Laser Squad

Project Documentation

Members

Kristian Salo, 481014

Katariina Korhonen, 526021

Heikki Jääskeläinen, 525763

Joonas Pulkkinen, 527347

Instructor

Pasi Sarolahti

Course

ELEC-A7150

C++ -programming

1. Overview

Laser squad is a turn-based tactics video game originally released in the 1990s. It has several different missions where the player tries to complete objectives such as eliminating the enemies or rescuing a hostage. A player has a team of characters, which can perform move, shoot, pick up or turn operations, thus using the player's action points. The characters are controlled individually.

The game implemented in this project is largely similar, yet simplified version of the original laser squad game. The game only has one possible mission – to eliminate the enemy boss – and less diversity and customizability in terms of characters, weapons and items. However, if one wishes to extend the functionality of the game, it should be easy since the game is programmed modularly to be easily extendable.

The fundamental gameplay mechanics of the implemented game are very analogous the original laser squad game. The players take turns in controlling their team of characters. A turn lasts until all characters are out of action points, or the player chooses to end the turn voluntarily. After each turn, all characters receive a certain amount of new action points. The game ends when the other team's boss character is eliminated.

The basic functional unit of the implemented game is a character. The game has three different characters: scouts, soldiers and bosses, which all have different properties. These properties include HP, action points, speed, an active weapon and an inventory of items. The inventory of items can include other guns, ammunition for them or food to recover lost HP. Furthermore, three different weapons are available: machine guns, rifles and pistols.

A character can perform five types of basic activities. Moving and shooting are the two principal activities that consume a character's action points. These are implemented with the help of pathfinding and line of sight algorithms. Therefore, a character can only move to cells that it has an open path to, and shoot enemies that are within the line of sight of the character. Moreover, a player can only see enemy characters that are within the line of sight of the player's characters. The other three activities are related to items: a character can pick up, drop or use items as desired.

The implemented game also includes a map editor in the form of a text file. When the game is started, the contents of the text file, GameLevels.txt, are loaded into the game, and the user has a chance to choose which of the maps he wants to play on. Therefore, a player can create new or modify existing maps simply by controlling the contents of a text file. The maps in the text file have include information about the maps structure, i.e. walls and open spaces, as well as character positions for both teams and item pick-up spots.

2. Software structure

2.1 Class structure, relationships and responsibilities

The program is implemented in a modular structure, i.e. different functionalities are separated into different classes. This should allow the program to be easily expanded, if additional features are to be added later. The figure below presents the final class structure in a UML-style diagram. Only the essential methods and fields are presented to abstract away smaller details.

