Project Documentation

1. Abstract

Our idea was to make a game where 1 or 2 players can direct airplanes and fight against each other or computer opponents. We were planning to make multiplayer with same computer and over network. We wanted to create a good AI, level editor and a game engine. We didn't manage to make multiplayer over network, nor different types of missions, but we're happy with the end result of our efforts.

2. Group

Tam Nguyen
Lauri Westerholm
Oskari Järvi
Lauri Blomberg

2.1. Responsibilities

Generally, everyone has all kinds of coding everywhere. However, they also have main responsibilities:

Tam Nguyen:
- Automatic Build System (Makefile).
- Defining project structures.
- README.md (instructions).
- Setting up git configurations (gitignore, gitattribute).
- Setting up doxygen configurations (Doxyfile).
- Team coordinating (slack).
- Consulting
- Creating tasks
- Modernize codebase by using C++17 features and refactoring.
- AI
- CommonDefinitions
- Entity : defined Entity structures and methods
- GameEngine:
  60 FPS rendering,
  user inputs reacting.
- PhysicsWorld:
  Make static objects visible to dynamic objects.
- Plane:
  plane movements
- ResourceManager
- World

        Killing list
- produceCommonDefinition.sh

Lauri Westerholm
- UI and its subclasses LevelEditor and MainMenu
- Button, ImageButton, TextInput (UI related classes)
- Level and LevelEntity (LevelEditor related classes)
- Stats
- main.cpp
- World
    - Hit detection
- GameEngine
    - GameOver screen
    - Scoring system

Oskari Järvi
- World- and PhysicsWorld -classes.
    - SFML and Box2d relationship.
    - Game physics
    - Creation of entities
    - Updating world
- Player controlled plane shooting.

Lauri Blomberg
- Entity class and it's subclasses
- Shooting for AI

3. Repository content

3.1. ./
Root of the repository contains Makefile which is used to compile the project binaries. It also contains Doxyfile for generating Doxygen documentation of the project.

3.2. /src
Source folder contains all source code and headers implemented apart from files that are used purely for testing. During compilation object files and the main executable *game* is build to src.

3.3. /test
Contains unit test mains for the project. These file were used during testing phases of the project. Test executables and objects are build to this folder by make test.

3.4. /data
Contains subfolders for data related files. Fonts are stored in *fonts.* All game entity and button related images are in *img* and one music file is in *music. logs* contains spdlog generated run logs (one for each run). *level_files* contains game levels which

are created by ingame level editor and *level_img* auto-generated images matching levels. *misc* contains all miscellaneous data files which don't fit to any other subfolder (i.e. files used to generate help screens and stats view)

