Project documentation - Micro Machines 5

Juuso Korvuo
Lauri Kurkela
Lauri Nyman
Henry Räbinä

Overview

This software is a top-down racing game inspired by Micro Machines. The objective of the game is to finish three laps before opponent players. The levels consist of roads and other material types (e.g., ice) which the players can drive on, and checkpoints which have to be reached in a certain order. In addition, the levels include physical objects (e.g., rocks, railings) with which the players can collide. The players may also collide with each other.

In this game, cars are used as the vehicle type, and the levels were planned accordingly. The maximum amount of players is four, and the controls for each player can be set manually. There are four levels that can be selected. In addition, there is a level editor that can be used to modify level 4.

The game has a menu screen, consisting of a start game -button, level editor -button, an options-button and an exit-button. The control scheme of the players can be changed in the options. The actual gaming screen is implemented such that the whole racing track is visible at the same time. This means that the screen will not be "moving" with the player. We used this approach because all the players will need to see themselves on the track at the same time.

The driving physics of the vehicles and collision detection are handled by the Box2D physics engine. The user interface and graphics are drawn using the SFML modules System, Graphics and Window.

Software structure

The fundamental structure of our program is that it consists of several screens, for example the Main menu -screen, the options-screen and the actual gaming screen. Each of these screens is displayed in the same SFML-window that is created in the very beginning of the program. To control when to display which of these screens, we have created several functions, each of which is used to display one screen. Two of these functions are in the file Main.cpp, and four are in the file Functions.cpp. These functions are also used to handle the different events done in these screens.

When the program starts, it first calls the function that displays the Main menu -screen. In this screen we have four buttons, three of which can be used to call another function that displays another screen, and from these functions we can further move on to other screens and so on. We think that this is a simple method to control the game. In addition, this makes the code quite easy to read and possibly modify in the future.

Our program has a total of seven classes: Level, Tire, Car, PhysicsObject, Player, Settings and Tile. Level describes the actual game level that contains for example the cars and different obstacles. Tire describes the tires of a car, that handle the movement of a car. Car contains the description

of a car that is controlled by the player. PhysicsObject ties a physical Box2D-body to a shape rendered using SFML. Player contains the description of a player of the game. Settings contains information about the game screen. Tile contains definitions for different road tile types. The class relationships are illustrated in Figure 1. Classes only have private member variables, because this makes managing the complexity of the code easier. These member variables can be accessed through public member functions, if necessary. No virtual classes were implemented because none of the classes share a similar role. A more detailed description of the classes can be found below the figure.