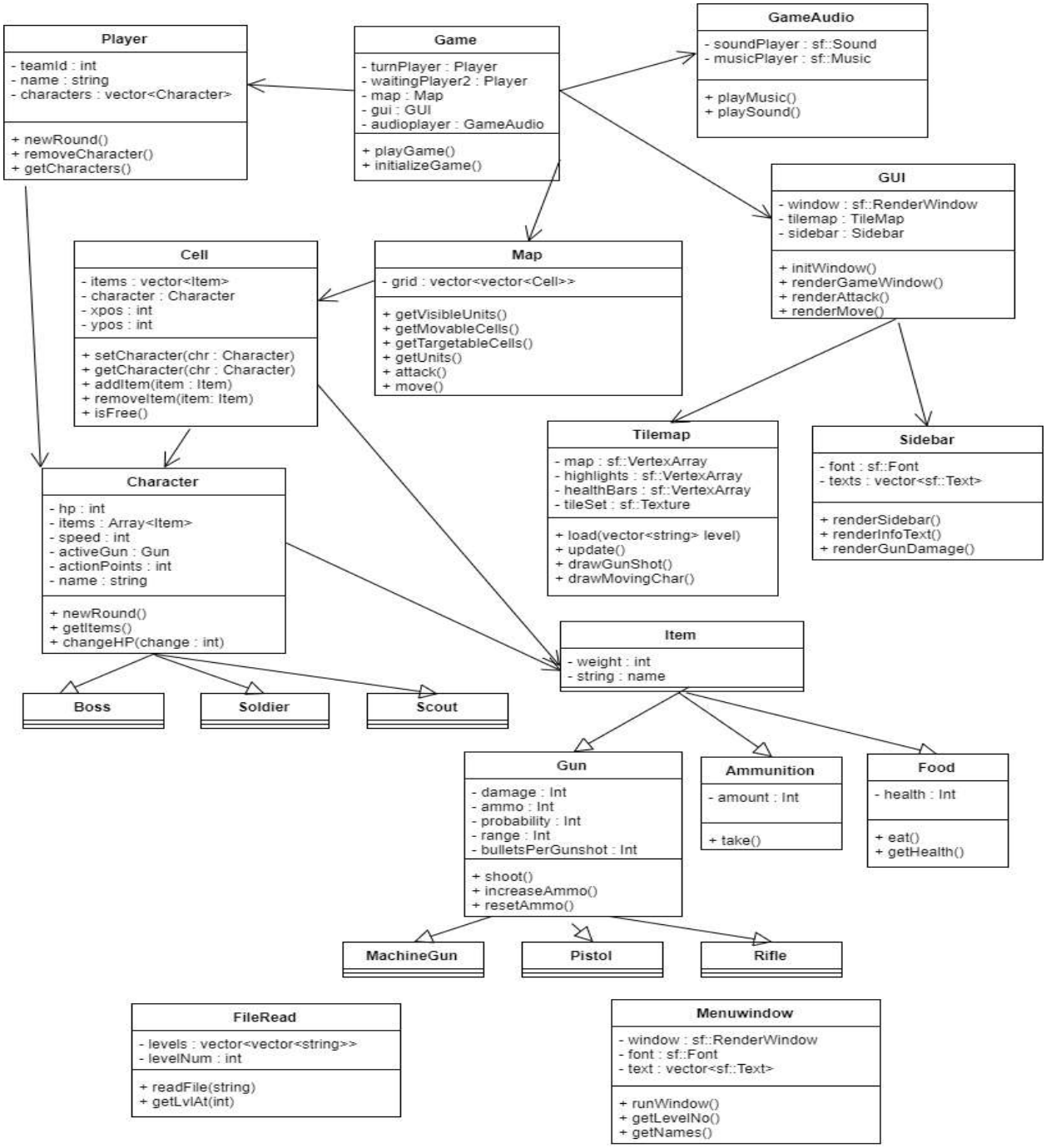

Figure 1. An UML-diagram of the class relationships in the Laser Squad game.

- main.cpp

  This is the entry point for the application. It is not shown in Figure 1, since it is only a brief function that calls other classes in the correct order: FileRead to read game level data from the text file, Menuwindow to get player names and map selection and Game class' initialize and play methods. Also, the exception handling and program shut down in such situations is handled here.

- Game

  The game class is the top-level logic for our game, it takes care of initializing, running and ending the game. It has instances of a map, 2 players, GameAudio and a GUI class that is used to draw the game. Game will keep track of the current players' turns and respond to events, such as mouse clicks with appropriate functions from classes character, cell and map.

- Player

  Each Player has their own instance of the Player class, that keeps hold of their name and the characters (units) they have at their disposal It also has a variable that tells on which team they are on.

- GUI

  The GUI class draws what Game class tells it to the screen. It has 2 private members, Tilemap and SideBar, that it uses to draw the map and the text in the sidebar, respectively. It has functions for initializing the drawing, updating every frame, and for drawing movement and attacks

- TileMap

  Tilemap gets textures from the sprite sheet and has functions for drawing everything game-related to the board. It does this using sfml-library and it's draw functions. TileMap contains 4 VertexArrays that it refreshes in order to get the wanted picture on the screen. Drawing occurs when either Update, DrawGunshot or DrawMovingChar gets called on a higher level (GUI).

- Sidebar

  Another drawing related class, used for rendering text. This mainly concerns drawing the sidebar, but also the info screen, game end screen and gunshot damage.

- Map

  The Map class instance represents the playing field of the game. It holds a 2-dimensional array of cells, that in turn hold all the necessary info on the current tile of the map. Map-class will also have functions that enable characters to act on the map. Move-

function allows a character to move on the board according to its speed and remaining action points. Cells the character can move to are mapped by getMovableCells, that returns a map which keys and values consists of cells that can be moved to, and the paths leading up to them. It uses a recurssion to achieve said result. Attack lets a character attack a target in its range. It gets all its needed information to shoot from the characters gun class. GetTargetableCells returns every enemy character in range. Map also has getVisibleUnits function that can be used to get return all the characters one player sees during his turn.

- Cell

  A cell is used to hold all the information about the point in a map, such as items and characters in it. It has functions to add or remove them.