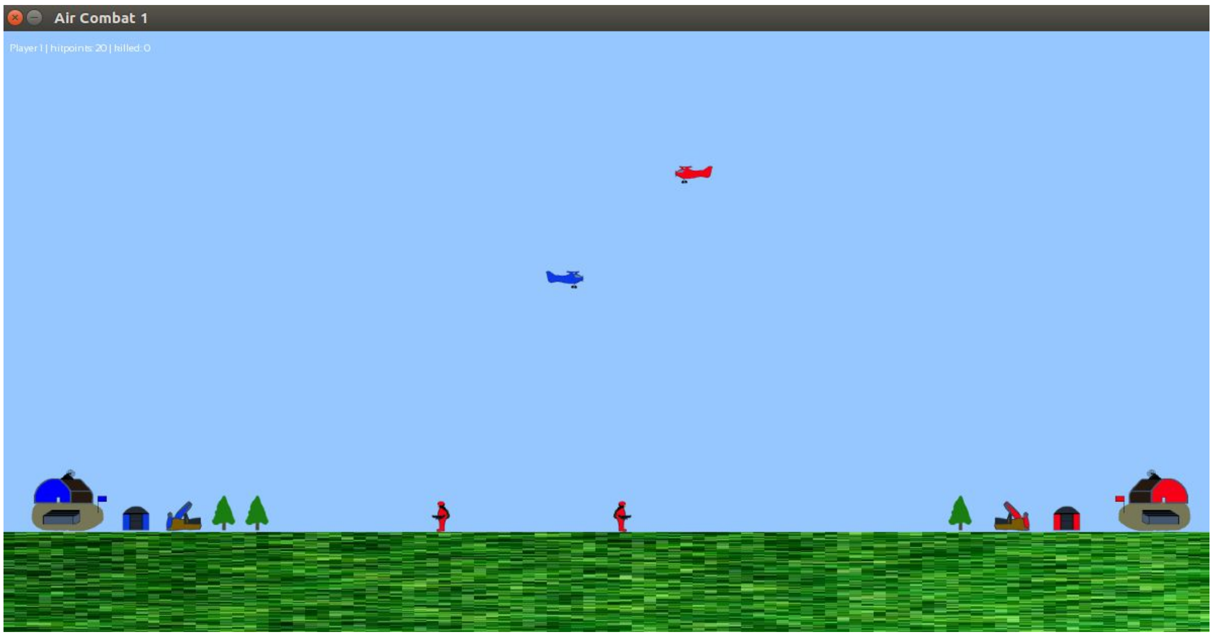
3.5. /doc

Contains the project documentation.

3.6. /plan

Contains the project plan and couple text files which were created during essential implementations, level format and main, were planned.

