Hill Climb Racing 2

Sauli Sjögren 427450, Ilkka Saarnilehto 351791,
Miika Aspiala 424738, Johan Pelkonen 356482

Overview

Our game starts from the main menu section. There are three ways to go: Levels, highscores and instructions. “Level” opens a sub menu to choose a playable level. “High scores” opens a new window where the high scores are written. “Instructions” opens a new windows where the instructions are written. There are three levels in three different difficulties. The level names are “Easy Beasy”, “Vanilla” and “Hard as Rock”. When a level is selected, the game starts immediately. While in game, you can control the vehicle by using the arrow keys and the spacebar. Up and Down accelerates the vehicle, and Left and Right steers the vehicle to keep it in balance. The spacebar brakes the vehicle. The player can return to the level selection by pressing the escape key. The purpose of the game is to collect coins to gather points (2 points per coin) and to drive to the goal point. If the player reaches the goal point, points will be increased with 50, 100 or 150 points depending on the game level. Each level has also a different length. The vehicle consists of two wheels, the chassis and the head parts. The levels has a ground, where the vehicle drives. The ground is randomly generated line that has gentle shapes on the first level and more steep shapes on the difficult levels. We didn’t add the head collision part to the vehicle, because we run out of time. We also left the unlocking levels out of this project so all of the three levels are playable all the time.

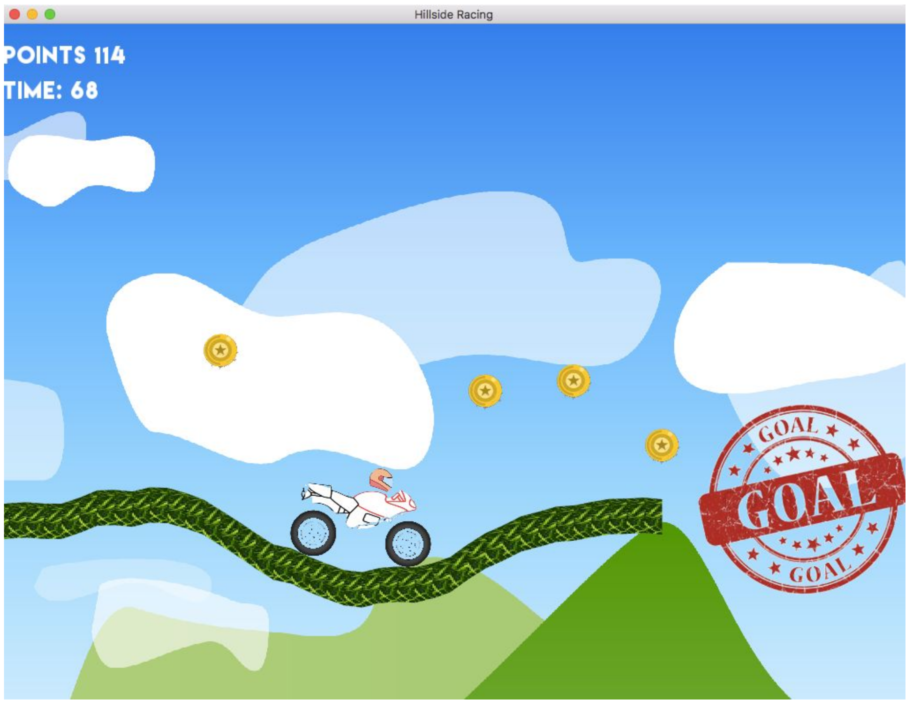


Figure 1. The game view with the goal point.

Software Structure

Basic Structure

The main loop that runs the program is built around the different states the program can be. These states are initialized as the program is started, and pushed into a vector “screens”. These states include e.g. the main menu, the highscore view and the game itself. While a recognized state is active, the program keeps running. When the actual game is started, Game class “open” function handles the actual game loop, drawing, updating and checking.

Each of the different states of the program inherit base class Screen. Screen has only one pure virtual function, open, and its purpose is to pass the window to the class that corresponds to the currently active state. When the open function in the correct class has been opened, the functionality alters vastly. In the case of the Game, it e.g. draws the entire gameworld and defines the physics, with the help of large set of other classes. In the case of MainMenu, it simply draws the menu screen and defines the controls.