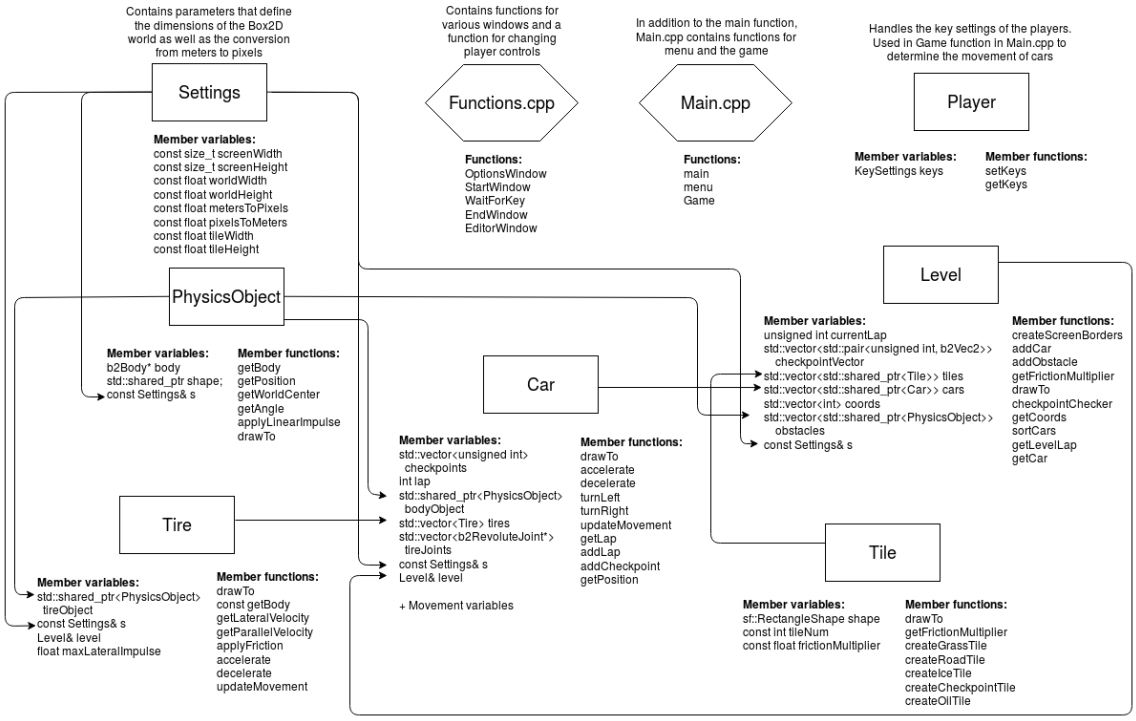

Figure 1: Class structure. Rectangles describe classes.

Level

The level class contains a matrix of tiles, where each tile element of the matrix defines the material of the road at some coordinates. 1800x1000 pixel level is defined by a 180x100 matrix, where each element corresponds to a tile that has a 10x10 pixel SFML shape associated with it. This allows easy queries to ask what the material of the ground is at some coordinates, and the vehicle physics are adjusted accordingly (decrease friction on ice, increase it on grass, etc.). The level also contains a list of all static obstacles (which use collision mechanics) in it. These physics objects are stored in a vector called "obstacles". The idea is that obstacles do not move under collisions, so the players have to dodge them. A pointer to each player's car is also stored in a "cars" vector. This is done to allow easy drawing of every drawable object: everything can be drawn to the screen by simply

calling level.drawTo(window), which calls the appropriate methods for each drawable object. A file from which the level is loaded contains values that represent different tiles and physics objects.

Tile

The tile objects are the building blocks of levels. The tile class is used to store information about a single block of the road, such as the friction multiplier and the color/texture. The class has static methods for creating different tile types (road, grass, ice etc.) and a drawTo method which is called when the level is drawn to a window.

PhysicsObject

The objects in the level which use features from the Box2D library will be physics objects. The purpose of this class is to tie the body of the object created using Box2D to the graphics created using SFML, to allow easy drawing of the objects. All obstacles in the level (rocks, railings) and parts of a car are PhysicsObjects or have PhysicsObjects associated with them.

Car

The Car class is the class used to create and control the drivable cars. The shape of the car's body is defined here, as well as multiple driving coefficients (max speed, acceleration, max turning speed and angle). The methods which handle the movement of the car are accelerate, decelerate, turnLeft and turnRight, which are called when the player presses the corresponding key, and the updateMovement method, which must be called for each iteration of the physics simulation (to handle friction and smoothly reset the tire positions when the player is no longer turning the car). These operations are performed by calling the corresponding methods for the tires of the car (see Tire class below). The car class also contains methods for handling the checkpoints in the level and asking the current lap number, as well as drawTo, which draws the car body and each tire to the window.

Tire

To enable more realistic driving physics, each tire of the car is modeled as its own object. The physical body of the tire is a rectangular PhysicsObject, which is created in the Tire constructor. The tires are connected to the car body using Box2D's b2RevoluteJoints. The tire class contains the same movement handling methods as the car, which are called from the car (none of the tire class methods need to be called directly). The tire class simulates friction by killing some of the lateral (with respect to the forward direction of the tire) velocity with each iteration of the physics simulation. With each iteration, a parallel friction force is also applied to the tire so that the car eventually stops moving if it is not accelerated or decelerated. The amount of friction applied per iteration is determined by the lateral and parallel velocities of the car, and the type of the tile the tire is on top of.