4. Gameplay

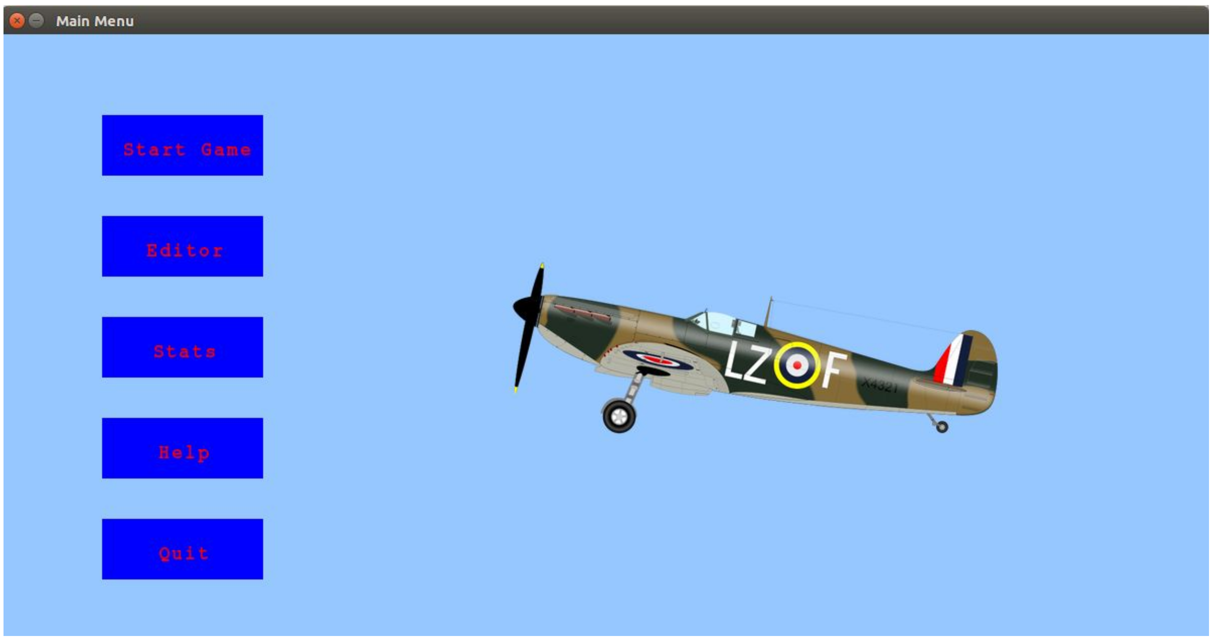


4.1. Read Readme.md for build instractions

4.2. Run *game* to start the air combat game

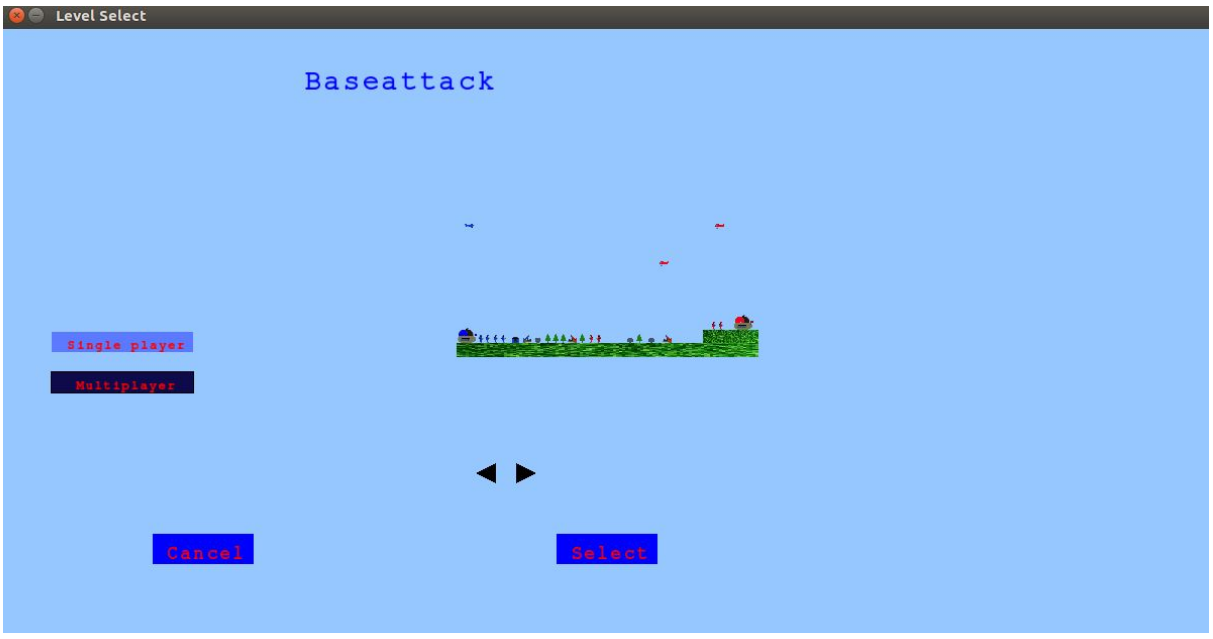
4.3. Main Menu

- Navigating
    - Tab
    - Arrow keys
    - Mouse
- Click button with mouse or Enter



4.4. Level Select

- Cancel / Select
  - Keyboard Tab & Enter
  - Mouse
- Single player / Multiplayer
  - Up & Down arrows
  - Mouse
- Level
  - Left & Right arrows
  - Mouse



4.5. Controls

Can be found in Main Menu by pressing Help

Single player:

A: Move left
D: Move right
W: Move up
S: Move down

Q: Rotate counterclockwise
E: Rotate clockwise

Z: Shoot

Multiplayer:

Player 1 has same controls as in the single player mode

Player2

J: Move left
L: Move right
I: Move up
K: Move down

U: Rotate counterclockwise
O: Rotate clockwise

M: Shoot

Use ESC to quit ongoing game and return to Main Menu

4.6. Winning & Losing (also in Main Menu Help)

Single player

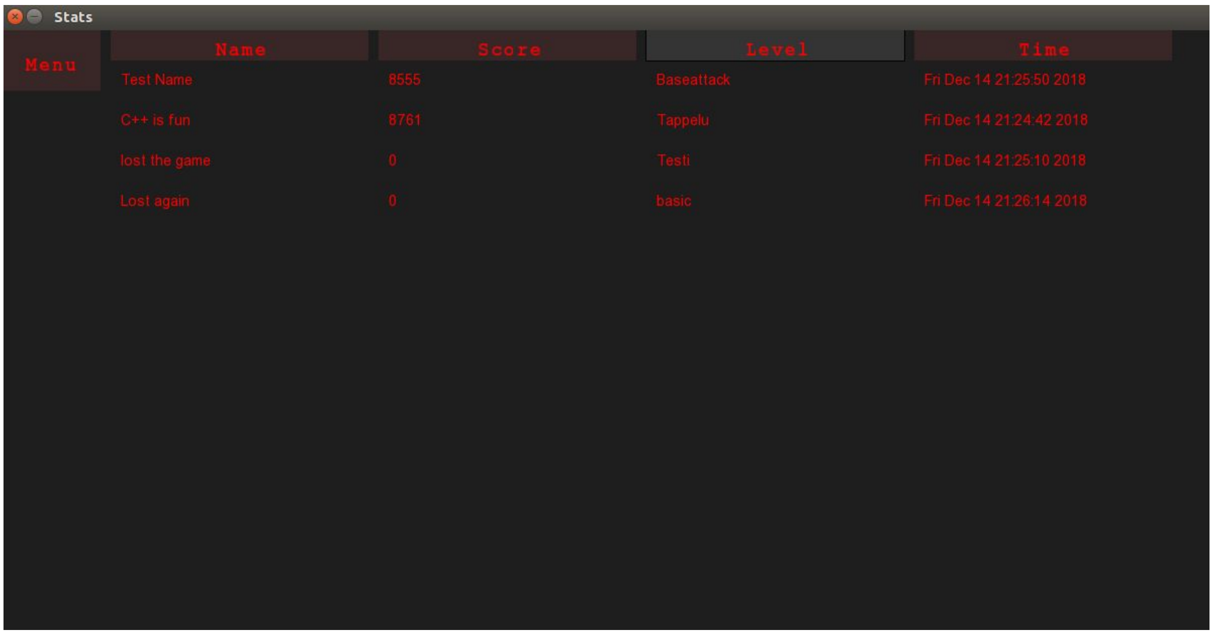
- Destroy all red planes and red bases to win
  - Score is awarded from kills and time (try to be as fast as possible)
  - Score can be found from Stats
- Blue plane destroyed -> lost
  - Score is 0