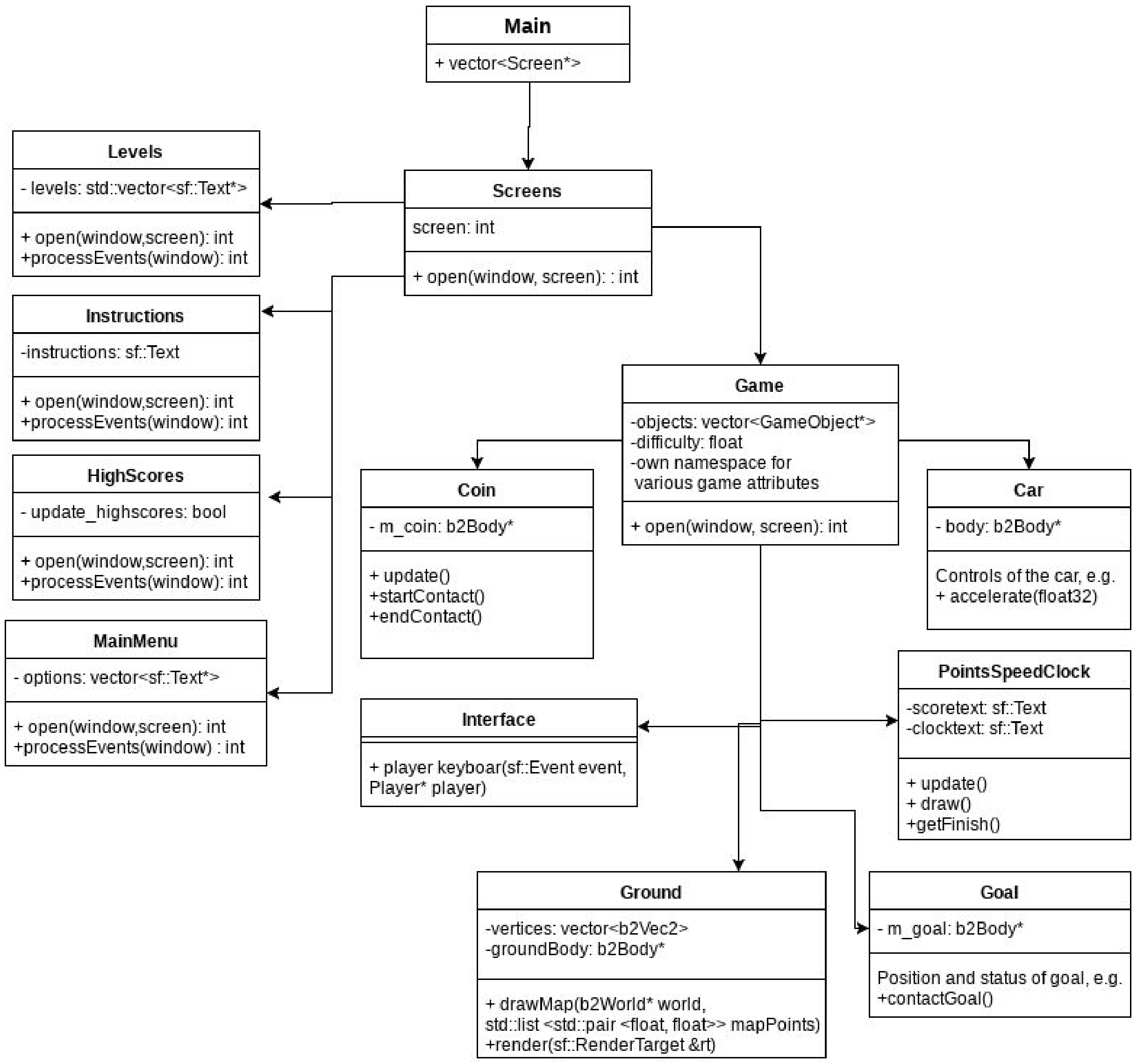
Game.hpp and Game.cpp handles the actual game, as said earlier. Game class function “open” handles everything around the game. First, the class will handle all the parameter initialization (map parameters, framerate of the game, background load and parameters, map physics creation, coins and goal initialization to the map, ground/terrain implementation). After this, all these are pushed to the objects that are drawn to the game. In addition, the user interface for the game(points, clock and finish line text) are initialized for the map. After this, the game loop starts. This loop monitors keyboard touches, background position, physics, graphics, screen texts and highscore saving points. In addition, it monitors coin collection and goal reaching. This loop also updates all the things I mentioned earlier. With the last command “window.display” this all is shown in the screen.

The project is reasonably modular. All the files with their function calls and definitions have been divided to .cpp and .hpp files (except Interface.hpp, GameObject.hpp, DEFINITIONS.hpp and Screens.hpp). The class structure is reasonably clear, and all independent and distinctive functionality has been divided into separate classes. The class structure is further illustrated in UML diagram next page.

As external libraries we adopted box2D to provide 2D physics and SFML to provide graphics.

