Project documentation

Angry Birds group 4

1. Overview of the project

The game is an Aalto themed spin-off on the original Angry Birds game, where instead of birds, the player hurls Aalto students from different guilds. The goal of the game is to destroy the fuksis (first-year-students) hiding in the fort, but without destroying any beer bottles that might be in the level.

In addition to the classic game, there are two game modes. Time trial, where the player is given a time limit that reduces score and loses the game if exceeded, and endless mode where the player can recruit new students by hitting them in the level.

The game also gives the user the ability to create new levels and edit existing ones in a level editor.

2. Software structure

The program depends on two external libraries: SFML and Box2D. The program uses SFML multimedia library for graphics, window and audio. Box2D is used for physics simulation. The main structure is roughly explained in this section, but more detailed documentation is available in an external document generated by doxygen.

The following diagram (figure 1) represents the basic class structure, and ownership between objects in the program. This also means that the lifetime of objects higher on the tree is guaranteed to be longer than the ones below. The program uses smart pointers and RAII principles in memory management. As such, if an object higher on the tree is destroyed, the whole subtree of objects is also destroyed. Random number generation, as well as some functionality like constants and free utility functions are organized into namespaces that can be used globally.

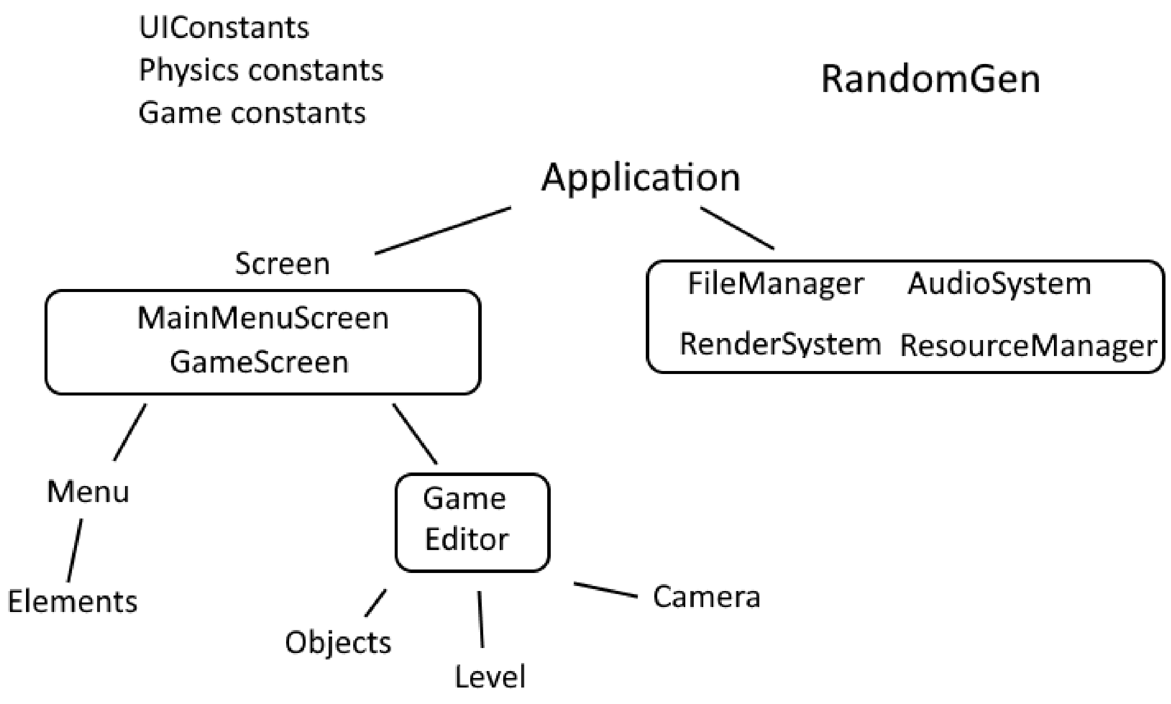


Figure 1, main structure of the application

The top level object is an instance of Application, that uses SFML Window module to create the main GUI window for the program, and handle events. It has an active subclass of Screen (GameScreen or MainMenuScreen), which defines what content is currently shown and active. Application also has two timed loops that are used to give updates for rendering, and physics. The physics loop is framerate independent, and uses a fixed timestep. The rendering loop will try to render at a target frame rate of 180, and releases leftover time for the cpu.