Multiplayer

- Destroy enemy plane
  - No score is awarded nor Stats entry generated

4.7. Stats

- Use buttons to sort stats
  - Name: sort by name
  - Score: sort by score
  - Level: sort by level name and score
  - Time: sort by time (newer first)
- Use arrow keys to change view
  - Up arrow: up one view
  - Down arrow: down one view

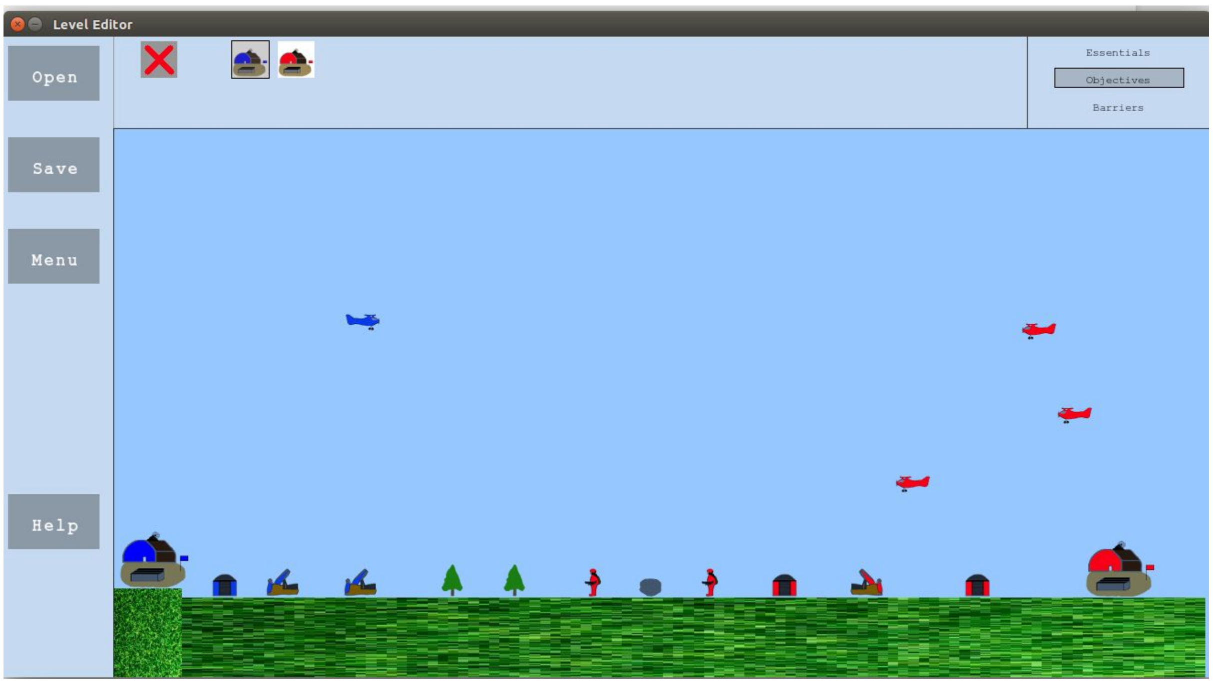


Name	Score	Level	Time
Test Name	8555	Baseattack	Fri Dec 14 21:25:50 2018
C++ is fun	8761	Tappelu	Fri Dec 14 21:24:42 2018
lost the game	0	Testi	Fri Dec 14 21:25:10 2018
Lost again	0	basic	Fri Dec 14 21:26:14 2018

