

# Micro machines 1 – Documentation

Antti Nousiainen, Oskari Järvinen, Anton Sulkava, Juho Aalto

## Overview

In this project we have created a racing game like micro machines and death rally. We have agreed on an initial scope for the features to be implemented. This game includes all the basic features, we planned beforehand. In addition, we have also included some of the additional features, which will make the gaming experience even richer and versatile.

The game has multiple different features:

- Artificial Intelligence players
- Supports multiplayer through a server
- Has tar spills and speed boosts
- Has weapons used to kill other players
- 2-dimensional acceleration AKA drifting
- Collisions
- Menus & other UI elements
- Multiple maps
- Music and sound effects
- Software for creating new maps
- Software for creating AI
- Software for creating new tile types for the game
- Unit tests

The game supports Linux and Windows and cross-platform play is possible. Our project also includes multiple tools for making collisions, maps and AI paths. These tools are included in the project and make developing multiple maps, different kinds of AI and tiles easier and painless.

## Software structure

The software can be divided to two main parts, the game itself and the server software. Both are integral for running the script, thus discussed in detail in this section.

The game's structure can be described as follows: The game relies on a  $X*Y$  sized grid. This grid contains cells that are indexed on (x,y) coordinates. These cells contain an ID value, which is equivalent to specific tile type. These tiles contain a collision matrix which is used to determine what is and what is not driveable. Additionally, the top two bits of tile ID is used in checkpoint detection and the whole ID is used for determining what sprite is drawn.

An overhead class for the Grid class is the Map class. This class is used to retrieve information from files and generating a grid from the data we receive from a map file. Additionally the map class stores information regarding all the different tile types used in the map and information regarding which music file is played while the map is playing.

The map class is passed to the class called “Game” this class stores relevant information regarding the game and its state. For example, the location of tar spills, boosts, networking and players. The Game class is basically used to run the game and therefore is quite big and complex. The Game class takes in a Map, a Player and a pointer to a network connection (local or external).

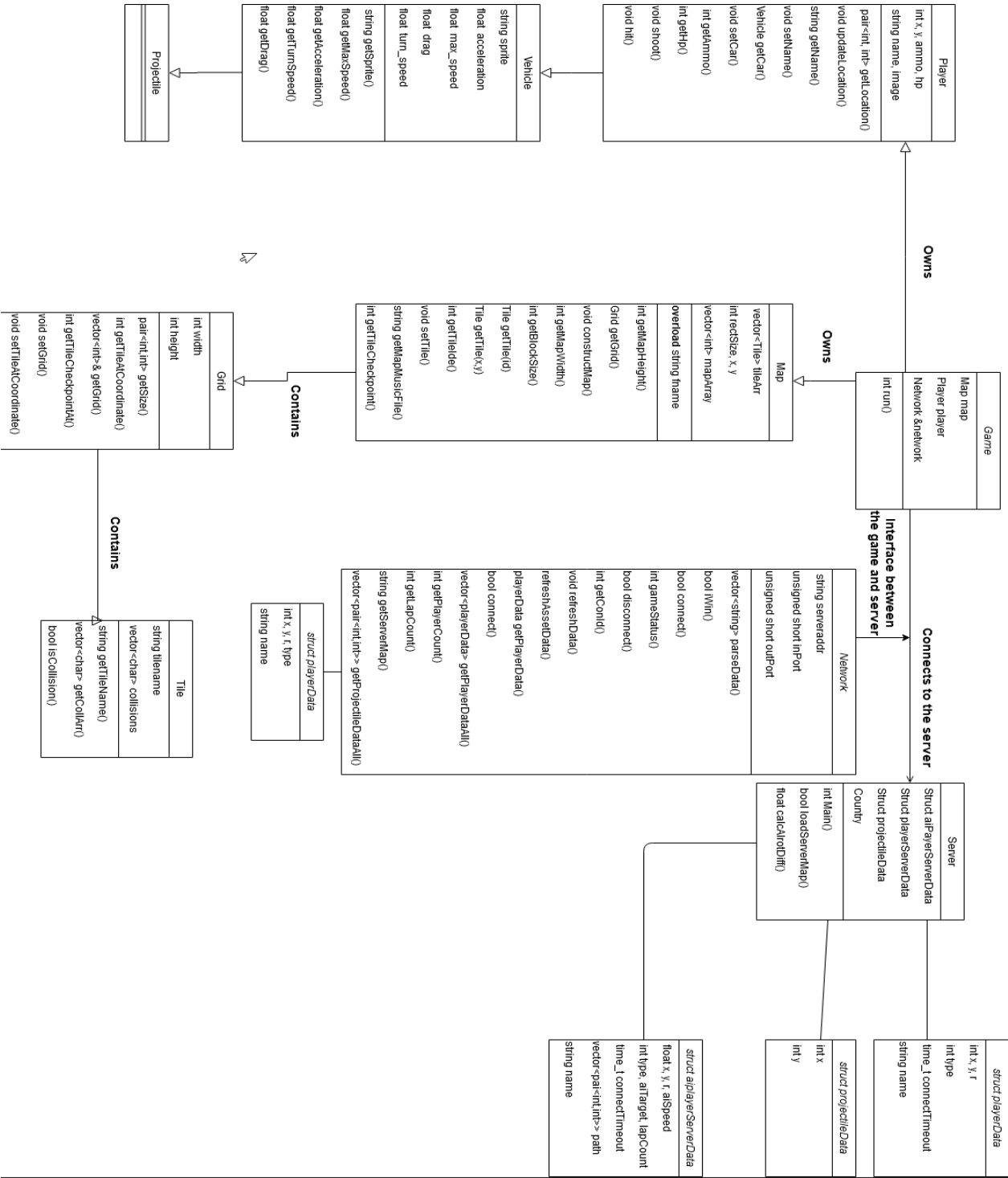
The local player and other players in the network are handled through the standard player class. The player class stores a vehicle class and information regarding ammo and health amount of the player. The vehicle class incidentally stores information regarding the vehicle, such as drag and acceleration values.