Algorithms and Data Structures

We have used a random map generator of our own to achieve different map setups in each play time. It uses C++ pseudo random number generator seeded with computer time to produce randomness into the maps. The random numbers are added into a acceleration vector, which decides the direction of the upcoming track. The acceleration vector is manipulated by spring force -kx, which pulls the vector towards the centric line. In addition there are a friction force to damp the movement of the acceleration vector to prevent too large oscillation. The random numbers are generated through normal distribution, which mean is accorded to the given difficulty. In this way, we can control the random generated maps difficulty by 1) Increasing the normal distribution mean to make the track steep. 2) Increasing the magnitude of given random numbers to make sudden movements in the track. 3) Lowering the spring force and friction force to make great up and downs.


Main
+ vector<Screen*>
Screens
screen: int
+ open(window, screen): : int
Levels
- levels: std::vector<sf::Text*>
+ open(window,screen): int
+processEvents(window): int
Instructions
-instructions: sf::Text
+ open(window,screen): int
+processEvents(window): int
HighScores
- update_highscores: bool
+ open(window,screen): int
+processEvents(window): int
MainMenu
- options: vector<sf::Text*>
+ open(window,screen): int
+processEvents(window) : int
Game
-objects: vector<GameObject*>
-difficulty: float
-own namespace for various game attributes
+ open(window, screen): int
Coin
- m_coin: b2Body*
+ update()
+startContact()
+endContact()
Car
- body: b2Body*
Controls of the car, e.g.
+ accelerate(float32)
PointsSpeedClock
-scoretext: sf::Text
-clocktext: sf::Text
+ update()
+ draw()
+getFinish()
Interface
+ player keyboar(sf::Event event, Player* player)
Ground
-vertices: vector<b2Vec2>
-groundBody: b2Body*
+ drawMap(b2World* world, std::list <std::pair <float, float>> mapPoints)
+render(sf::RenderTarget &rt)
Goal
- m_goal: b2Body*
Position and status of goal, e.g.
+contactGoal()

UML class diagram of our project. The diagram is simplified: some of the less important utility classes have been left out, and each presented class only contains the most important methods and variables.

Instructions for compiling and use

The Hillside Racing program supports Linux and it can be build and run with the next commands. These commands builds the Box2D library and the game code, and after that opens the main menu screen.

How to build the game first time you download the repository

Download the git repository and navigate inside.
Navigate to the src folder.
Write the next commands (for building and running the game for the first time) to the terminal when in src folder:
1. chmod +x hillclimb.sh
2. ./hillclimb.sh
Wait for a while(about 30-60 seconds) when the game builds. When the build is ready the main menu screen will open automatically.
3. Enjoy the game!

How to run the game after build is done
After that you need to run the command: ./main if needed to run the game again. If this does not work do this in the src folder.

1. make clean main
2. wait for a while (about 30-60 seconds)
3. make run
4. enjoy the game

The instructions for playing the game
“Level” opens a sub menu to choose a playable level.
The levels are: “Easy Beasy”, “Vanilla” and “Hard as Rock” and they have an increasing difficulty
“High scores” opens a new window where the high scores are written.
“Instructions” opens a new windows where the instructions are written.

The keys
Up - accelerate
Down - reverse acceleration
Left - tilt backwards
Right - tilt forward
Space - brake
Escape - returns to the last menu screen

Testing

Most of our testing was handled via user testing and with function that gives bike parameters. Large amount of different user scenarios were run through with both our team member and outsiders. Additionally all Box2D related content was preemptively tested in the testbed provided by Box2D. For testing the independent modules different kinds of outputs and logs were also frequently used, e.g. the position parameters of the different game objects were captured and observed.
For memory management and handling leaks Valgrind was used in several stages of the project.

Worklog

The roles of the project was roughly the same that was stated on the project plan. This division was made roughly by class structure.

The roles

Miika
- Ground class. Generates the map with random generator (the level is different on each time) and increases the difficulty on harder levels.

Johan
- Car class. Generates the vehicle and implements the movement. The physical modelling of the vehicle.

Ilkka
- Main, main menu, levels menu and screens classes. Implements the menu screens and the changes between different events of the game. User interface.

Sauli
- Coin, goal and game classes. Creates the coins and the goal, and added the point counter. The contact between vehicle, coins and goal. In addition, makefile implementation, library handling and first stage testing.

The timetable

Week	Event
46	Planning phase, starting to study the Box2D and the SFML library.
47	Trying to link the Box2D and the SFML libraries and to make a simple main file that can use both. A lot of hours spent to the linking problem
48	Halfway meeting with the assistent. We showed a Box2D Testbed model that had a vehicle-like structure and it was controllable.
49	Starting to get something to the screen. Starting to implement the base classes such as car, ground and game classes. At the end of the week and hours and hours, we started to get a game-like structure.
50	Testing and modifying the playable game. Adding the highscores, modifying the vehicle parameters and splitting the code to header and source files. Finally making the game playable for Maari-house linux computers.

