of the resource management system in the program. Type definitions SoundHolder, TextureHolder, TiletexHolder and FontHolder are used to hold the heavier sf::SoundBuffer, sf::Texture and sf::Font classes which have media loaded.

To access the data from the holder classes, ‘Get’ function is called with an enum/string ID which specifies the exact resource we want. The return value of get can now be used with lighter classes (e.g. sf::Sound or sf::Sprite) although in the current state, the program uses sounds and music which are loaded into the SoundPlr and MusicPlr classes directly.

States/StateStack



State stack system is used to change the state of the game, for example from the main menu state to the game state. The state stack works as a simple state machine, where states are stored in a stack in which the state on the top is the active state. This allows quick and seamless transition from state to another. The struct ‘State::Content’ (Defined inside state) is used to transfer media/information between states.

When we want to change the state, we simply request a state pop which removes the top most state (the active one) and request the new state to be added on the top. When the next update is called the state change will be executed.

The main State in the program is the GameState. This is where all the gameplay happens. This state controls user input, world and player data and directs information between them. The gamestate also loads the saved settings and sets them in action.

World

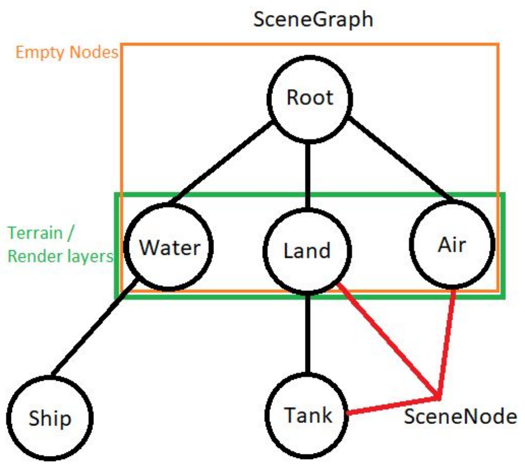
The World class handles all the events happening in the world. During creation the class loads needed assets (textures, map, passive and starting units) which are then stored in the State::Content struct. The World class uses the SceneGraph to update and render units information and sprites. It in addition handles the destruction of the SceneNodes/Units.

The planned CommandQueue wasn’t used in the final implementation.

TMX-parser

The Map class is responsible for handling the TMX parsing. The map is created using an external software called Tiled, which is a graphics based tilemap creation software. The maps consist of TMX-files which consist of multiple layers of data for the tiles. The tiles use tilesets which are image-files that contain a texture for the tiles. The tiles are identified via global tileID:s. Class Map loads a TMX::Map (using the tmxlite library for SFML) into the world and draws it. It also reads location data from different object layers which are then passed onto the World class. This location data is used to place flag points, iron piles and obstacle hitboxes to their designated areas.

Rendering/SceneGraph

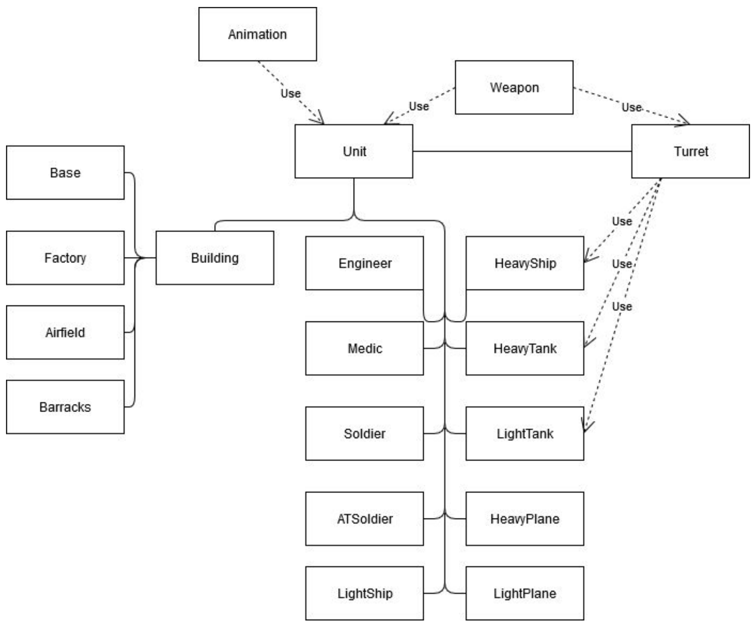


For rendering objects into the world, the game uses the ‘SceneGraph’ in which ‘SceneNodes’ are connected as a tree structure. The root of the tree is an empty

SceneNode which the SceneNodes are given a pointer to. This way we call the whole SceneGraph to be iterated through from any node of the tree.