The network class is used to bridge the local player between the server that is run while playing. While being part of the game structure, the main use of the network class is to connect the player with the server. The other function of network class is to pass and receive information from the server and serving it to the game.

The server software is a separate piece of software that must be started in the background if the user wants to play locally. Otherwise it is possible to connect to a server over the network. The server – client communication uses UDP in its data transfer. The server structure is as follows: When the server is started, it first loads a server mapfile, which contains the map filename and all AI paths for the corresponding client mapfile the client needs for connecting to the server. After this the server enters a loop used to reset the server after a game ends. Within this loop there is another loop that is the core of the server. This loop handles incoming and outgoing requests, all server command handling as well as updates the locations for all AI players present. The client software is dependent on the server, and can not function without it.

The final piece is an additional file used to run tests. This is run manually and is used after implementing features.

The overall class relations of the game and server are given in the diagram below:



Two external libraries are used in this project: first one being SFML and the other one being Catch2. SFML is used to handle everything from network to audio. We use the SFML internal libraries to build menus, draw graphics, handle socket etc. and play sound effects and music.

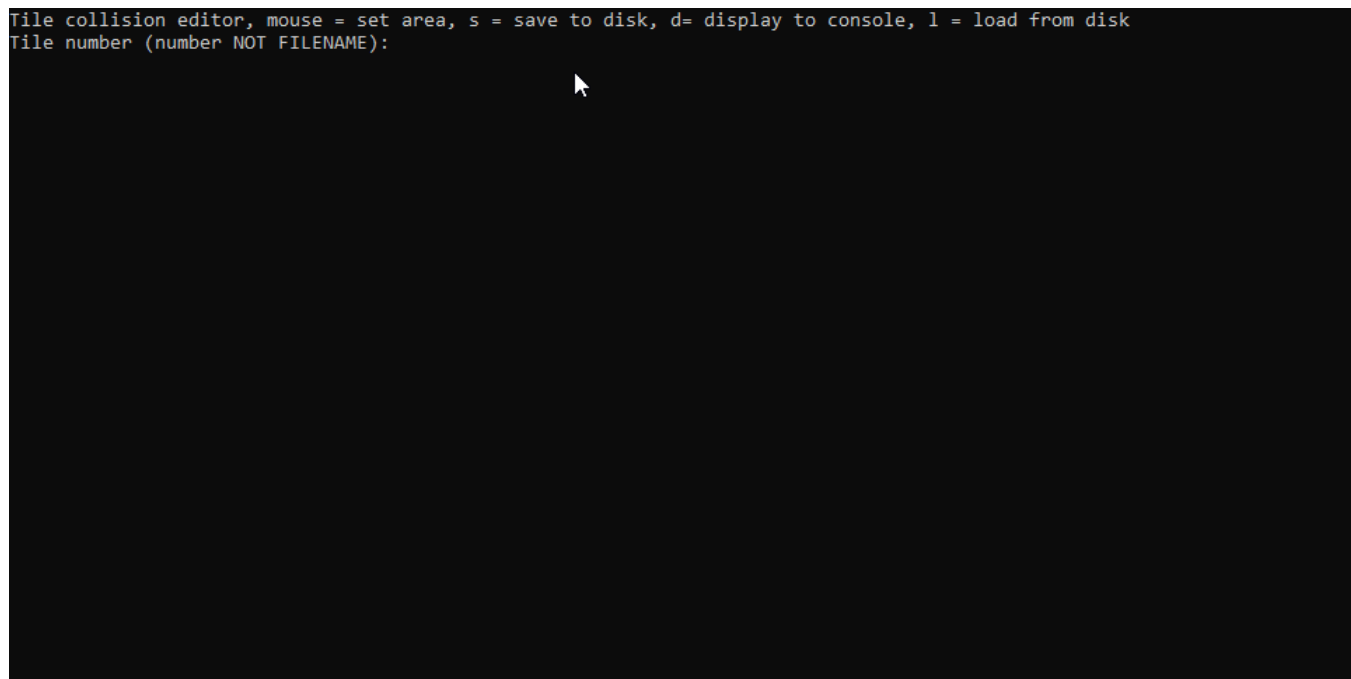
The other external library is Catch2. This is used in unit testing. The library provides easy way to generate tests and receiving information about issues etc. after changing implementations/ the program. These tests are not run automatically after implementation change, but rely on the programmer to be run.