Some figures

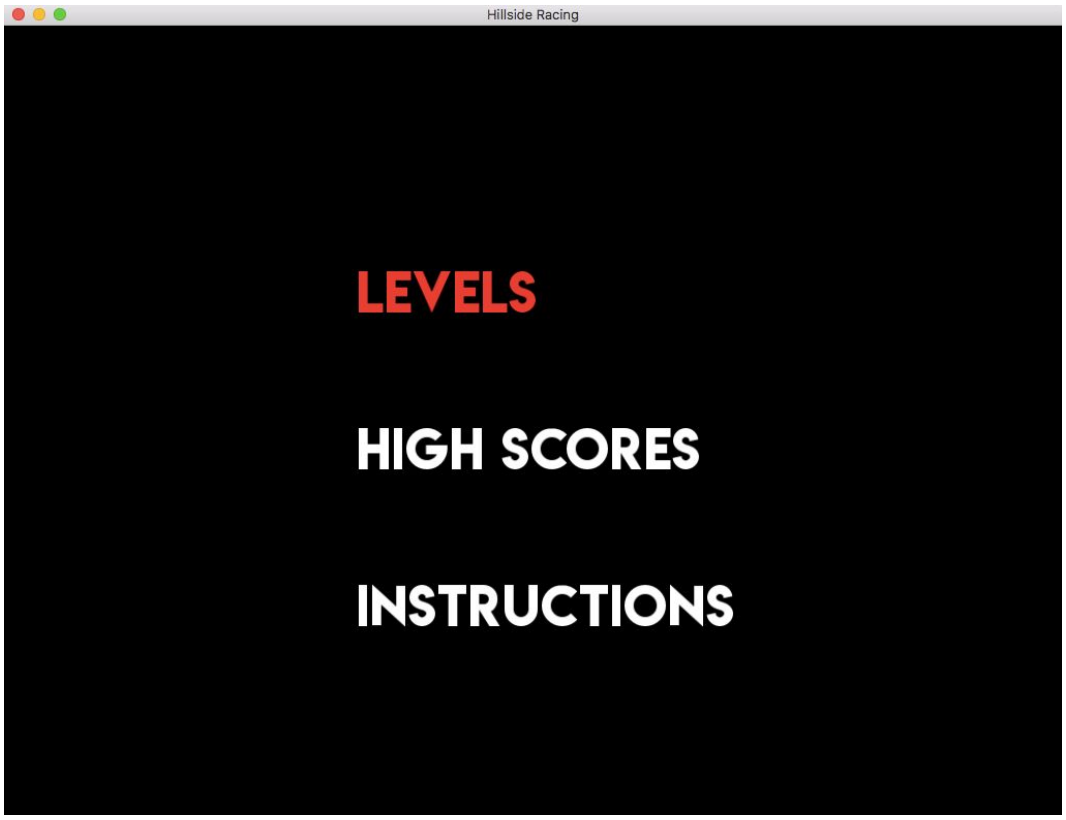


Figure 2. The main menu screen.