- Character

  Character is a virtual class used to portray units on the map. All characters have common variables such as speed, health actionpoints and inventory. Its children; scout, soldier and boss, all have different initial and max stats. Characters have functions to remove and add items to their inventory. They also have a changeHp-function that allows the character to heal, take damage and die. NewRound is used to give a character a new set of actionpoints.

- Item

  A virtual class that has at least 3 different children: Gun, ammunition and food. They all have different uses but they are kept under the same parent class to keep other classes tidier.

- Menuwindow

  Draws the menu screen that shows up at the start of the game. Gets level number and players name from the user.

- Fileread

  Reads a text file and creates vectors of strings from each of the levels found in it. After that they can be returned by getLvlAt function.

- GameAudio

  A simple class for easier use of sfml library's sound functions. Reads sound files to memory so that they can later be used only by calling playSound function with the sound name as a parameter. Music is handled similarly.

2.2 Implementation details

Pathfinding in the game is implemented recursively, it starts from the selected characters position and advances to all its neighboring squares and so on. It creates a map that it updates every time it arrives in a new cell or when its path to the cell is shorter than the previous path. The recursion ends when it has advanced characters action points * speed number of cells, which is the maximum movable distance of a character. The end result is a map that can be easily used to poll if a character can move to a cell. If it can, the map returns a path to said cell.

Line of sight is implemented using Bresenham's line algorithm, simply just iterating over the cells on the line between start and end point. If it finds a character or a wall vision is blocked. The game also implements field of vision. It is merely an extension of line of sight. Since player can see all of his own characters but only enemies in vision, we have to cast lineOfSight from an enemy character's position against all friendly characters. If none can see it, it is hidden. This is then repeated for all enemy characters.

The game utilizes mostly vector containers for storing runtime data and iterators for iterating them. For example, the grid variable in Map is a 2D vector of Cell objects. Also, each character has a vector of items and each player has a vector of characters. Vector was chosen since it provides fast random access in *O(1)* and fast insertion and removal of items to the end in *O(1).* These are the operations that are mostly used in the game: the elements in all the vectors mentioned above are randomly accessed many times when the software is run. The slowest operations for the vector container, insertion and removal in other places than the end, are only occasionally used for the smaller containers. Thus, vector was the best container choice in terms of performance.

The graphics rendering was implemented with the help of built-in VertexArray class of the SFML library. This was chosen to reduce the number of calls to the draw() function. The other option would have been to separately draw each tile block and character to the screen every frame, which would be very taxing on the GPU. With the VertexArray implementation, only a few draw() calls are needed, since a single VertexArray includes information about the whole drawable background or all drawable characters.

The game uses shared smart pointers for all resources that are dynamically allocated (in the heap) at runtime such as characters and items. Only few traditional pointers are used, and they always point to resources stored in class variables (in the stack), such as Cell objects stored in the Map class’s variable grid. Since memory management is handled by smart pointers, rule of three didn’t need to be explicitly implemented. However, all classes that contain dynamically allocated resources have a copy constructor, assignment operator and a destructor to demonstrate that RO3 is understood. Copy constructors and assignment operators simply copy all variables and destructors are left empty, because of using smart pointers. Such classes include for example Player, Map, Cell and Game.

The game is implemented in an object-oriented style and thus utilizes virtual functions and dynamic binding. For example, the character class has virtual functions for dropping items and

changing HP. Also, dynamic binding is used in the Player class' array of characters that can contain soldiers, scouts and bosses, and use each child class' own implementation of the virtual functions.

The exception handling in the game is mostly related to file opening situations. These include opening GameLevels.txt file, sprite images and fonts. If any of these files is not found, an exception with a descriptive message is thrown. These exceptions are caught in the main function, and result in the info message being printed to standard output and game shutdown. These are the only situations in which the program execution is ended. In other exceptional situations, e.g. if a dynamic cast fails or an audio file is not found, the functionality is simply not executed, and the program continues to run normally, since these exceptions are not crucial to the program. However, an info message is printed to the standard output.

3. Instructions for building and using the software

The program uses the SFML library for the user interface and sounds for the game.

The program can be compiled easily with CMake. `CMakeLists.txt` in the `src/` folder contains the script for generating the makefile. `cmake_modules` folder contains the file `FindSFML.cmake`, which is a helper file for `CMakeLists.txt` for finding the SFML sources.