## Tools

The project comes with three different tools used for creating new immersive and awesome maps for the game. These tools are used to build new tile collisions, map layouts and AI paths. To compile the tools, on windows run the “buildToolsWin.bat” and on linux “buildTools.sh”.

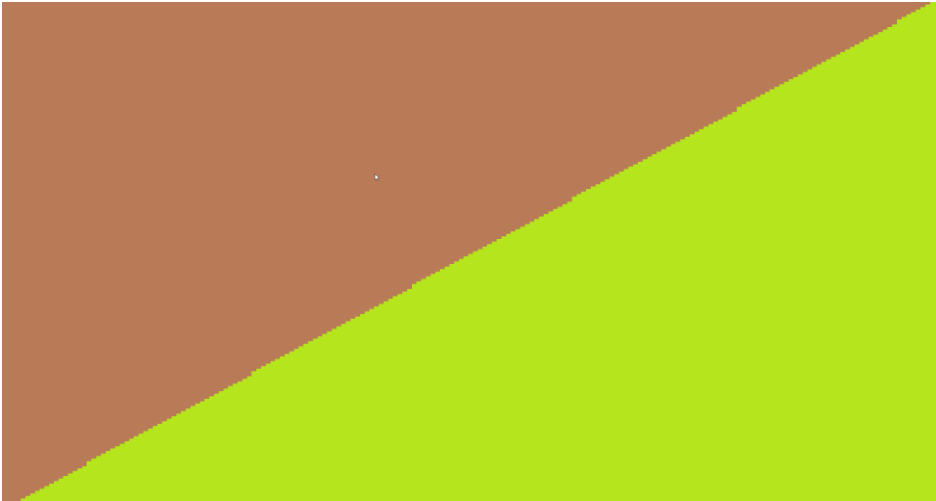
## The tile editor

The tile editor is used to generate new \*.dat files. These \*.dat files contain the collision matrix that is passed to the game and through that the game generates collisions. To use the file editor, the build script compiles an execute into the asset folder, so you just need to run the program called “collisionMaker.exe” or equivalent on linux . This view should appear on your screen:

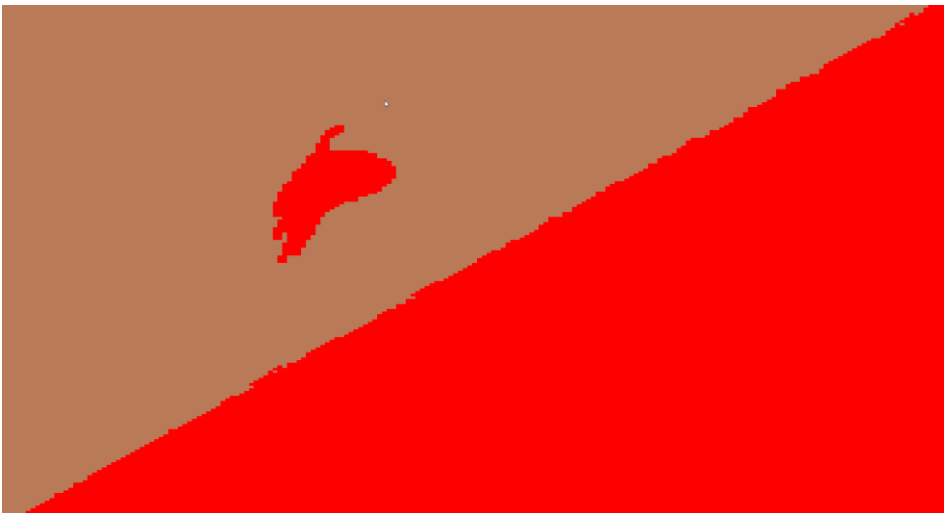


```
Tile collision editor, mouse = set area, s = save to disk, d= display to console, l = load from disk
Tile number (number NOT FILENAME):
```

To create a new collision for an tile, you have to have an sprite for it. To make the editor work properly, you should provide a file named accordingly: tile[user number].png. This way the editor can load the sprite and you are able to craft collisions to it. To retrieve a file for this, pass only the number of the file, nothing else. As an example, if we have an sprite named tile100.png, you pass the number 100 to the editor to open the file. The collision editor looks something like this:



Basically the editor loads up the png file and provides a way to draw collisions to the file. Drawing a collision works by pressing left mouse button and drawing. It produces a red trail which means a collision is on that pixel. To remove a collision just use right mouse button on a collision to remove it.

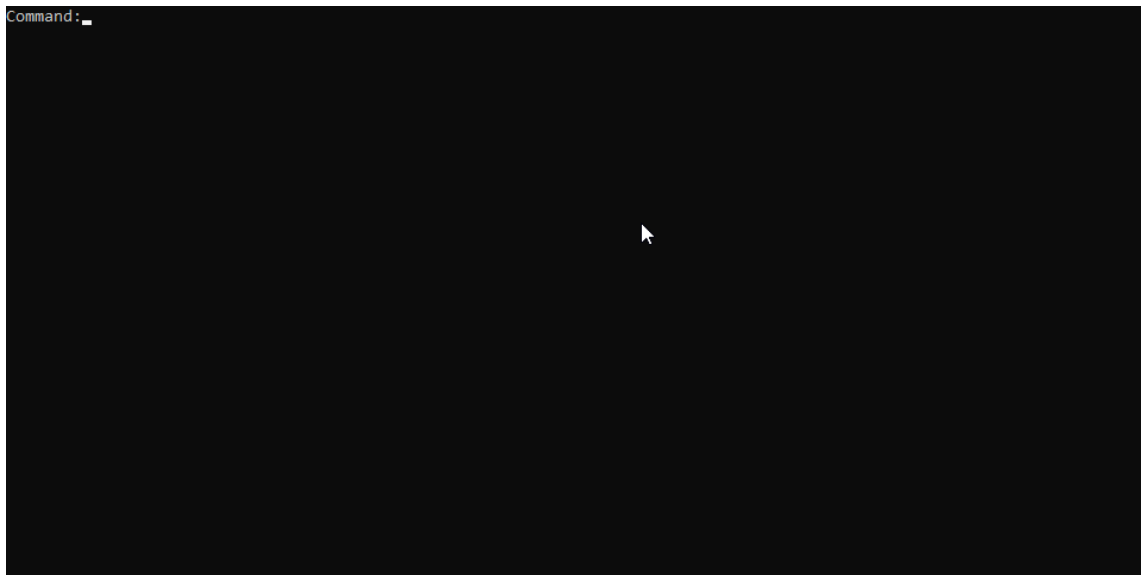


(red area is full of collisions)

To increase or decrease brush size use the arrow keys. Additionally, you can load an old collision file by typing the letter "l" and providing the proper number. In addition saving works by typing "s". This creates a temp file which is converted to a proper file after closing the program.