Application also constructs a single instance of a FileManager, AudioSystem, RenderSystem and ResourceManager. These systems abstract underlying SFML code, and offer functionality such as drawing sprites, playing sound effects etc. AudioSystem and RenderSystem depend on the ResourceManager, and the ResourceManager depends on the FileManager.

Here is the basic concept. The program handles resources, like sprites, sound effects and fonts by preloading a single instance of them from a file, which is then stored in the ResourceManager. The physical resource can be accessed by classes like the AudioSystem and RenderSystem through an unique integer ID. Since the resources that the game needs are known at compile time, there can be an enumeration containing all the IDs for all usable sprites and sounds as named constants. In game logic objects such as wooden boxes and such don't directly deal with images, audio clips or file paths, nor do they have to care about that. They instead use the id, which RenderSystem and AudioSystem can recognize and use to identify the physical resource. This way, resources are never loaded twice, and they are guaranteed to exist, and are never leaked. In the case of a missing image file (or similarly audio etc.), the id will instead point to a missing texture image in the ResourceLoader.

Screen is an abstract class, and subclasses of it define what content is shown when the screen is active. This content can come in two categories, Menu content and Game content. A Menu consists of Elements, such as buttons, textboxes, labels and such. These elements have a rendering method that takes a const RenderSystem& as a parameter, and they use the provided functionality to draw themselves accordingly. They also receive updates and events from the parent screen, and have handlers for events like clicking and typing keys. Their position and size is stored as a percentage, in comparison to the total size of the window.

Game on the other hand will construct a Level with data loaded through FileManager, and manage GameObjects within that Level. It has a Camera that can be moved and zoomed according to things that happen in the Level, and can be used to transform objects' positions in Box2D's world space to their corresponding positions in screen space.

GameObjects have a GameObjectType that identifies what type of object they are. This can be used to call the correct subclass constructor, filter out GameObjects (such as looking at all the objects that are Fuksis) and also to get their GameObjectGroup. GameObjectGroup is a more general identifier that gives more general information about the type of a GameObject (GameObjectType::wood_plank vs GameObjectGroup::block) The group also determines the drawing order of GameObjects: objects are drawn in the first group from oldest to newest, after which the next group is drawn and so on.

Elements and GameObjects can respond to input, and get updated regularly, because Application will refresh the active Screen every iteration of its loop, and pass events such as key inputs to the Screen. When a Screen is refreshed, it will update Game and all Elements based on elapsed time since the last update, and ask both to render their content with the provided RenderSystem. Game updates the physics and calls game logic, and renders each GameObject. After this, Menu will ask all of it's components to render.

Since the physics are run at a slower, fixed timestep, the positions are not updated every frame. To give smoother movement, all GameObjects store their previous state at the start of their update, and this is then used when rendering to interpolate positions and rotations between physics updates.

3. Using the software

3.1. Building the project

The project is build by the following steps:

1. Make sure CMake is installed, and added to PATH. You can verify this with \`cmake --version\`. (Installing CMake https://cmake.org/install/). Also make sure SFML dependencies are installed.

2. Navigate to the root directory of the project, or specify it in the next step with \`-S <path-to-dir>\`. Alternatively on a Linux system you can skip all of the following steps by running build_and_run.sh from the project root directory ($ sh ./build_and_run.sh).

3. Generate a build system with CMake, for example \`cmake -S . -B build\`.

Additionally, you may need to provide the preferred generator, for example \`cmake -S . -B build -G "MinGW Makefiles"\` (listed in the CMake documentation in cmake-generators https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html)

(Note that the performance of the application is significantly slower if run in debug mode. If this is an issue, explicitly specify -D CMAKE_BUILD_TYPE=Release, however this should be the default if no parameter is given)

4. Build the project with \`cmake --build build\`

For single-configuration generators like Make, this will build the project in release mode unless specified with \`-D CMAKE_BUILD_TYPE=Debug\`. For generators like Visual Studio, this is instead specified with \`--config\` in step 4. and defaults to \`Debug\`

5. Make sure that when running the application, you run it from the output folder (build/), and not the project folder (angry-birds-group-4/). This way, relative file paths used by the application will point to the correct data/ folder.

(It’s not a huge problem, but levels will get saved to data/ in the project folder, which isn’t originally intended)

SFML dependencies (Linux)

SFML depends on a few other libraries that must be installed.On Windows and macOS, all the required dependencies are provided alongside SFML.