CMake was tested to function in Aalto Linux machines. The procedure for an out-of-place build using it is:

- go to git/laser-squad-1/ directory

- create a new directory for the build

- navigate to the build directory

- run cmake from the build directory

- generate the executable

- run the executable

```

mkdir build

cd build

cmake ../src/

make

./laser-squad_ex

```

When you run the executable, the game should show you a starting menu screen. You need to choose names for Players 1 and 2, and also the Map number you want to play. Player names only accept letters, and the Map takes numbers. Move between the choices using ***arrow keys UP and DOWN*** or ***ENTER***, then finally select Start game at the bottom and press ***ENTER*** to start the game.

3.1 Gameplay

In the game, press ***i*** to open and close the info screen. The info screen tells you the important commands for interacting with the level, such as picking up and dropping items. Scroll the map with ***arrow keys*** and move/select your units with ***Left mouse click***, and attack enemy units with ***Right mouse click***. Press ***ENTER*** to pass the turn to the other player. Both players have one ***Boss*** character. The goal of the game is to defeat the other player's ***Boss*** character. The game ends when the ***Boss*** character is defeated.

Different characters and items have different properties. For example, ***Soldier*** character has 10 action points, or AP to use, 20 hitpoints or HP, a speed value of 2, and is wielding a Rifle weapon by default. Moving and attacking consumes AP, and the player can choose to save up their AP for next turn, when they receive more AP (but not above a certain maximum).

Characters can't walk through walls or on top of other characters, but they can walk on a treasure chest, which will contain different items. Picked items go to the character's inventory, where the character can use the item or drop it. Defeated character units will also drop their loot on the ground. Food items heal the character, while guns and ammunition are used for shooting. Ammunition is loaded up the character's active weapon on use, and will be depleted upon shooting enemies. Different weapons also have different properties, for example rifle has a range of 5 squares and does 3 hitpoints of damage, but it has 80% chance to hit. Character can also change the active weapon to a different one from its inventory.

Figure 2. An example starting position of the game.

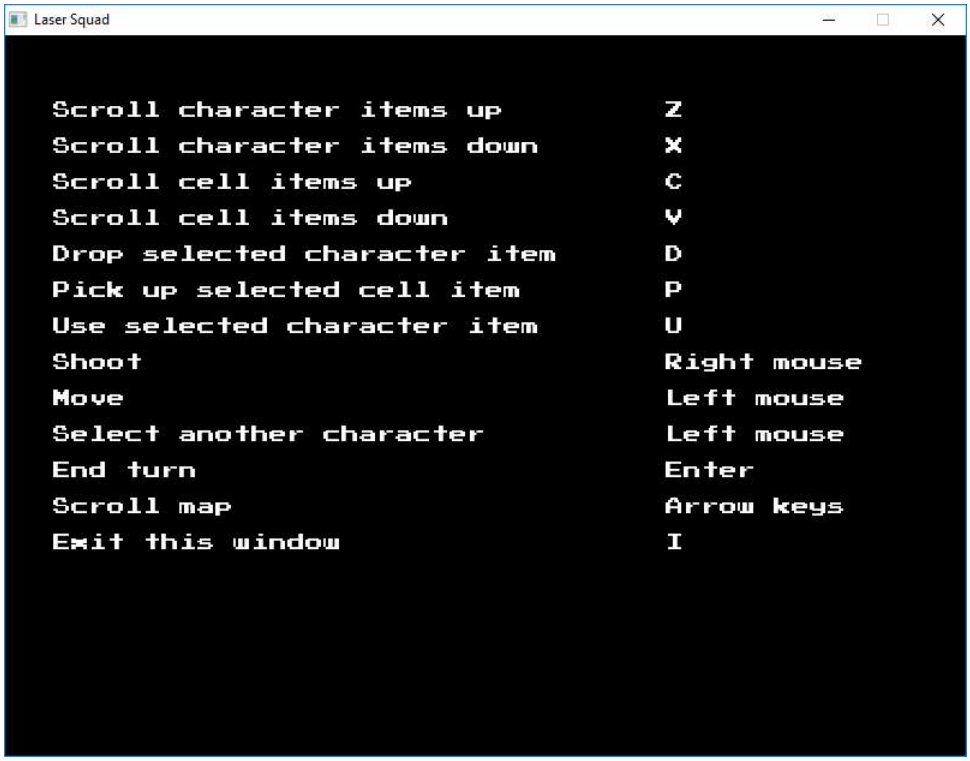

Figure 3. Info screen showing other functions of the game.