Player

The player class represents the player of the game. It contains information about the controls a the players. In this program the players do not have any other information, for example names or points, but this approach would allow them to be implemented easily. Different players are just objects of this class, named player 1, player 2, player 3 and player 4.

Settings

The class Settings contains information about the dimensions of the screen, world and tile. It also contains the coefficients for making transformations between pixels (used in SFML) and meters (used in Box2D). A single instance of the class is created when starting the program, and the values stored in it are not meant to be modified at any point. The Settings object is stored as a reference within many objects in the game, for example cars. It is used for drawing the objects to the screen correctly.

Constants (not a class)

The Constants.hpp file defines constant values used in several places. Currently it only contains multipliers for converting degrees to radians or vice versa. Some "magic numbers" in the code could be moved here in further development (e.g., the friction multipliers of the different tile types, the constants which define the driving properties of the cars...)

Instructions for building and using the software

Instructions for compiling the program

Cmake is used to compile the software. In the src folder, type the following command in the terminal:

```
$ cmake .
```

To finish the compilation process, give the command:

```
$ make
```

You are now able to run the game by typing the following command:

```
$ ./main
```

Instructions for using the program

When the program is run, the main menu -window will first come up. In the menu window, the user can choose to start a new game, open the options menu, open the level editor or exit the application.

In the options menu, the user can modify the controls of the players. When the user clicks on the Options-button a window will open in which are seen the default keys for each player. To change a certain key of a certain player, the user has to click on the correct key under the correct player. When the user clicks on some key, the color of that key will become blue. When the user now presses some button on the keyboard, the selected key will change to the key that was pressed (assuming that the user clicked on some key that is allowed to be used as a control for the players).

In the level editor, the user can make changes to map 4. When the user clicks on the Level editor -button, a window with green background will appear. If the user wants to return to the main menu or exit the game, the user must first press ESC on keyboard. In this case Main menu- and Exit-buttons will appear. To make these buttons disappear, the user must click on the ESC again.

To load the current version on map 4 on the screen, the user must press key L on on the keyboard. This will make map 4 appear on the screen. To save the map to map4.txt, the user must press S. It is not necessary to load the current version of map 4 on the screen, because the user can also just start to create the map 4 from scratch on the empty green screen.

New road materials and different obstacles can be created in the level editor by pressing certain keys on the keyboard. When these keys are pressed, a corresponding tile or obstacle will be created on top of the mouse cursor. The keys and their corresponding actions are as follows:

- Key 1: Grass
- Key 2: Road
- Key 3: Ice
- Key 4: Small rock
- Key 5: Another type of small rock
- Key 6: Large rock
- Key 7: Another type of large rock
- Key 8: Horizontal railing
- Key 9: Vertical railing
- Key 0: Diagonal railing
- Key +: Diagonal railing
- Key o: Oil
- Key c: Checkpoint
- Key space: Starting point for a car
- Key R: Remove a starting point

It is important to note that the user must make exactly four starting points in the game field. If there are too few starting points, that map cannot be played. Pressing the key S will save the changes. Furthermore, checkpoints must be inserted in the order that the cars are expected to drive on the level. The last checkpoint dictates at which point the lap changes; therefore, the last checkpoint should be approximately 10 tiles after start.

When the user clicks on the Start Game -button in the main menu, a new window will be opened in which the user can choose the correct amount of players and the map to be played. The default amount of players is one and the default map is map 1. The number of players can be chosen by clicking on the corrent number under the heading "Number of players" and the map can be chosen by pressing the correct map. The game can be started by clicking on the start-button on the screen. It is important to note that the game will not start if the keys of all players are not unique.

Clicking on the start-button will start the actual game. The players can control the cars by using the keyboard keys that were selected for each player. When some player has completed three laps in the game, measured from the first checkpoint, the game will end and a new window will appear telling which player has won. There are also Main Menu- and Exit-buttons on that window. The game can be paused by pressing ESC.