4.8. Level Editor

4.8.1. Read Help from Editor

4.8.2. Use mouse to navigate & click buttons



5. Project structure

5.1. Technologies

We used following technologies to make this game possible: C++17, g++, Box2D, SFML, spdlog, gdb, Makefile, git, Bash, Gimp, Valgrind, Doxygen

5.1.1. C++17 : A modern programming language with extensive libraries and syntax sugars make coding easy and safe. For example, “auto”, smart pointers, containers, for-each, constexpr, lambda-functions, override, etc.

5.1.2. g++ : compile our game. Version: 4:5.3.1-1ubuntu1

5.1.3. Box2D : a physic engine which mimic a real world in software world. This engine make the game feel realistic.

5.1.4. SFML Version: 2.3.2+dfsg-1 : a simple and fast multimedia library helps to project Box2D world to a user screen. We used this library to draw and play music.

5.1.5. spdlog Version: 1.6-1 : is used for logging game engine states. However, the library on Aalto Linux machine is old and it isn’t in high priority, so spdlog is not widely used in our project.

5.1.6. gdb : GNU Debugger is used for debugging application and modules.

5.1.7. Makefile: build automation tool to generate desired files/binaries.

5.1.8. git : Our chosen source code management tool. The best part is this works locally.

5.1.9. Bash: a linux shell script used to generate values for variables in CommonDefinitions.cpp and CommonDefinitions.hpp

5.1.10. Gimp : is used to generate textures for SFML.

5.1.11. Valgrind : for memory debugging and memory leak detection.

5.1.12. Doxygen Version: 1.8.11-1 : For creating the documentation from the source code.

5.2. Classes

5.2.1. AI

Determines entitys next action in relation to its surroundings. Calls move and shoot functions in appropriate classes.

5.2.2. Artillery

Entity’s subclass that implements artillery objects and their shooting. Artillery entities can only shoot planes.

5.2.3. Base

Entity’s subclass that implements bases for both teams. In single player has a significant role: player’s goal is to destroy both enemy bases and planes.

5.2.4. Bullet

Entity’s subclass that implements all the bullets. Bullets cause damage to all entities apart from its owner.

5.2.5. Button

Object created to be a part of the user interface. With help of UI, Buttons can handle mouse move, click and keypress based actions.

5.2.6. CommonDefinitions

Contains constants and namespaces that are used by many other classes.

5.2.7. Entity

Masterclass for all entities. Implements all the basic functions for them as virtual functions.

5.2.8. GameEngine
Class integrating World and user input. Updates World according the input and also acts as a link between main menu and the base game.

5.2.9. Ground
Entity’s subclass that implements ground. Ground entities are static and planes destroy immediately when they hit Ground.

5.2.10. Hangar
Entity’s subclass that implements hangars for both teams. Hangars are static entities without any major functionality.

5.2.11. ImageButton
Subclass of Button. ImageButtons implement an image as a button.

5.2.12. Infantry
Entity’s subclass that implements infantry objects, their moving and shooting. Infantry move on the ground and shoot all enemies that come within their range.

5.2.13. InvisibleWall
Entity’s subclass that implements side and top borders for map. InvisibleWall entities are static. They stop other entities from moving outside of the map.

5.2.14. Level
Used for creating levels for the game in the in-game level editor. Adds, modifies and removes LevelEntities on LevelEditor input.

5.2.15. LevelEditor
A UI derived class which is used to create user interface to the in-game level editor. This class also updates Level based on user input.

5.2.16. LevelEntity
Entity used during level is created in the in-game level editor. Level contains and updates LevelEntities.

5.2.17. MainMenu
Creates the main menu when game is opened and handles its functions

5.2.18. PhysicsWorld
Creates a world for Box2d physics objects. This class also handles creation of the Box2d bodies which are the physical representations of entity-objects.

5.2.19. Plane

Entity’s subclass that implements plane objects, their moving and shooting. Planes fly and attack other entities with bullets and bombs. Planes also have significant role in the gameplay: winning and losing depends on which team’s planes get destroyed first.

5.2.20. ResourceManager

Class which is used to get access to textures and other important resources in the base game and in the level editor. The main purpose of this class is reduce amount of heavy memory intensive resources.

