

ELEC-A7150 C++ Programming

Autumn 2017

Project documentation

Micro-machines-1

Miika Karsimus 479437

Martin Vidjeskog 432539

Vili Karilas 552794

Hannes Pitkänen 475127

Table of contents

1. Overview	2
2. Software structure	3
Class descriptions	3
Class diagram	5
External libraries	5
3. Instructions	6
Building	6
Generating doxygen documentation	6
Using the software	7
Testing	11
Work log	12
Conclusion	14

1. Overview

Our task was to implement similar top-down racing game as legendary Micro Machines from the '90s. We set the following goals in the project plan

- Fun to play and has no major bugs
- Never crashes
- All the minimal requirements and some extra features, including weapons, sounds, different vehicles and possibly a simple artificial intelligence
- Random track generator
- Filehandler (loading/saving tracks)

All minimal requirements are implemented:

- Basic gameplay with simple driving physics (self-made) and multiple players (1-4)
- Multiple (2) tracks loaded from XML files
- Game objects which affect to the gameplay (Oil spills)
- Fun gameplay!

In addition, the game contains the following optional features

- Sound effects (weapons, collisions, explosions, engine, menu and pickup sounds)
- Split screen mode
 - The right view follows the first vehicle and the left view follows the second vehicle. Also AI can be used in split screen mode.
- Weapons (gun, heat-guided missile, turbo boost)
 - All weapons can be used by pressing the same key.
- Artificial intelligence
 - Drives around the track but does not shoot player. AI is very fast which is why it does not need to shoot .
- Four different Game modes
 - time trial
 - normal race: 1 human vs 1-2 AI
 - split screen: 2 humans vs 0-2 AI
 - practice

2. Software structure

Every class is stored in a separate file. Classes and functions are declared in header files (.hpp) and implemented in source files (.cpp). We have tried to create a natural class structure: Every single entity is represented in a single class. Classes, which share some common features, inherit the base class. For example Weapon is the base class for Gun, MissileLauncher and Turbo (speed boost) and Race is the base class for TimeTrial and SplitScreen. Another reason to use inheritance, is to utilize dynamic binding and runtime polymorphism: We can store pointers to Weapon-derived objects in the same container and the actual virtual function to be called, is determined runtime, depending on the actual type of container item.

Class descriptions

Vector2D: Utility class for representing a vector in XY-space (x and y components). Acceleration, velocity and direction of a vehicle has type Vector2D. Defines basic vector operation, such as calculating dot product or angle between two vectors or vector rotation.

VehiclePhysics: Defines physics engine for a vehicle. Has nothing to do with graphics. The basic principle is, that the position of the vehicle is calculated, based on its acceleration, velocity, angular velocity, initial position and of course elapsed time. Calculates the new velocity, when a collision occurs. Also missile and bullets utilizes this class to simulate moving.

Vehicle: General class representing vehicles. The most important members are shape and physics. Shape is a graphical SFML shape, which is drawn on the screen and physics is an instance of VehiclePhysics class. Vehicle class synchronizes shape with vehicle, so that shape has always a correct position, based on computations in VehiclePhysics. Vehicle also contains for example a list of player's weapons and many other features.

Car (inherits Vehicle): Car class shares features with Vehicle; nothing new is implemented here.

AIVehicle (inherits Vehicle): Defines AI algorithms which computes the proper control values for AI, such as acceleration and angular velocity in order to keep the vehicle on the track.

AICar (inherits AIVehicle): Shares the features with the base class. Nothing new is implemented here.

Track: Contains track items, such as walls, finish line, checkpoints and weapons on the track. All elements are read from an XML file during constructing Track.

Game: Includes the main loop for the game and changes states between different menus and actual race. SFML window is created here as well as Race instances.

Race: Contains instances of the following classes as attributes: Track, Camera, Vehicles (in container). The game logic is defined here. The most part of the graphical features and game logic are defined here, e.g. showing player status, lap times, elapsed time, remaining HP, places in the race.

TimeTrial (inherits Race): Class for representing a time trial race. Defines a couple of own features specific to time trial, in addition on inherited Race functions. Overrides a few virtual function from the Race.