On Linux, make sure to install these before building: freetype, x11, xrandr, udev, opengl, flac, ogg, vorbis, vorbisenc, vorbisfile, openal, pthread

The exact name of the packages may vary from distribution to distribution. Once those packages are installed, don't forget to install their development headers as well.

3.2. Game instructions

The game uses relative paths and in order to function correctly the program must be executed in its local directory, for example launching it in terminal, or using the build_and_run.sh. This will open the application as a 800x600 window. The size can be adjusted, but it isn’t limited so some sizes won’t give meaningful results (such as a 10x10 pixel window, or a 20x600 window).

The application searches all levels available in the levels directory and shows them in the main menu (figure 2). Levels can be played, edited, created and deleted from the side menu bar.

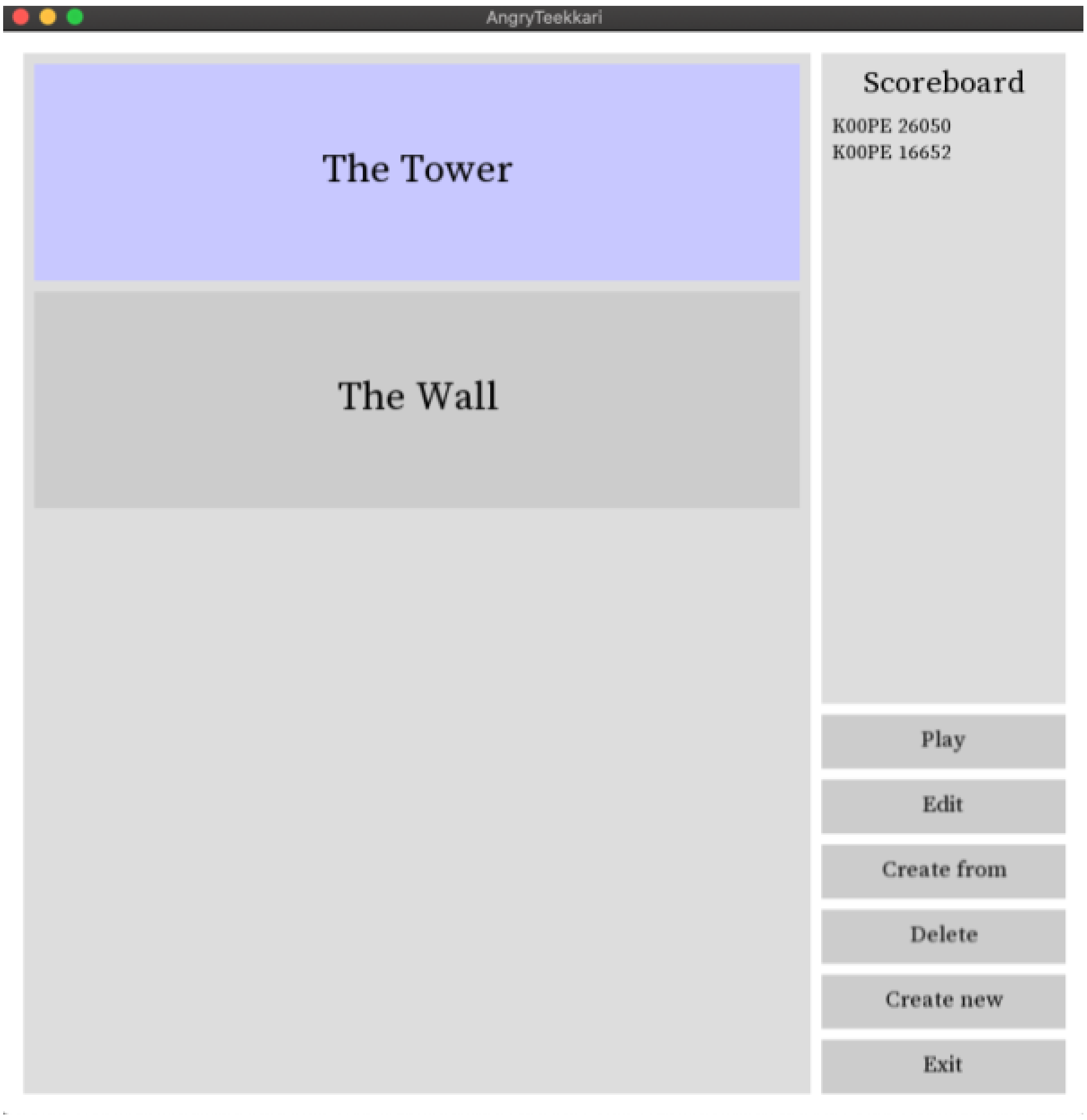


Figure 2, main menu

The gameplay (figure 3) is quite similar to the famous Angry Birds game, but the order of the projectiles can be freely chosen from the side bar by clicking thumbnails of characters. The characters are launched with the mouse by left-clicking the gamearea: Holding down the mouse will aim the cannon, and releasing will launch. The force of the shot is determined by the mouse distance to the cannon.