3.2 Customizing maps

The starting position of the game level is loaded from a file called GameLevels.txt, so levels can be edited with any text-editing software. Modifying this file will change the layout of the map, and can be used to create new levels. A valid map must be of a rectangular shape and surrounded by walls (# character) on all sides, and must have exactly one boss character for both players (B and b characters). Otherwise, you are free to change how the level looks like. Empty space can be denoted with a dot (. character), but any character not used for other purposes can be used. A level must be ended with a blank line to denote that the level is finished. Figure 4 demonstrated how a level looks like.

```
; Real level 1: 40x40
########################################
#........#.APF..................#...F.#
#.S.B.S..#.AMF..................#....F#
#........##########################.#####.###
#..C.C...#.....................#.............#
#........#.......#......#.....#.....#.....######
#................#......#.....#..#..#.F.#
#................#......#.....#..#..#...#
############.....#R.....#.....#..#..#...#
#........#.......#F.....#.AA.#..#..#...#
#..F..P..#.......###..##########..#..#...#
#........#.....................................#
#.............########################
#........#.....................................#
############.......#######.....#####.#..#..#
#................#....#.....#RA#.#..#..#
#F.........#.....#....###...#..#.#..#..#
#A.........#...###.....#.....#..#.#..#..#
#..........#.....#....#.....#..#.#..#..#
#####.#######.....#######.....#..#.#RA#..#
#.........A#.................#..#.#####..#
#.........F#.....................#
#.....#######.......##############.......#
#..................##############.......#
########............#.....................#
#.....................#.....................#
#.....######.........#..........#######...#
#........#.........#..........#.........#
#.F.....A.#.....................#.........#
#........#.....................#.........#
#####...##########...#.........#..P..A..#
#.........#.R#.......#.........##########
#....#....#..#.A.....#.....................#
#....#....#..#.......#.....................#
#....#....#..#....F..#.....#.....#.....#
#....#....#..##########.....#...c.c..#
#....#.FF.#...#.......#.....#.....#
#.MP.#.AA.#...#..#...#.....#..s.b.s.#
#....#....#.F....#.............#.....#
########################################
```

Figure 4. An example of a game level from GameLevels.txt. The meanings of the characters are further specified in the text file.

4. Testing

File Laser_Squad_testi2 in tests folder in git is created for testing the methods in Character and Item classes and in their subclasses. The test file does not call any functions outside the classes Character, Item or their child classes and it is not used in the actual game. At the beginning of the file, all the tested functions are listed, and the mark 'x' after the function name indicates that the test was successful. Different situations have been attempted to take into account while creating test. For instance, it is good to test that the pure virtual function changeHP() does not change hp under value zero if hp is smaller than the change value.

The test named main is used to test FileRead, Map and Cell. Testing was basically done visually, comparing the seen state of the map to those that they should be. Functions such as readFile, getLvlAt, getMovableCells, and line of sight were all tested.

Probably the most important part of testing the game was playing it throughout the whole development process. This meant testing the different functionalities such as movement, pick up, drop, item use, attack, switching turns, scrolling the map etc. Also, strange and unexpected actions were tried: clicking outside the map, attacking non-attackable cells, moving to cells that should not be movable, loading invalid game level text files, typing invalid names or map numbers. This exposed many errors that were the fixed and also reassured that the game works as expected in normal and unexpected situations.

Valgrind was utilized to test the memory handling of the final game. It was run in Aalto University's Linux machines with the following command line:

valgrind –leak-check=full -v ./laser-squad_ex

The output of the test was expected: the definitely lost, indirectly lost and possibly lost blocks all had 0 bytes. However, some blocks were present in the still reachable part. Since the program utilized smart pointers, which automatically release their memory when they go out of scope, this leak was deduced to be because of something else than the program code. One possible solution was the usage of the c++ standard libraries or SFML. As stated in Valgrind's site, section 4.1: "Many implementations of the C++ standard libraries use their own memory pool allocators. Memory for quite a number of destructed objects is not immediately freed and given back to the OS, but kept in the pool(s) for later re-use. The fact that the pools are not freed at the exit of the program cause Valgrind to report this memory as still reachable." Therefore, this leak was not considered a problem.

5. Work log

5.1 Distribution of Work

The class implementations of the UML diagram presented in Section 2 are divided for each group member so that the workload would be distributed as evenly as possible. The strengths of each group member have also been taken into consideration when deciding the distribution of work.

The distribution of work among the group is the following:

Heikki: Map, Cell, GameAudio

Joonas: Game, Filereader, GameLevels.txt

Katariina: Character, Item

Kristian: GUI, Game, Player, Tilemap, Sidebar, Menuwindow

5.2 Weekly work report

As can be seen from the weekly work reports, the group met quite a few times. Therefore, issue tracker was not used as extensively as it might have been used had the group met less. The meetings enabled the group to discuss bugs and assign them to correct developers.