SplitScreen (inherits Race): Class for representing a split race. Defines a couple of own features specific to split screen race, in addition on inherited Race functions. Overrides a few virtual function from the Race.

Weapon: Abstract (pure virtual) base class for weapons. Defines empty virtual functions, which are overridden in derived classes.

Gun (inherits Weapon): Class representing a normal laser gun. However, bullets are separated from this class.

Bullet: Class for a single bullet which uses VehiclePhysics to simulate moving. The most important attributes are shape (graphical bullet) and physics (instance of VehiclePhysics).

Turbo (inherits Weapon): Vehicles' speed boost. Overrides few virtual functions from Weapon to work with turbo.

MissileLauncher (inherits Weapon): Launches missile. Overrides virtual functions from the base class. Includes missile attribute (instance of Missile class).

Missile (inherits VehiclePhysics): Heat-guided missile, which utilizes vehicle physics for moving. Follows an opponent vehicle and tries to damage/destroy it.

Camera: Controls the movement of 2D camera during a race. Can change between different views (general, static, left side, right side). Allows user to zoom the camera in/out with keyboard. Always follows either a single vehicle or the first vehicle on the left side and the second vehicle on the right side. This class also makes it possible to draw static (not moving) texts on the screen, e.g. lap times, player status.

Line: Class representing a straight line which has a starting point and endpoint. Has a function which tells if two lines intersects with each other. Used by Polygon class.

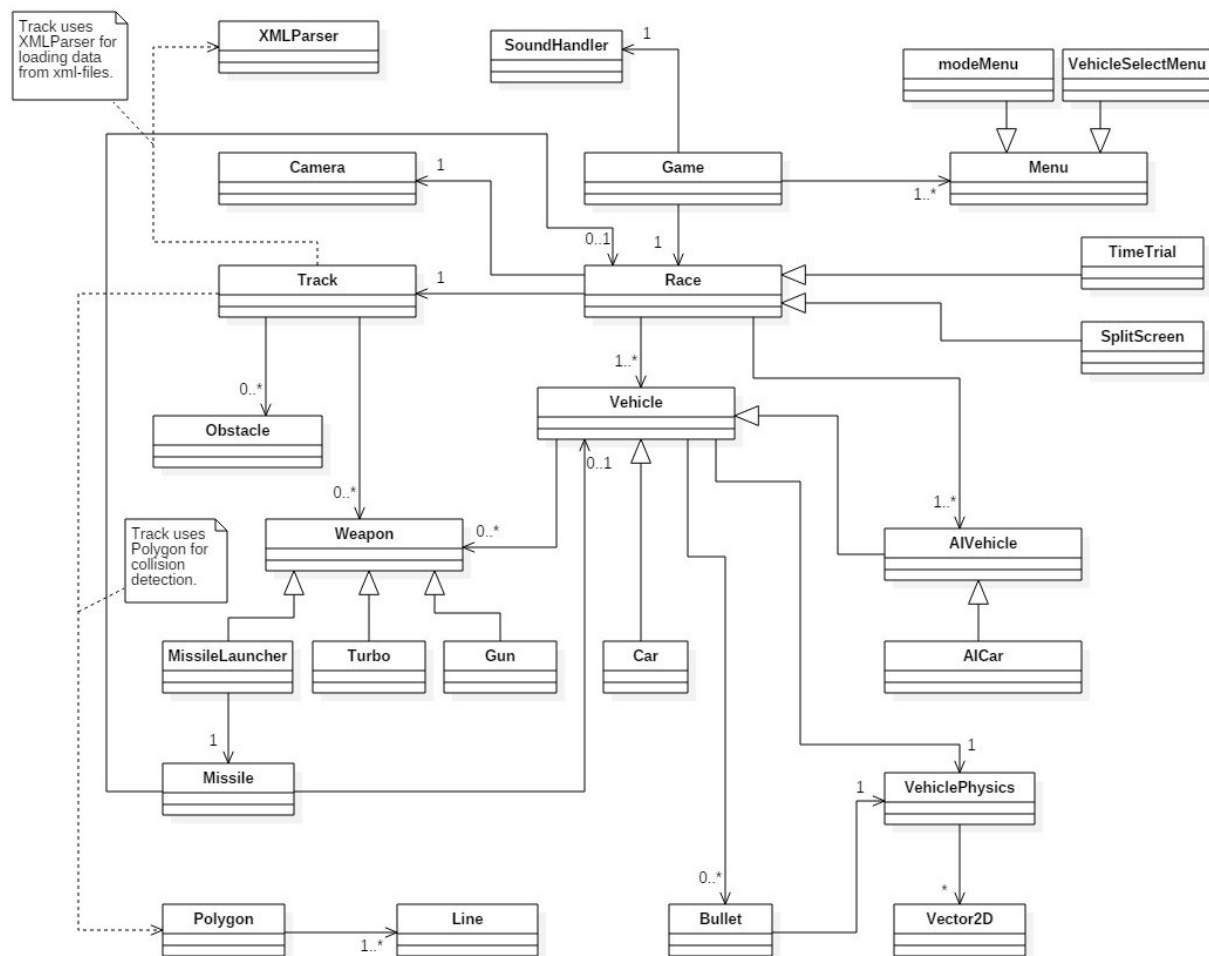
Polygon: Class representing a polygon which consists of lines (Line object). Has a function, which tells if two polygons intersect (collide) with each other. Used for collision detection between vehicles and track walls in cases where SFML's build-in collision detection is not enough.