When there are Teekkaris on the screen, their abilities can be activated by right clicking anywhere. All Teekkaris will use their ability, after which it becomes unusable.

The view can be zoomed in and out by scrolling and the zoomed gamearea can be explored by holding down the middle button of a mouse and dragging the screen.

A level is completed when all Fuksis are destroyed. Conversely, if the player runs out of Teekkaris, the game is lost. Teekkaris that have not been used will give a large amount of extra points, so the player should try to use as few Teekkaris as possible.

There are three different game modes available to play and create levels for. There is the standard one, where the goal is to destroy all fuksis in a level. Time trial is similar, but the time ticks down instead of up. If it goes to zero, the game will end. Lastly there is endless mode, where players keep playing levels endlessly. All the teekkaris acquired from previous levels are preserved for the next ones. Endless levels can give more Teekkaris by offering Teekkari pickups, or by adding starting Teekkaris directly.

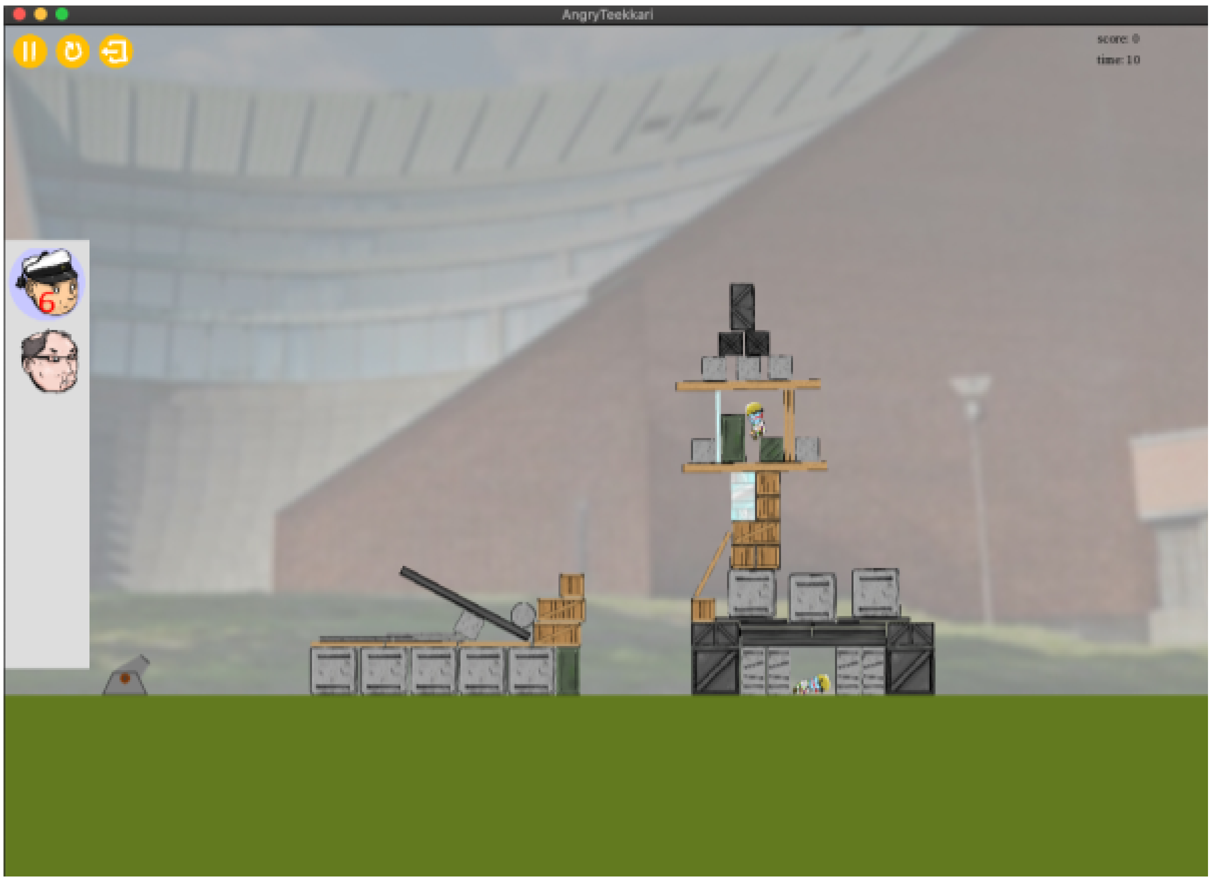


Figure 3, gameplay

In the game editor (figure 4) levels can be fully edited and created. Blocks can be inserted with left-click and removed with right-click. Blocks can also be dragged with a mouse and rotated with arrows up and down when dragging. Objects placed on top of an existing object will be destroyed. Projectiles are removed with a left-click from the list. Also the gamemode can be changed under the level name. Under the gamemode box can be seen max score, that is the theoretical maximum score (if all blocks and fuksis are destroyed with only a single teekkari). Under that, users can set the score limit that is required for maximum points from a stage.