The children for the root are also empty nodes. These nodes represent different render layers / terrain types of the SceneGraph. As the tree is traversed in pre-order the whole branch under one layer node is gone through (in both update and draw) before moving to the next layer node.

Units/AI



The units are represented by the abstract class ‘Unit’. This provides resources and default implementations, some of which can be overridden in derived classes, for things like receiving damage, attacking and moving. Unit is a very large class and many units leave some functionality unused, but the increased complexity of a divided class structure would’ve made refactoring prohibitively time-consuming.

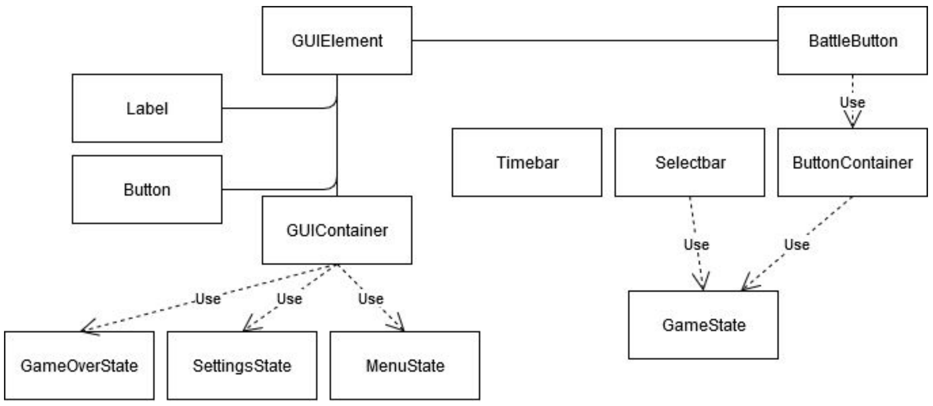
Individual units are represented by their own branching classes, one for each unit type. These define all the unit characteristics and are further derived into the final definitions for the different sides, which means equivalent units on different sides are identical in all but graphics. Buildings have their own branching class, which gives all of them the same base stats only altered by turrets in defensive buildings, as well as spawning functionality.

Turrets in units like tanks, ships and defensive buildings are represented by a SceneNode derivative called Turret, which provides functions for using the weapons and logic for independent turret control or unit-level control. Turrets are attached to their parent unit as ‘hidden children’, which aren’t seen by SceneNode and are instead handled by unit.

Weapons are handled by their own class, which handles cooldowns, firing, weapon stats and ammunition. It has functions for changing ammo types, automatically selecting the best shell for a given target, and firing. The coordinates aren’t handled by Weapon, rather they are given as parameters to the ‘Fire’ function. Weapons can be attached either to turrets or directly to units.

‘Projectile’, which is a SceneNode derivative that handles all the projectiles fired by weapons. These handle all the projectile stats, such as damage, speed and splash radius, as well as collision checking. Firing is achieved by initializing the projectile, whose constructor adds it to the scenegraph.

GUI



Menu GUI

The menu GUI consists of GUIElements which are stored and managed by a GUIContainer. For example buttons are packed in the creation of a menustate into the containers in which they can be cycled through and activated. The button W/Up cycles the current selected button up and buttons S/Down cycle down. Space activates the key.

Match GUI

The match GUI is more complicated than the menu GUI. It has mouse interactable buttons, a minimap, a timerbar, a selectbar, a select rectangle and unit interactions.

All units which are not passive are selectable and most have push buttons which are shown after selection. The player can select units with click and drag during which a select rectangle is displayed. The buttons activate the unit's abilities, for example Buildings can spawn units or engineers can harvest iron.

The player can move or attack with the selected unit by clicking a point on the battlefield. If the point is an enemy, selected units will attack, if not the game checks if there are no obstacles after which the unit starts heading to the point. Having multiple points the GUI displays lines to show where the unit is heading.

Above the unit buttons is the select bar which shows info about the selected units. The player can see selected units health and type.

The timerbar shows a timer when a unit is constructing or spawning a unit. After the bar has finished is the action ready.

The data window displays important game data to the player. For example the player can see the game timer form there. In addition, players, money, manpower, iron and Victory points are shown.

Sounds

The MusicPlr and SoundPlr classes handle playing music and sound effects in the program. When they are initialized they load .wav -files and pair them with IDs similarly to resourceholders. Because the classes hold many loaded sound files, they are created once in the start of the program and passed to needed places as pointers inside State::Content.