5.2.21. Stats

Class which is used to create window for showing single player game statistics. Handles both the interface and reading & sorting the stats.

5.2.22. Stone

Entity’s subclass that implements stones as obstacles. Stone entities are static and cause damage when planes hit them.

5.2.23. TextInput

User interface related class which gathers user input to textbox and allows modifying the text. This class is mainly used to get player name at the end of a game and level name in the level editor. TextInputs use a std::set as a blacklist to sanitize disallowed characters (i.e. spaces from level names). Level names use more strict set than player names or level descriptions.

5.2.24. Tree

Entity’s subclass that implements trees as obstacles. Tree entities are static and cause damage when planes hit them.

5.2.25. UI

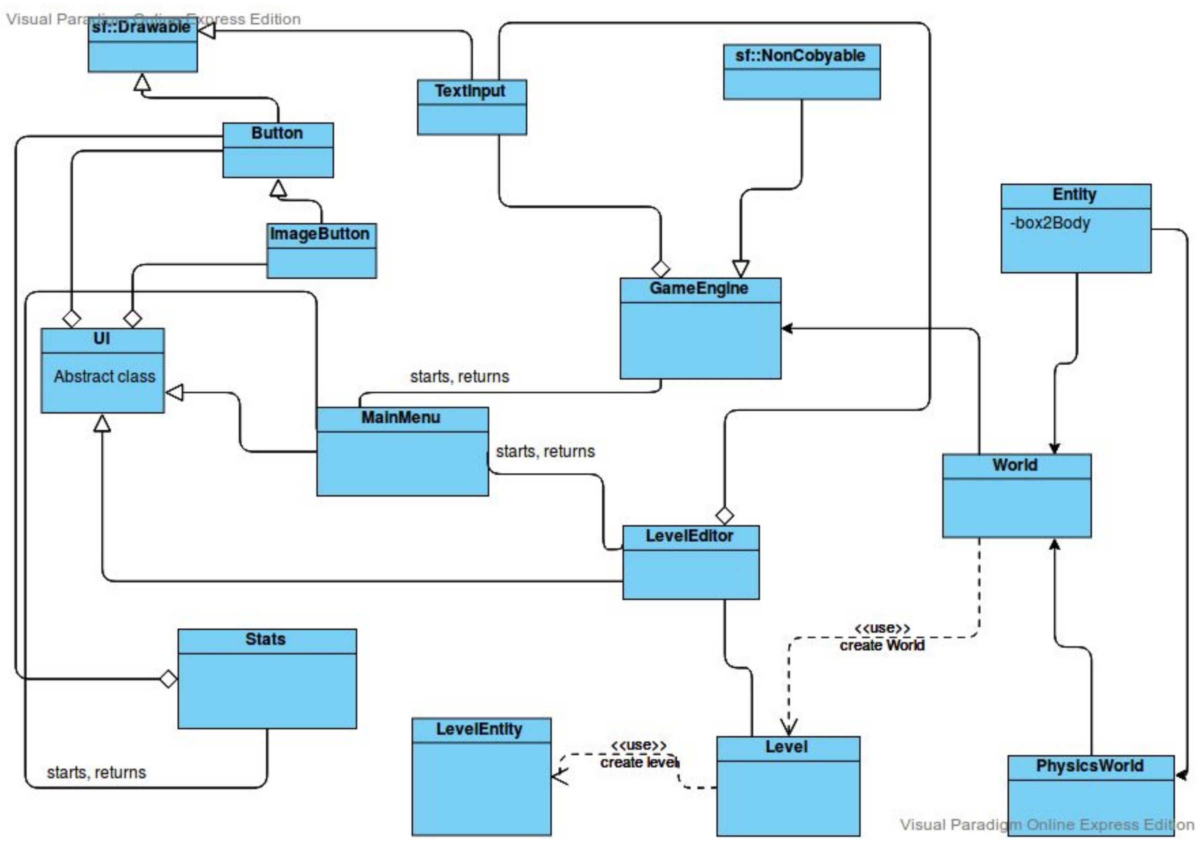
Base class for user interface. Implements basic user interface updating through SFML event polling.

5.2.26. World

Creates the main world that holds all entities. This class reads level files and creates entity-objects and their Box2d physics -bodies from it. World retrieves information about the physical representations from PhysicsWorld-class and updates entities according to it. This class also holds collision detection, score counting and drawing entities to the game window.