## The map editor

The map editor is a text based tool to ease the creation of map files. Running the program is similar to the tile editor. Just compile and run the editor called “mapEditor.exe” or equivalent on linux. This view should pop-up when run the program:



The program takes on commands which are provided below:

command reference:

list- show current data

music <8 character filename> - set music file for map

setw <width> - set map width

seth <height> - set map height

setlinks <amount>- set amount of different tiletypes used in map

link <tile\_index> <tilenumber>- link tile type to tile array index

tile <x> <y> <tile\_index>- set tile in map to match the tile at given tile array index

save <filename>- save map file

load <filename>- load map file

new- clear all program map variables

clearmap- clear map tile contents

A picture of an map working map:

```
Command:load map4.map
Command:list
MAPWIDTH:16
MAPHEIGHT:16
STARTX:1620
STARTY:340
MUSICFILE:music002
LINKDATA:
02 03 04 05 06 07 08
MAPDATA:
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 00 00 00 00 00 00 46 00 00 00 00 05 01 01
01 02 00 00 00 00 00 00 46 00 00 00 00 00 05 01
01 00 00 00 00 03 01 01 01 01 04 00 00 00 00 01
01 00 00 00 03 01 01 01 01 01 01 04 00 00 00 01
01 00 00 03 01 01 01 01 01 01 01 04 00 00 00 01
01 00 00 01 01 01 01 01 01 01 01 01 00 00 01
01 FFFFFFFC6 FFFFFFFC6 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 00 00 01 01 01 01 01 01 01 01 01 00 00 01
01 00 00 01 01 01 01 01 01 01 01 01 01 00 01
01 00 00 05 01 01 01 01 01 01 01 01 02 00 00 01
01 00 00 00 05 01 01 01 01 01 01 02 00 00 00 01
01 00 00 00 05 01 01 01 01 01 01 02 00 00 00 01
01 04 00 00 00 00 00 00 FFFFFFF86 00 00 00 00 03 01
01 01 04 00 00 00 00 00 FFFFFFF86 00 00 00 00 03 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
Command:
```

While passing a music file, do not add a file extension behind the file name. Also the file name should be exactly 8 characters long. Currently only supported file type is a \*.wav file.

## The Server Map –editor

The server map editor is used to edit what file the server should tell the client to load on their machine. Additionally, this tool is used to create, edit and delete AIs from a game session. The tool generates files that are named according to a map with a file name of \*.srv. Commands for editing and creating files provided below:

command reference:

list- show current data

addai- add new ai player to mapmap <map\_filename>- define mapfile to be used by client

useai <ai\_number>- select ai player for editing

removeai<ai\_number>- remove wanted ai player

addc <x> <y>- add new ai coordinate at end of list



insertc<coord\_id> <x> <y>- insert new ai coordinate at index

removec<coord\_id>- remove selected coordinates from list

removecl- remove last item in ai coordinate list

save <filename>- save server map data to file

load <filename>- load server map data from file

new- clear all program map variables

A complete map server file

```
Command:load map4.srv
Command:list
ClientMap:map4.map
AI's:1
Selected AI:0
Current path:
ID:0 X:1620 Y:340
ID:1 X:550 Y:310
ID:2 X:2250 Y:440
ID:3 X:2820 Y:910
ID:4 X:2830 Y:2180
ID:5 X:2300 Y:3499
ID:6 X:1010 Y:3549
ID:7 X:480 Y:2350
ID:8 X:330 Y:840
Command:
```

## Building and how to use the program

Building the program is pretty straight forward, the main difference is, as always, between operating systems. Thus we will handle building the software for windows and unix separately.

### Windows:

- A **mingw64 7.3.0** or higher version is required to be able to compile the program.
- A **SFML source folder** should be located under the source folder. **Version 2.5** or higher is required.
- Build the client game code by running **buildClientWin.bat** and to build the server run **buildServerWin.bat**
- To build tools, run **buildToolsWin.bat** under the assets folder. If you encounter errors while running the tools, copy the \*.dll files under the source folder to remove linking issues.

### Linux:

- Install SFML from a package manager.

- To build the client game code by running **buildClient.sh** and build the server by running **buildServer.sh**

Running the game doesn't differ from each other platforms that much, you mainly run different scripts to compile executables. However, the overall process is somewhat set in stone. It is important that before running the game, you **must** have a server setup, either a local one or a external one. Running the server can be run either through the executables or through the terminal/cmd. The server asks the user to provide how many players it expects to receive and which map it is going to run. Running the server from cmd happens like this:

Windows

```
serv.exe map4.srv 2
```

Linux

```
./serv map4.srv 2
```

While the server is waiting for other players to connect, it will halt the client program, thus causing not responding to apper. Do not close the window at that point.