MainMenu: Defines the main menu. Created in Game class.

SubMenu: Defines the submenus under the main menu. Created in Game class.

XMLParser: This class is used to read track elements from XML files. Called from the Track constructor. Utilizes TinyXML2 library, which is downloaded from GitHub.

Class diagram



Picture 1: Class diagram

External libraries

SFML is used for drawing graphics. The basic interface between game logic and graphics is the Race class: It calls update functions to update the position of vehicles and all other objects. Almost all graphics are drawn in the Race class.

TinyXML2 is used for parsing XML files. We have not pre-compiled TinyXML2 to static/dynamic library, because it is small and lightweight project and very fast to compile with the other files. TinyXML2 contains only one source file and one header file. Instances are created in the Track constructor.

We have not used Box2D for physics. All physics are implemented by ourselves, including collision detection and handling.

3. Instructions

Building

The project is build by using CMake and Make. Run the following commands (\$ refers to Linux terminal):

```
$ cd src
$ mkdir build
$ cd build
$ cmake ..
$ make
```

To run the app, type the following:

```
$ make run
or
$ ./app
```

CMake finds SFML and links it to the project. We have used FindSFML.Cmake script to locate SFML from a Linux computer. The script is downloaded from github and it is also available at Aalto Linux machines.

Generating doxygen documentation

We have used such coding and commenting style, that it is possible to generate documentation directly from the source code. To generate html document files, type

```
$ make docs
```

It stores the html files in doc/html directory (in project root). After that, navigate to doc/html and open index.html with you web browser to see the class documentation. Of course, Doxygen must be installed in order to be able to run the previous command.

The following commenting style has applied:

1. Write the general class description in header file above the class declaration.
2. Write a short description above each member function in header file: What the function does, what is the meaning of parameters and how to use the function. If the function is self-explanatory, comments can be omitted.
3. Write short inline comments to source files inside the function body, if there are lines that are difficult to understand without comments.

The idea is that the class can be used without knowing the exact implementation of member functions. It should be enough to see header file comments and function declaration. Consider the following function (member in Vector2D class)