5.3. Class hierarchy

Recommended entry point for UML diagram is MainMenu which is used to start other parts of the game.



Notice: UML diagram of the class hierarchy is missing all Entity subclasses and ResourceManager & CommonDefinitions. The decision to exclude those is made in order to make UML diagram more clear.

For more detailed class relationships, run doxygen. It produces full hierarchy with all method, connections and resources shared between the classes.

5.4. Memory usage and pointer use

We have utilized smart pointers in many places. Namely, shared pointer and smart pointer.

Shared pointer is used when there are possible more than owners. This is used in about ten different classes.

Unique pointer is used when there is one owner. The only class that used this unique pointer is ResourceManager. ResourceManager loads all textures in “std::map<Textures::ID,std::unique_ptr<sf::Texture>> resourceMap”. If other

other classes need a texture for drawing, then resource manager will give a reference to the wanted texture instead of loading a texture separately.

We have not used malloc, calloc, realloc nor new anywhere in our code.

We have tested our game with valgrind to detect memory leak. As the result shows below, we don't have memory leak:

```
==179517== HEAP SUMMARY:
==179517==   in use at exit: 169,932 bytes in 559 blocks
==179517==   total heap usage: 197,947 allocs, 197,388 frees, 406,325,163
bytes allocated
==179517==
==179517== LEAK SUMMARY:
==179517==   definitely lost: 0 bytes in 0 blocks
==179517==   indirectly lost: 0 bytes in 0 blocks
==179517==   possibly lost: 0 bytes in 0 blocks
==179517==   still reachable: 169,932 bytes in 559 blocks
==179517==              suppressed: 0 bytes in 0 blocks
==179517== Rerun with --leak-check=full to see details of leaked memory
```

5.5. Container types

In general standard library vectors are used when containers haven't had any special requirements apart from need to store unspecified amount of objects. Thus, std::vector is the most used container in the project.

In some cases we have needed to add items to the front of the container. Then we have used the simplest solution, std::deque which is basically a doubly linked list. The singly linked list, std::list, has been utilized in some occasions to replace vector when many items are added during runtime and container is only accessed by iterating through it.

For mapping object instances together we have used std::map. Map is well-functioning container for those fixed-size data structures. For storing items in dictionary style we have used std::set which makes possible quick keybased lookups.

6. Testing

The design philosophy of the project was to implement parts of the game separately and after the parts were finished, integrate them. Thus, LevelEditor, MainMenu, Stats and the base game were first implemented on their own. They all have their own unit test mains in the test folder.

The testing itself was mostly visualization based (not many pure testing programs were used). We first implemented and debugged drawing of the objects to state where it was matching to the object coordinates. Then we used on-screen position to figure out bugs.

assert is used in source code for code validation.

For severe bugs, namely segmentation faults, and issues which we had trouble to debug, we used gdb. Runtime access to variables and their states was really helpful.

We compile every time with following flags: -std=c++17, -Wall, -Wextra and -pedantic. And compiler does not show any warning nor error on Aalto Linux machines (Paniikki, Maari A and Maari B).

At the end of the project we also tested our program, *game*, with Valgrind. The only memory leaks Valgrind reported were caused by sf::RenderTarget::clear which means that the leaks are within SFML not in our code. Valgrind also reported some invalid read of sizes relating to Entity getPosition(), getTeamId(), getTypeId(). Thus there still probably exists some entity removal related issues which are causing the invalid reads.

6.1. Known bugs

- Sometimes AI shooting creates the bullets to strange places.
- Infantry starts to spin in some place when it keeps changing it's direction with every update cycle, so 60 times per second.
- It happens that infantry or artillery entities clip with other objects thus creating vertical force sending them upwards.
- AI planes can crash each other sometimes.
- (When opening an old level level editor allows user to overwrite the same old level. There is a really minor bug by which user is also allowed to overwrite some other already existing level. The bug is so small that it wasn't ever fixed, would be really easy to solve though)
- (When exiting the game, SFML font deconstructor may cause a segmentation fault. Happens very rarely and the bug isn't in our code)

7. Possible improvements

It would be possible to add many improvements to the game that we didn't have time to implement. We only have hangars on the map but they are just immobile targets. We could make it so that they repair the planes damages and resupply the ammunition when plane touches the hangar or lands close to it.