Testing

The programming was done such that every member of our group tested the piece of code that he made, before that code was added into the final program. The testing was mainly done by making some very simple test programs that just run some specific code. For example to test the physics engine, we made simple program that just opens a window and it creates some simple shapes that use the engine and that can be controlled with the keyboard. In this way the physics enigine could be tested before it was added to the final code.

To test the program for memory leaks, we used valgrind. Valgrind-tests were made about once every week so that possible memory errors would be relatively easy to fix. To test whether the conditional structures (for example if-statements) worked as expected, we made some simple print-commands inside some specific conditional statements, and then we run the program such that it should go inside those conditional statement. Then we just checked whether it printed to the console what it was supposed to print. We think that this is very simple but still quite effective method for testing conditional statements. With this method we can also quite easily trace down where a possible mistake has been made.

Worklog

Division of work

In the division of work we took into account the different interests of each group member and the different programming backgrounds and skills of group members. Our goal was to divide the work such that the programming would both effective and meaningul for every team member. Specific division of work can be found in the table below.

Table 1: Division of work and responsibilities.

Group member	Responsibilities
Juuso	Getting the graphics drawn using SFML to sync with the Box2D bodies (implementing the PhysicsObject and Settings classes, and the drawTo method hierarchy). Implementing the main gameplay functionality: driving physics (Car and Tire classes), and the physical obstacles and some of the tile types (in the Level and Tile classes). Wrote the test programs for PhysicsObject, Car and Level classes. Created map 1. Minor contributions to documentation.
Lauri K.	Wrote a Make file at first and then implemented the CMake files: Automatized building the Box2D library and linking SFML and everything to the project. Worked on the level editor (mainly loading a map and a few different tile types) and wrote methods related to checkpoints, counting laps and sorting players in classes Car, Tile and Level.
Lauri N.	Implementation of a rudimentary level editor, and making modifications to it. Responsible for error handling. Partly responsible for the documentation.
Henry	Was responsible for creating the menu-screens using SFML. Was also mainly responsible for combining different parts of the project (made by other group members) into the final program and making everything work together. Also drew the very nice maps 2, 3 and 4 and was partly responsible for the documentation.

Outline of the project

The group met at least once every week to discuss progress and plan ahead. We communicated with each other using Telegram to discuss urgent matters.

Table 2: Outline of the project.

Week	Tasks	Time used
Week 45	We received our topic, and started to look into SFML and Box2D. Also Make and CMake was studied and the first Make files were written.	Juuso: 10 h Lauri K.: 12 h Lauri N.: 0 h Henry: 7 h
Week 46	We wrote the project plan, and got started with the implementation of the menu window, the CMake, level editor and driving physics. We implemented the first design of the PhysicsObject class. CMake files were implemented: Building the Box2D library was automatized and SFML was linked using find.	Juuso: 15 h Lauri K.: 12 h Lauri N.: 9 h Henry: 9 h
Week 47	We made modifications to PhysicsObject class. We created Player class, Car class, Tire class, Tile class, and Settings class. We implemented a rudimentary level editor.	Juuso: 15 h Lauri K.: 8 h Lauri N.: 3 h Henry: 8 h
Week 48	We modified the driving physics and re-strucured the code. For the mid-term meeting, we had tentative implementations for all of the seven classes: • Player class • Car class • Tire class • Level class implemented with one level • Tile class • Settings class • PhysicsObject class	Juuso: 15 h Lauri K.: 6 h Lauri N.: 6 h Henry: 8 h
Week 49	We started to work on the project documentation and finished the code such that everything was working correctly.	Juuso: 1 h Lauri K.: 8 h Lauri N.: 8 h Henry: 12 h
Week 50	We finished the documentation and made some final modifications to the code, for example added comments and removed unnecessary code.	Juuso: 3 h Lauri K.: 2 h Lauri N.: 5 h Henry: 6 h