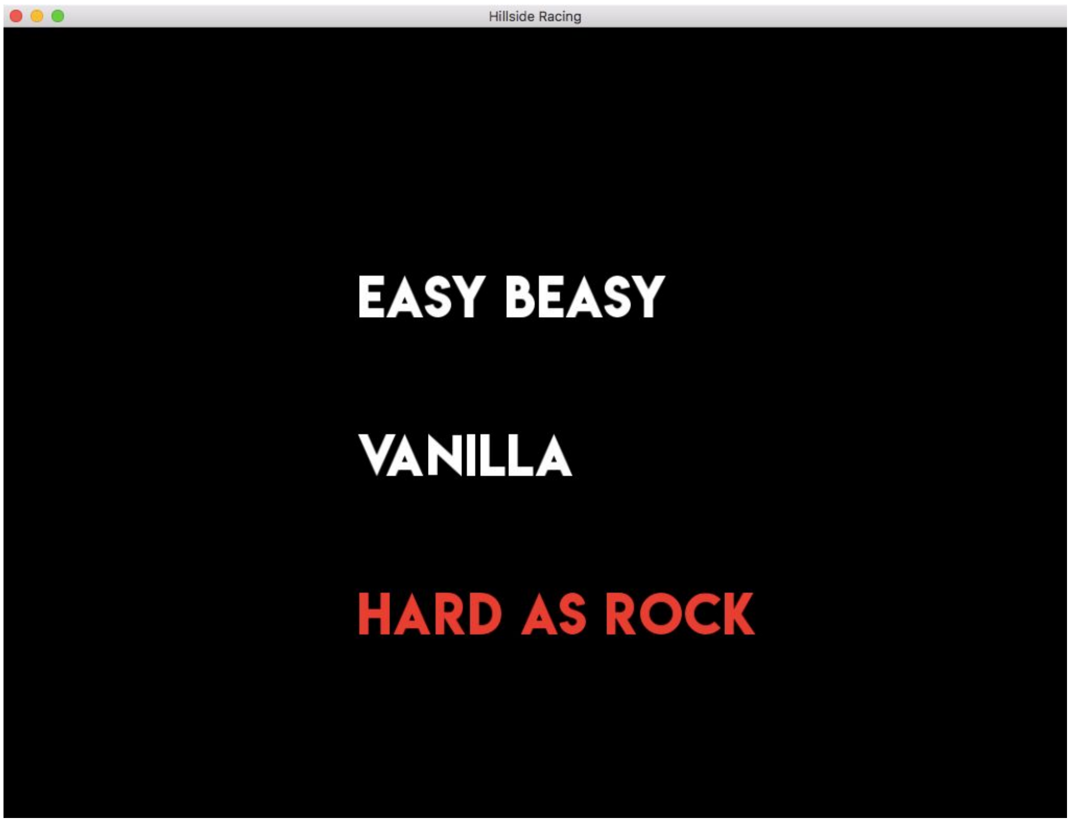


Figure 3. The levels screen.

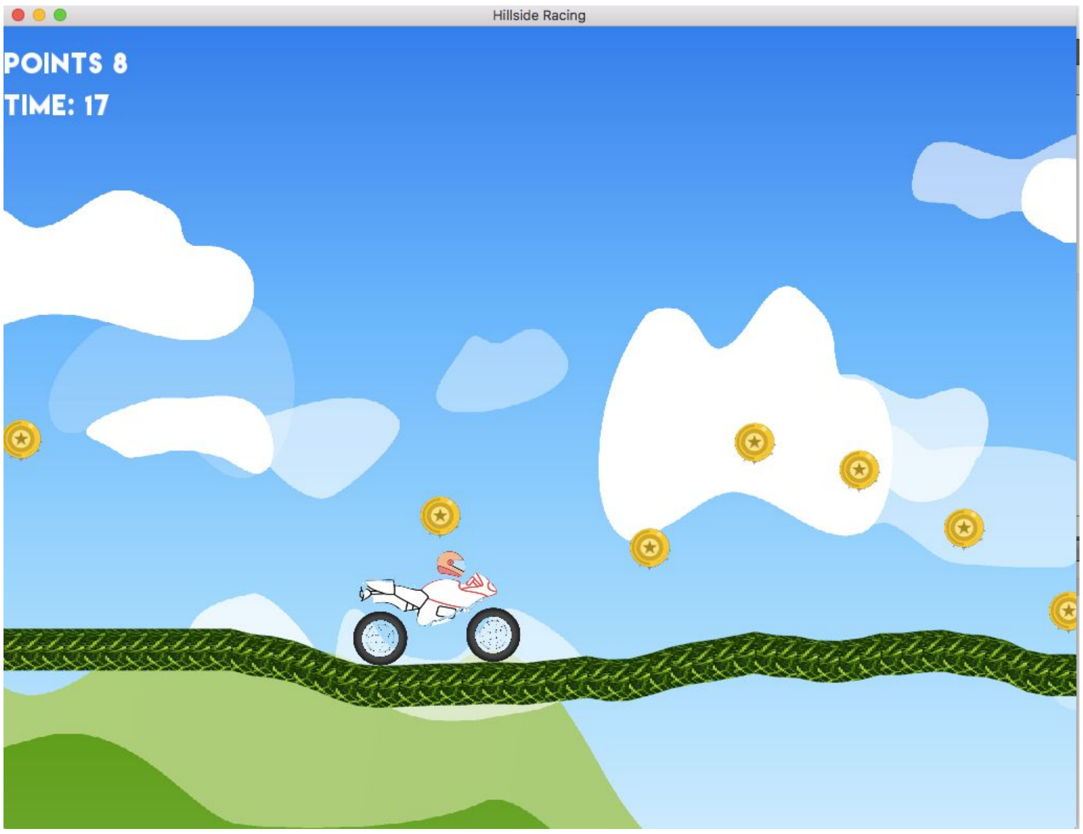


Figure 4. The game view.