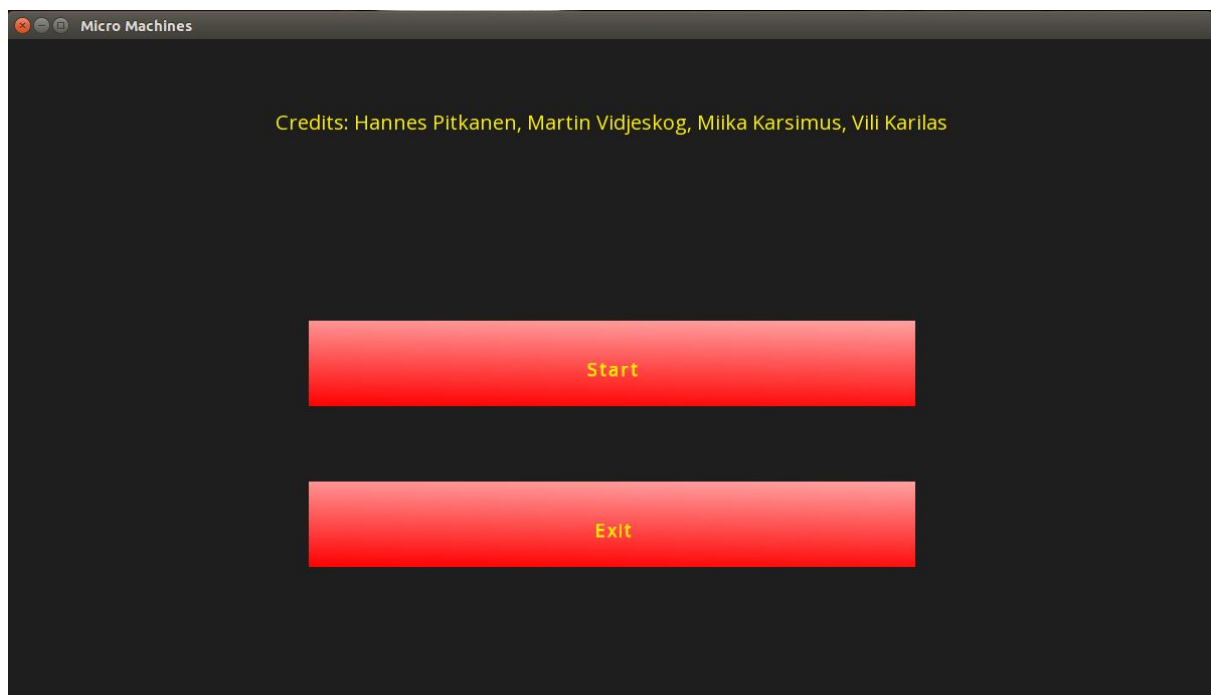
```
/// Return an unit vector parallel to this vector.  
Vector2D getUnitVector() const;
```

It is easy to see, what this function does and how to use it, without knowing the implementation in cpp file. Of course, in case of more complex functions, it is necessary to see also the implementation in cpp files.

Using the software

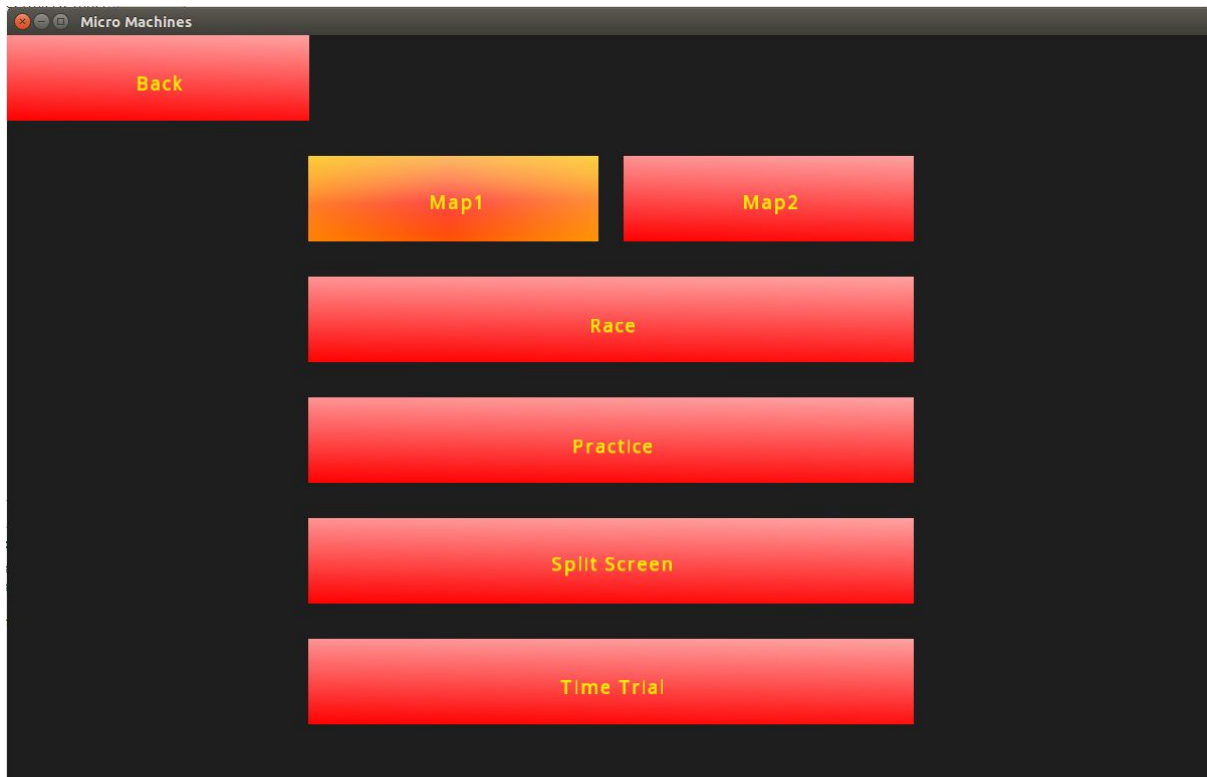
Menu:

When you start the program, main menu opens. There is two options, “Start” and “Exit”. The first option opens a menu where user can select map and game mode. The second option exits the game. The main menu is shown in picture 2.



Picture 2: Main menu

Let's assume that user selected “Start”-option. This opens the second menu (picture 3) where user can select map and game modes. Map1 is selected as a default. If user wants to use Map2 instead, click “Map2” button. After that, it's time to select the game mode.



Picture 3: Map and game mode selection menu.

Game modes:

Table 1: Game modes

Race	Race with opponent(s). Race will last 5 laps, and the car that finishes first will be a winner.
Practice	Practice on your own. This is a good option to learn, how to play this game!
Split Screen	Play with your friend using split screen mode. Race will last 5 laps, and the car that finishes first will be a winner.
Time Trial	Try to beat the target time. No opponents.

After choosing the game mode, a new menu will open (Picture 4). This will be the last one. In this menu, user should choose, how many AI drivers you want and how many human players. Some options that are available in this menu may be blocked because the game mode you selected does not support all of the options.

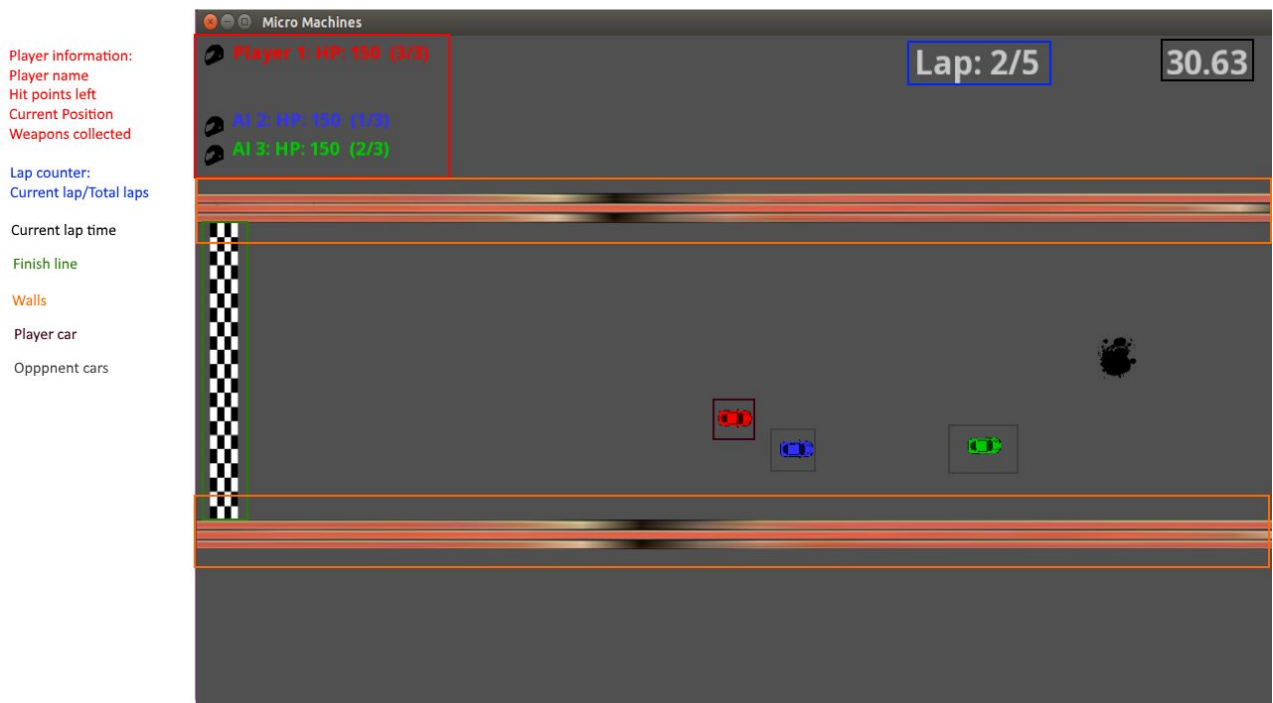


Picture 4: Final menu

After choosing opponents, user can hit the “Start!”-option and the race will start.

Basic gameplay:

Players should drive around the circuit until all laps are completed. The first player to do this is the winner. Each car has hit points. Player will lose hitpoints, when he hits the wall or someone is using some kind of weapon against the player. If a player’s hitpoints reach zero, player’s car will be destroyed. Hitting the wall will also result lost of control, and the car starts to slide. See “Weapons” chapter for more information about weapons, and “Obstacles” chapter to learn more about obstacles. Basic information during race is explained in picture 5.



Picture 5: Basic gameplay

Car Controls:

Player 1:

W = Throttle

A = Turn left

S = Brake

D = Turn Right

E = Use weapons

Player 2:

↑ = Throttle

← = Turn left

↓ = Brake

→ = Turn right




L = Use weapons

The camera view can be zoomed in and out by pressing N and M keys (both views in splitscreen).

Weapons:

Before you can use the weapons, you must collect one. Weapons can be found random locations on map. You will collect a weapon just driving through of it. When you have collected a weapon, you will see it on player information box. Keep in mind that weapons must be used in same order than they were collected! Also, missile is very powerful weapon and does not need much skill to destroy enemy. This is why number of missiles in race is limited for two. All the weapons are described in table 2.


Table 2: Weapons

	Turbo This will improve your car acceleration. Probably a little bit hard to control for inexperienced players but you will learn to like it.
	Missile Launch a homing missile to destroy your opponents. Flying time is 10 seconds. After that, the missile is destroyed.
	Laser Gun: Shoot a laser ray to damage/destroy your opponents. Goes through every wall.

Obstacles:

Obstacles should be avoided! They will spawn random locations on map. This happens when the race starts. So every time user starts the game again, locations of obstacles will differ more or less. Currently the game supports one obstacle. Check table 3 for more information.

Table 3: Options

	Oil splat You will lose a car control, if you hit with this obstacle.
---	---

Testing

The game was mostly tested with gameplay during the development. Many hours were spent by looking for errors in racing situations and even more time was spent wondering what causes them. There is no unit-tests but we used valgrind for testing memory leaks (Picture 6). Valgrind shows one error but this is most likely caused by some “feature” in SFML. This is because the game does not use any manual dynamic memory allocation (new keyword). Debugger tools were also great help in testing and error hunting.

```

==3684== by 0x40107CA: _dl_init (dl-init.c:120)
==3684== by 0x4000C69: ??? (in /lib/x86_64-linux-gnu/ld-2.23.so)
==3684== LEAK SUMMARY:
==3684==    definitely lost: 56 bytes in 1 blocks
==3684==    indirectly lost: 56 bytes in 1 blocks
==3684==    possibly lost: 0 bytes in 0 blocks
==3684==    still reachable: 354,030 bytes in 2,187 blocks
==3684==    suppressed: 0 bytes in 0 blocks
==3684== For counts of detected and suppressed errors, rerun with: -v
==3684== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
mevid@mevid-VirtualBox:~/Workspace/micro-machines-1/src/build$

```

Picture 6: Valgrind output

Work log

Vili Karilas: Responsible for all menus, sound handling/sounds and the weapon Turbo. Also parts of the Game class.