To run the actual game, you can either run the game.exe or in unix by typing “./game” into the terminal.

A main menu appears:



From the menu you can set your player name, the address where to connect, default value is local host and connecting to a server that runs in the address provided in "Set address".

## Testing

For our testing purposes, we decided to use an external library Catch2, which is used in unit testing. Our testing was executed manually because automated testing wasn't really necessary in this project. These tests ensure us that our functions are actually working and they return the correct values. They are easy and fast to run, and the test file is very customizable and accessible.

We divided the test file into different test cases, that include testing the map and tile functions, vehicle and player functions, and some network tests. In the map test case we test our map functions, such as getting the tiles with different parameters, getting the size of the grid, setting the tiles and etc. The vehicle and player test case includes functions that return different variables of the vehicle and player variables, such as location, ammo, hp, speed, sprite and etc. The network test case is used to test different aspects of the network, which include connecting the player to the network, getting the servers map, getting connection status, playercount and checking the connection ID.

We found these tests to be useful for checking if any bugs appear after making changes or merging the code. However some of the testing was also just done by running the client and visually seeing

the game. The testing on our project was managed well and we were able to locate the bugs and various issues very fast. The outcome of our test file stated if there were any tests that were not passed, once the tests pass the following outcome was displayed to us.

To run test, just execute the bat file "runTests.bat" in windows or run the script "runTests.sh" in unix.

=====

All tests passed (35 assertions in 3 test cases)

As you can see, the test file is pretty clear with its outcomes. For an example, I disconnected from the server. This will give an different outcome where it shows which assertions didn't pass. I will only list a few here as it would be very lengthy to show them all.

Test if network works properly.

-----

main-test.cpp:65

.....

main-test.cpp:73: FAILED:

CHECK( aa.connect(pelaaja) == true )

with expansion:

false == true

(Here would be all of the other assertions that didn't pass.)

=====

test cases: 3 | 2 passed | 1 failed

assertions: 35 | 30 passed | 5 failed

From this outcome we would be able to spot which methods are not working and than we would try to fix them. We think that this was a pretty good system for checking the functionality of simple methods that were easy to check.

## Work log

Our work logs are included below, we divided the work by taking into account everyone's interests, strengths and other factors like time constraints. We had some main interests, which we used to roughly divide the work but we slightly deviated from them when necessary. Mostly we worked on aspects that were relevant at that point of time in the project.

### Antti Nousiainen

Week	Description of work	Time used (h)
29	Planning the project and researching about SFML and etc.	5
30	Initial game engine, initial network code, initial collisions, code merges and much more, added build scripts	30
31	Improved game engine, improved collisions, improved network code	15
32	Tile editor, other game engine tweaks	10
33	Busy at work	0
34	Redid all menus, looked at implementing collisions for projectiles on server, initial 2d collisions	10
35	Added 2 map editors, added projectile code to server, did changes to client and server map format, redid collisions to make them actually work, added runscripts, formatted and commented code, added ai players, added win condition and checkpoints to server and for ai, did lots of debugging, added ability to restart game without closing executable to client and server software, added command line parameters to server and client software... lots more too, check commits in git...	40

### Oskari Järvinen

Week	Description of work	Time used (h)
29	Planning and learning necessary technologies	5
30	Work on how to generate a map for the game. General game engine stuff.	15
31	Loading maps from a file	10
32	Real world work, mainly read some stuff about cmake	5
33	Did nothing since too busy week at work	0
34	Made myself familiar with cmake and catch2 testing. Additionally, brainstormed on how to further project with team members and how to implement certain features	10
35	Tried to implement cmake and implemented unit test. Also implemented music into the map files and how to read it to the game and play it. Wrote the documentation.	30

### Anton Sulkava

Week	Description of work	Time used (h)
29	Planning the project and researching about SFML and etc.	5
30	Working on the projectiles for the game and shooting,	10
31	Polishing up the shooting mechanism and adding different variables for the players. (HP, ammo, etc.)	5
32	Including some audio for shooting and testing music file.	5
33	Starting to work on different game objects (boosts, tar spills), adding visuals for them and randomizing their positions.	10
34	Polishing the game objects and getting them to work properly.	5
35	Adding a goal line system with checkpoints to the game and win condition with lap amount. Worked on the documentation as well.	10

Juho Aalto

Week	Description of work	Time used (h)
29		
30		
31		
32		
33		
34		
35		