Use instructions

Build and run

- Must have SFML 2.5.1 installed

Build (In the src directory)

1. Delete older cmake files (Cmake folder, CmakeCache, cmake_installs, MakeFile)
2. cmake CMakeLists.txt
3. make

Run

4. ./Jatkosota

Controls

Menu controls

W or Up	Select button above
S or Down	Select button below
Space	Activate action / cycle through settings

Match controls

Click button	Activate button
Click and drag	Select all units inside selection rectangle
Click unit	Select single unit
Shift + click (When unit selected)	Add a point to the units move path
Right click (When unit selected)	Deselect units
Left click (When unit selected)	Move or Attack target
W or Up	Move camera up
S or Down	Move camera down
A or Left	Move camera left
D or Right	Move camera right
Mouse scroll	Zoom in or out

How to

Obtain Iron:

- Engineers can obtain iron, move a engineer unit close to an iron pile and press the harvest iron button and the engineer will start harvesting iron and bring it back to base

Buy a unit or structure

- Engineers are used to build structures. The engineer needs to be still to build a structure. Press the wanted structure button to build it close to the engineer. Same can be done with buildings, but they only spawn units

Obtain Victory points

- To obtain victorypoints simply move your unit to a flagpoint. You need to have more units on the flagpoint than the enemy.

Testing and Debugging

Due to large workload and time constraint testing wasn't done properly (for example no unit testing was done). Large amount of testing was done with just running the program and checking if the features worked properly.

Git branches

To avoid having only unbuildable code in the master branch we used feature branching. When we began coding a new feature a new feature branch was made which would be merged into the working master after the code was tested to run properly without errors or crashes.

Test main

We also had a test main file to test individual functions like loading settings from a file.

Visual testing for Graphics

For GUI features visual testing had to be done. Modifying the code and then checking the GUI as the program ran was an essential part of the GUI testing.

Error Handling

To avoid unknown bugs/errors we used runtime_error to throw exceptions messages in the most critical parts of the program (for example when loading resources or files into the game).

GDB with core dumps

Handling pointers wasn't always successful. Initially we mostly used raw pointers with some smart ones. Mixing different independent raw and smart pointers sometimes caused the objects to be unexpectedly deleted, and especially unique pointers caused a lot of segmentation faults.. To debug these, we generated core-files which we traced back with gdb to get debug information about the problems. In the end we had to mostly switch to smart pointers, mostly shared ones with weak pointers used for temporary reference. Raw pointers are still used for SceneNode parent and getting the scene graph root from World.

Division of work

Person	Main responsibility	Additionals
Otto Simola	GUI, Main structure, Graphics (Visuals and code)	Buy system, unit spawning, world creation
Joona Repo	World, Map, TMX-Library	Map creation
Justus Ojala	Unit structure, Unit AI	Misc structural changes, Units
Zakhar Burda	Units, Units AI, enemy AI	Unit structure, changes affecting units in other files.

Worklog

Week 1 (13. - 19.7.) - Created files and empty class structures, planning and coding the main run frame

Creating files system and template classes	Otto
Begun creating resource classes	Otto
SFML Game Development research	All
Planning	All

Week 2 (20. - 26.7.) - The main frame and resource classes. Begun coding other segments

Finished resource classes	Otto
Preparing own laptop and git systems to work	Zakhar
World-class and map design	Joona
Map perspective changes	Joona
Map texture search	Joona
Meetings	All

Week 3 (27.7. - 2.8.) - Making progress towards the program to be runnable, parser library working. World and units class structures begin to form.

Program class	Otto
Headers making for units	Zakhar
Creating list for monitoring units	Zakhar
Configuring for Linux	Justus
Implementing class prerequisites and definitions	Justus
Initial implementation of Worldmap class	Joona
TMXlite parsing library installation and study	Joona
Meetings	All

Week 4 (3. - 9.8.) - StateStack system working, implementing the parser library and continuing the Unit class structure

StateStack ja State	Otto
SceneNodes	Otto
Menu GUI elements	Otto
Working on Unit.hpp\cpp	Zakhar
Working on Unit's children	Zakhar
Unit-related headers, small Unit improvements	Justus
Worldmap class divided into classes World and Map	Joona
Class map for TMX-parsing implemented	Joona
Test map creation	Joona

● Map loading functionality implemented	Joona
● Meetings	All

Week 5 (10. - 16.8.) Cmake working, the game runs visually. GUI and world graphics started.