Week 1	6h	Mainly spent setting up visual studio with git & cmake.
Week 2	8h	Looking at what's been made and how it works, learning SFML. Made class SoundHandler. And making of the engine sound (didn't count this toward the hours here).
Week 3	20h	The majority of the time went into finding out about state machines and struggling to implement one. The single commit that was made was quite large to keep the program usable for others.
Week 4	20h	Tinkering around with the messed up state machine, adding support for multiple sounds.
Week 5	30h	Converted menus into a virtual class system, added and made all remaining sounds, including implementation to play all of them (except explosion). Fixed quite a few bugs with the event handling in the menus, and cleaned up messy code.

Hannes Pitkänen: Obstacles and xml parser and xml files.

Week 1	5h	Learning how to use git and making programming environment ready for the project.
Week 2	0h	Nothing. :(
Week 3	20h	Working with Xml parser. Created xml file for map 1. Created Example.xml.
Week 4	5h	Started working with obstacles.
Week 5	20h	Working with Oilsplat obstacle. Also created new xml file for map 2.

Martin Vidjeskog: collision detection, tracks, artificial intelligence, homing missile.

Week 1	5h	Setting up development environment and writing project plan.
Week 2	5h	Started developing Track-class and read SFML-documentation.
Week 3	10h	Continued developing Track-class and started implementing collision detection. Also some other little things like creating Point-structure. Lot of testing.
Week 4	15h	Creating first track, changed collision detection to use our own classes and functions (getGlobalBounds() was not accurate enough). Polygon- and Line-classes with collision.hpp and collision.cpp were created for more accurate collision detection.
Week 5	20h	Added spawn points for weapons, started AIVehicle- and AICar-classes, implemented algorithm for AI-movement and started to design second track.
Week 6	25h	Created MissileLauncher- and Missile-classes and algorithm for their control. Also bug-fixes and other fine-tuning. Writing Project documentation.

Miika Karsimus: Driving physics and controls, camera and view control, split screen, game logic (Race, TimeTrial), gun, parts of the Game class.

Week 1	5h	Created CMakeList.txt, setting up building environment and writing project plan.
Week 2	7h	Started to implement driving physics (Vector2D, VehiclePhysics and Vehicle classes).
Week 3	15h	VehiclePhysics was almost ready. It had still a few bugs (car gets inside a wall).
Week 4	15h	View control: control the view with keyboard. Camera follows the car. Completed VehiclePhysics, including collision handling. Started working on Race, TimeTrial and SplitScreen classes. Fixed collision bugs.
Week 5	15h	Game logic: View control, split screen. All texts on the screen (laps, player status, lap time etc.).
Week 6	30h	Gun, gun bullets, destroy other vehicles with gun. Reduce car HP if a collision occurs. New features to time trial: Save laps to file, load the best lap. End game when the lap count has reached. New themes for tracks.

Conclusion

The overall result is rather good. The game works well and has no major bugs. Of course, we could have implemented more features, such as collision between two vehicles or different kind of vehicles, but the time was very limited. Due to the limited time, we didn't use very much efforts for testing: no external testing software/libraries were used.

We tried to follow good programming style and use robust features of modern C++, including smart pointers, exception handling, lambdas and dynamic binding. There is room for improvements but all in all, we are quite satisfied with the result.

The **rule of three** is a rule of thumb in C++ that claims that if a class defines one (or more) of the following it should probably explicitly define all three:

- destructor
- copy constructor
- copy assignment operator

https://en.wikipedia.org/wiki/Rule_of_three

By looking the source code of the project, one might think that the rule of three was not used very efficiently because there are many places where only virtual destructors were defined but copy constructor and copy assignment functions were missing. We have explicitly defined virtual destructors, because it is a good practice in case of inheritance, and in some cases, the compiler needs virtual destructor to be defined. But looking closer to the destructors, one can see that there is not a single destructor having any code in them. They are empty destructors and that is the reason why neither copy constructor or copy assignment functions are needed. No resources are allocated, so there is no need for releasing, moving or copying resources.