There are six different shapes of blocks that all have six different material variations:

Materials:

Wood

Wood is light, and breaks rather easily. The largest block gives 400 points.

Metal

Metal is extremely durable, but susceptible to electricity. It gives the least points of all materials.

Glass

Glass is brittle, but heavy and hard to penetrate. It gives the highest points.

Plastic

Slightly lighter and more durable than wood. Gives slightly more points.

Rubber

Rubber is light and bouncy, but not very durable. It gives low points.

Concrete

Concrete is the heaviest material, but not as durable as metal. Hitting concrete is very damaging for a teekkari.

On top of that, there are beer bottles that deducts score when broken, tnt blocks that explode when destroyed, pickups that give you a new teekkari to use when hit and fuksi that is used as an ingame goal.

Teekkaris that are available in a level are chosen from the bottom of the object list. All teekkaris have different abilities that are:

IK teekkari

IK teekkari knows all the weaknesses of materials, and deals more damage to all block types. He has high penetration and damage against lighter materials.

SIK teekkari

This teekkari can zap metal blocks and destroy them. After a short charge, electricity will jump to the nearest metal block, and deal large damage to connected metal.

TEFY teekkari

This teekkari creates a gravity well that pulls nearby objects towards itself.

Prodeko teekkari

This teekkari “innovates”, dealing damage over time to blocks around him. The air pressure from his spinning hands creates lift, and pushes nearby objects around.

TIK teekkari

TIK teekkari causes a small bug, teleporting to the right. He can bypass obstacles, and hit hard to reach targets.