We could also make bombs as other weapon type for players plane. AI could also use them when it notices any non-plane entities under it.

In the game there are already many different kinds of entities. It would be easy to implement different kinds of missions, like destroying enemy's base, surviving for 3 minutes or destroying X enemy infantry. We could also add some factory entity for special missions. They could also be player protecting them from enemy attacks.

We could also change the physics implementation to more airplane-like qualities so that planes would have turning radius and with low speed they would fall to ground.

Also we could improve AI-planes so that they shoot only straight ahead. Also shooting could be improved by AI calculating where it would need to shoot to hit with the players current trajectory.

8. Self-Assessment

1. Overall design and functionality (0-6p)

  * 1.1: The implementation corresponds to the selected topic and
scope. The extent of project is large enough to accommodate work for
everyone (2p)

We have implemented almost all features listed by the course homepage and the game is playable. We have over 500 commits and 27 classes in src/ within about one month. Thus our performance is really productive.

=> 2p

  * 1.2: The software structure is appropriate, clear and well
documented. e.g. class structure is justified, inheritance used where
appropriate, information hiding is implemented as appropriate. (2p)

Base class variables are mostly protected. Most derived classes use solvely variables of the base class. Some derived class have extra variable like Plane.hpp.

Most derived classes use base class methods by default, but some derived classes like, Artillery, Infantry and Plane might override shooting and movements methods.

Many others independent classes (no inheritance) have member variables in private.
See chapter 5.3 for our class hierarchy.
=> 2p

  * 1.3: Use of external libraries is justified and well documented. (2p)

See Chapter 5.1 Technologies
=> 2p

2. Working practices (0-6p)

* 2.1: Git is used appropriately (e.g., commits are logical and frequent enough, commit logs are descriptive). (2 p)

We have many commits and the commit logs are short and descriptive of the changes made.

=> 2p

* 2.2: Work is distributed and organised well. Everyone contributes to the project and has a relevant role that matches his/her skills. The distribution of roles is described well enough. (2p)

We all had good idea of what was expected of us and everyone contributed to the project.

=> 2p

* 2.3: Quality assurance is appropriate. Implementation is tested comprehensively and those testing principles are well documented. (2p)

We have tested our code by various methods: valgrind, asserts, gdb and test/-folder.

=> 2p

3. Implementation aspects (0-8p)

* 3.1: Building the software is easy and well documented. CMake or such tool is highly recommended. (2p)

We have working automatic building systems, Makefile. Building is simple, just invoke *make*, or *make run* to build run our game. These commands works in src/ and project's root folder.

Documents are generated by Doxygen from project's root folder by *doxygen* command
=> 2p

* 3.2: Memory management is robust, well-organised and coherent. E.g., smart pointers are used where appropriate or RO3/5 is followed. The memory management practices should be documented. (2p)

We use smart pointers and have not used calloc, malloc, realloc nor new statements at all. Therefore, we have not seen our memory leak from Valgrind test.

See chapter 5.4 Memory usage and pointer use.

=> 2p

 * 3.3: C++ standard library is used where appropriate. For example,
containers are used instead of own solutions where it makes sense. (2p)

We have used vector, deque, list, map and set for specific case to store elements.

See chapter 5.5 for our use of containers.
=> 2 p

 * 3.4: Implementation works robustly also in exceptional
situations. E.g., functions can survive invalid inputs and exception
handling is used where appropriate. (2p)

Textbox limits users from inputting arbitrary keys.

We have try catch statements to handle exceptional cases.
=> 2p

4. Project extensiveness (0-10p)

 * Project contains features beyond the minimal requirements: Most of
the projects list additional features which can be implemented for
more points. Teams can also suggest their own custom features, though
they have to be in the scope of the project and approved by the course
assistant who is overseeing the project. (0-10p)

We implemented the following features of the project:
Basics:

- Graphical game with simple graphics
- Simple physics
- Some sort of enemies

General:

- Different kinds of enemy ground troops (infantry, air defence machine gun...)
- AI enemy planes
- Multiplayer (on same keyboard)
- Level editor?

We don’t have
General:

- Hangar/base where friendly airplanes can land & repair possible damage
- Different kinds of weapons (machine gun, bombs to destroy the enemy infantry...) to the airplanes

Advanced:

- Multiplayer over network
- different kinds of missions

=> 8p