Week 44

- Everyone: Project subject was decided and the group formed. (2h)

Week 45

- Everyone: First group meeting (division of work, project planning) (2h)
- Kristian: Wrote the project plan, read the SFML documentation, installed it and created the first drafts of classes Game, Player and GUI. (10h)
- Katariina: Wrote the project plan, created the first drafts of classes Character and Item and their subclasses Soldier, Scout, Boss, Ammunition, Food and Gun. (7h)
- Joonas: Wrote the project plan, implemented first version of FileRead class functions and created GameLevels.txt file. (12h)
- Heikki: Wrote the project plan, and planned implementing classes Cell and Map (4h)

Week 46

- Kristian: Continued working with classes Game, Player and GUI. Created GUI helper classes Tilemap and Sidebar. Created a draft version of the GUI without any connections to the internal game logic. (10h)
- Katariina: Continued working with classes Character, Item and their subclasses. Created a separate main function for testing purposes (not going to be a part of the final program implementation). Started testing and debugging of methods. (10h)
- Joonas: Fixed the problems with FileRead class and tested its functions. (6h)

- Heikki: Started working on Cell and Map, made first iterations of the classes and pathfinding. (8h)

Week 47

- Everyone: Second group meeting (presentations of individual process, planning, connecting the classes together). (3h)
- Kristian: Worked on class sidebar that displays player, cell and character information. Worked on neatly displaying the text. Created CMakeText file and tested the project for the first time in Aalto Linux. (6h)
- Katariina: Added three new child classes for Gun: Pistol, Riffle and MachineGun. Continued working with classes Character, Item and their subclasses. (5h)
- Joonas: Removed unnecessary functions from FileRead, as they were transferred to the Map class, designed a testing level and first real level for the game. (6h)
- Heikki: Developed Cell and Map classes further, added first versions of line of sight and field of vision (8h)

Week 48

- Everyone: Mid-term meeting. After that a short group meeting (planning what to do next, decided a day for the fourth group meeting next week). (1h)
- Kristian: Connected the dummy GUI to Map, thus enabling the rendering of the map loaded from GameLevels.txt. Created an info screen. Added the ability to scroll through player and cell items in the sidebar. (10h)
- Katariina: Continued working with classes Character, Item and their subclasses. Added some new variables and methods (6h)
- Joonas: Evaluating other group's work, writing some project report. Bug fixing and debugging. (4h)
- Heikki: Got line of sight and pathfinding working. Made constructor function for Map so it can be properly initialized from ReadFile. Also made a separate test class main for testing reading file, line of sight, field of vision and pathifinding. Also changed most of pointer to shared pointers. (12h)

Week 49

- Everyone: Group meeting (agreed on last modifications and bugfixes, schedule, division of work for the project plan). (2h)
- Kristian: Improved the GUI: highlighting of movable cells, highlighting of active character, different sprites for different characters. Worked on character actions and implemented pickup, drop, shoot and move to Game. Implemented game logic: switching turn, consuming action points, game ending conditions and screen to display the winner. Refactored class Game to be more readable. (14h)
- Joonas: Evaluating other group's work, debugging, diagonal scrolling, boss exception check, README.md file. (11h)

- Heikki: Made constructMap add scout and boss units properly, made it also spawn items from text files. Modified getMovableCells, implemented getTargetableCells and attack. Fixed bugs in lineOfSight, Sidebar and Player. Created class GameAudio and added sounds to the game. Cleaned up Map and Cell with typedefs. (12h)

Week 50

- Everyone: Writing the project documentation. (3h)
- Kristian: Created move animation and shoot animation. Created a menu screen. Added highlighting of targetable enemies. Cleaned the code and did thorough testing in Aalto Linux. Fixed small errors in all classes the caused warnings in g++. Added exception handling and RO3. Modified CMake to work with game audio and copy include files. (14h)
- Joonas: Debugging, writing documentation, walls exception checking. (6h)
- Heikki: Made menu screen advanceable with enter key. Added updating health bars which meant that character classes need to be changed. Made characters fade when out of action points. Cleaned up the sprite sheet to get better visuals in game, which also meant editing drawing of players and moving them to a new vertex array. Fixed final bugs in line of sight and action point cost for moving. Added rule of three for Map and Cell. (15h)