● Cmake ja make files	Otto
● Making textures	Otto
● Finding assets (Music, textures, sound effects)	Otto
● Main menu	Otto
● Settings menu	Otto
● Working on structure of Units	Zakhar
● Heal, attack, build etc functions for units	Zakhar
● Constructors, destructors to units	Zakhar
● More Unit functionality	Justus
● Started buildings	Justus
● Framework for spawning units	Justus
● Map loading functions fixed	Joona
● Tile texture loading and saving implemented	Joona
● Implemented update function for game updates	Joona
● Textures for final map	Joona
● Meetings	All

Week 6 (17. - 23.8.) - Creating GUI and implementing unit graphics

● GUI data windows	Otto
● GUI game state buttons	Otto
● Minimap	Otto
● GUI Player input View moving, selecting and move	Otto
● Timerbar	Otto
● Making textures	Otto
● Working on structure of Units	Zakhar
● Classes of buildings, infantry for FIN, USSR sides	Zakhar
● Explosion animation for HeavyTank	Zakhar
● Functions for Units	Zakhar
● Unit AI related functions	Zakhar
● Virtual system on Linux with cmake	Zakhar
● Moved aggressive, mobile to Unit	Justus
● Abstract constructors protected	Justus
● Early branch implementations	Justus
● Improved spawning framework	Justus
● Attack implementation	Justus
● Implementing projectile	Justus
● Movement changes	Justus
● Implementing turret	Justus

● Unit logic framework	Justus
● Implementing weapon	Justus
● Implementing required world functions	Justus
● BuildScene() function for game initialization created	Joona
● Texture loading changes	Joona
● Map location system	Joona
● Final map design	Joona
● Player-class hasAliveUnits() -functionality	Joona
● Meetings	All

Week 7 (24. - 27.8.) - Unit functions like buy, attack, move. Finishing GUI, AI and players. Passive unit handling and functions.

● Unit creation and placement	Joona, Otto
● Game over screen	Otto
● Loading screen	Otto
● Building and engineer buy system	Otto
● Texture making	Otto
● Unit movement and graphics	Otto, Justus
● Victorypoint system	Otto
● Iron resource system	Otto, Joona
● Map obstacles	Otto, Joona
● Selectbar	Otto
● GUI Player input, create paths, attacking	Otto, Zakhar
● Engineer autonomy	Otto, Zakhar
● Medic Heal final implementation	Otto
● Solving Virtual system problems	Zakhar
● Make units distinguish allies and enemies	Zakhar
● AI related functions	Zakhar
● Enemy AI related functions	Zakhar
● Coordinate, draw changes	Justus
● Finalized unit, turret, projectile, weapon	Justus
● Changes to world, gamestate, scenenode	Justus
● Memory changes	Justus
● Final map created	Joona
● Collision system (obstacles) added to map	Joona
● Implemented tile transformations	Joona
●	
● Meetings	All

No proper working log was held, so these might be few hours inaccurate

Week	**Joona Repo**	**Justus Ojala**	**Zakhar Burda**	**Otto Simola**
1	10	3	4	20
2	15	3	13	16
3	15	8	8	16
4	15	15	11	20
5	15	15	18	40
6	20	25	29	60
7	10	55	38	45
total	100	124	121	217

Conclusions

Future improvements

If we continued the development of this game here are some things we would focus on:

- Includes and unit data stored in own files
- Huge classes like Unit, GameState and World could be broken into subclasses
- Menus with mouse buttons instead of keyboard buttons

Known bugs and deficiencies

Due to ambitious goals and time constraints, some features were left unimplemented or were implemented in a hurry, leaving bugs in the game. For the sake of honesty and not to hide any here are few we found.

- Engineer can’t spawn all the units properly
- Stat balancing (damages, ranges, armor, speeds…)
- Defensive buildings have no hitboxes (reason known)
- Units will continue to try to attack destroyed user-designated targets
- Splash damage crashes the game (don’t use HE or heavy planes!)
- Destroying all units, including the base, doesn’t end the game
- Enemy AI doesn’t send units to attack the player
- Tanks lack machine guns
- Projectile starting points aren’t offset based on gun locations
- Spawning multiple types of units breaks the game
- Before post-deadline commit 231a163e06
    - Game does not end when time runs out
- Before post-deadline commit 86e9a96d80:

- Units can only fire at their own terrain layer
- Units will try to fire at designated target out-of-range
- Planes kill themselves with bombs (related to the two above)

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 tmxlite :https://github.com/fallahn/tmxlite

TMXlite reference: https://github.com/fallahn/tmxlite/wiki