Inkubio teekkari

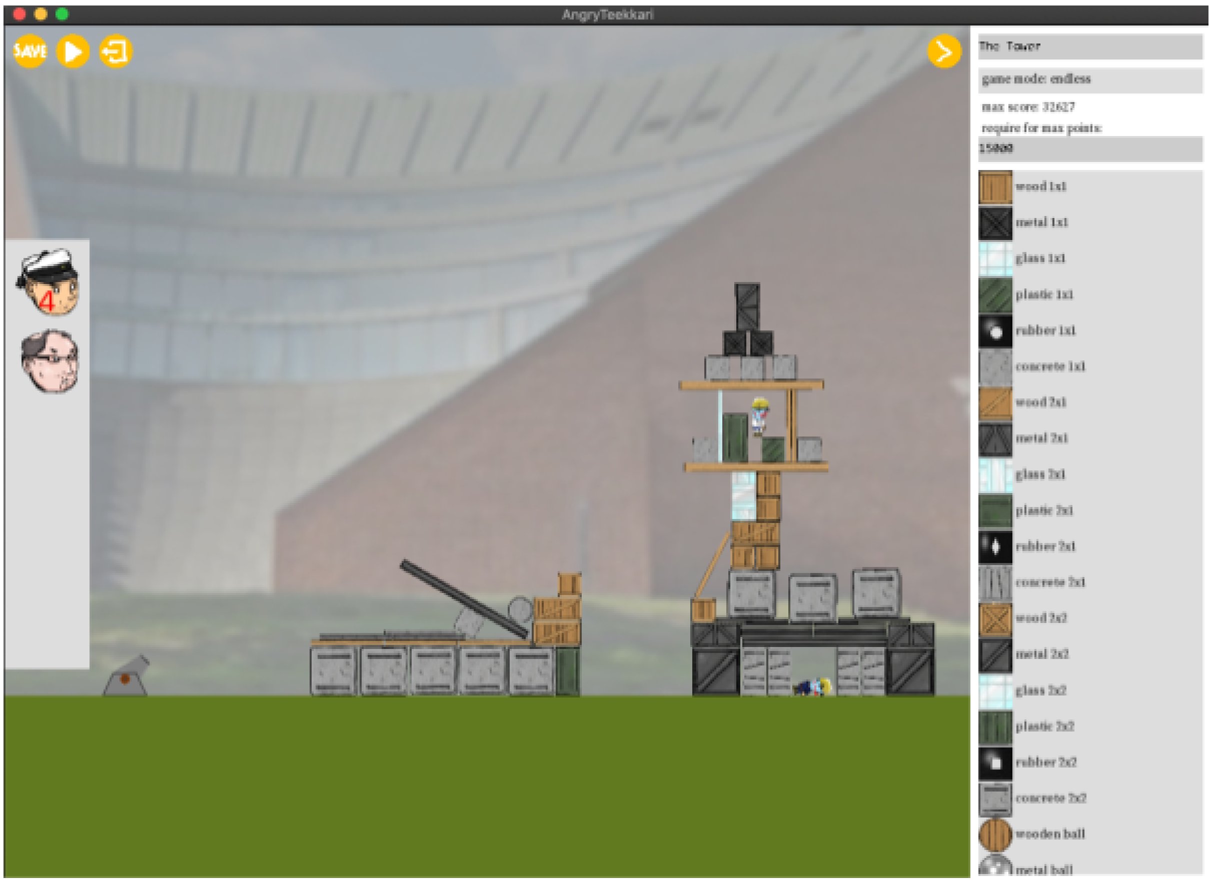
This teekkari spawns a strong cow that falls straight down from this teekkari’s location.

KIK teekkari

This teekkari throws three wrenches towards the clicked position. The wrenches deal extra damage to wood, but can only destroy one block each.

The Professor

Professor integrates over the map, causing massive destruction in its path.



The Tower

game mode: endless

max score: 32627

require for max points:

15000

Figure 4, Game editor

4. Testing

Valgrind was used for debugging memory and memory leaks. Gameplay was first tested with a hardcoded testfile, TestFile.hpp, which can be found in tests directory. Some manual tests were done during the development.

5. Worklog

Tommi Korpelainen

1st Week (45): Writing the project plan document, writing the basic structure of the application and creating an UML class diagram. **(3 hours)**

2nd Week (46): Implementing framework and overall structure. **(48 hours)**

3rd Week (47): Implemented simple box2d physics (breakable boxes that fall). **(38 hours)**

4th Week (48): Implemented basic teekkari ragdoll, which will be inherited by fuksis and special teekkaris. **(60 hours)**

5th Week (49): Basically finished Fuksis, Game ending, Teekkari abilities and some effects. **(38 hours)**

Aleksi Rintanen

1st Week (45): Writing the project plan document. **(2 hours)**

2nd Week (46): Implementing Game class and basic functionality. **(3 hours)**

3rd Week (47): Created test level, object id grouping and Game methods. **(5 hours)**

4th Week (48): Created the cannon. **(13 hours)**

5th Week (49): Restricting camera bounds, changes to Game, adding some methods to RenderSystem, implementing Editor, implementing teekkari abilities **(26 hours)**

Ilari Tulkki

1st Week (45): Writing the project plan document and discussing about some implementation details with Tommi. **(6 hours)**

2nd Week (46): Creating meeting notes and begun implementing UI. **(6 hours)**

3rd Week (47): Created menu and more UI elements. **(32 hours)**

4th Week (48): Created even more fancy UI elements. **(43 hours)**

5th Week (49): Finished GameScreen, EditorScreen and other UI. **(32 hours)**

Joonas Palmulaakso

1st Week (45): Writing the project plan document. **(2 hours)**

2nd Week (46): Implementing GameObject class and its subclasses. **(5 hours)**

3rd Week (47): Implemented GameObject methods, but had problems with git and the work was unfortunately lost. **(4 hours)**

4th Week (48): Implemented loading and saving levels. **(10 hours)**

5th Week (49): Changes to file saving and loading, animated effects, explosion force to physics effects. **(12 hours)**