11.12.2020

Micro Rally
Project documentation

ELEC-A7151

Elias Nurmi
Hilda Palva
Markus Tuominen
Ville Pihlava
Veeti Kahilainen

Overview

Micro Rally is a driving game in which the players can race with different cars in easily modifiable environments. Up to six players can join a game over the internet. The view of the game is top-down: the camera follows the player's car around the track. A minimap which shows the whole track is located on top-right of the screen. In addition there are sound effects in the game which occur for example when a car collides with an object.

The most important functionalities are driving physics, obstacles and collisions. The driving physics are implemented with the Box2D library. Any number of different tracks can be loaded from /res/maps.json. The tracks are grid-based, and every tile has set driving characteristics. Other objects, such as static tire stacks, movable boxes, boosts and oil spills are also loaded from the map file.

When the user starts the program, the main menu asks if the user wants to host the game, join a game or change settings. If the user chooses the join-option, they must enter the IP address of the host after which they are connected to the lobby and can choose a car and chat with other players. The host player chooses the track to play, amount of laps and their own car. When all players have chosen their cars, the host player starts the game.

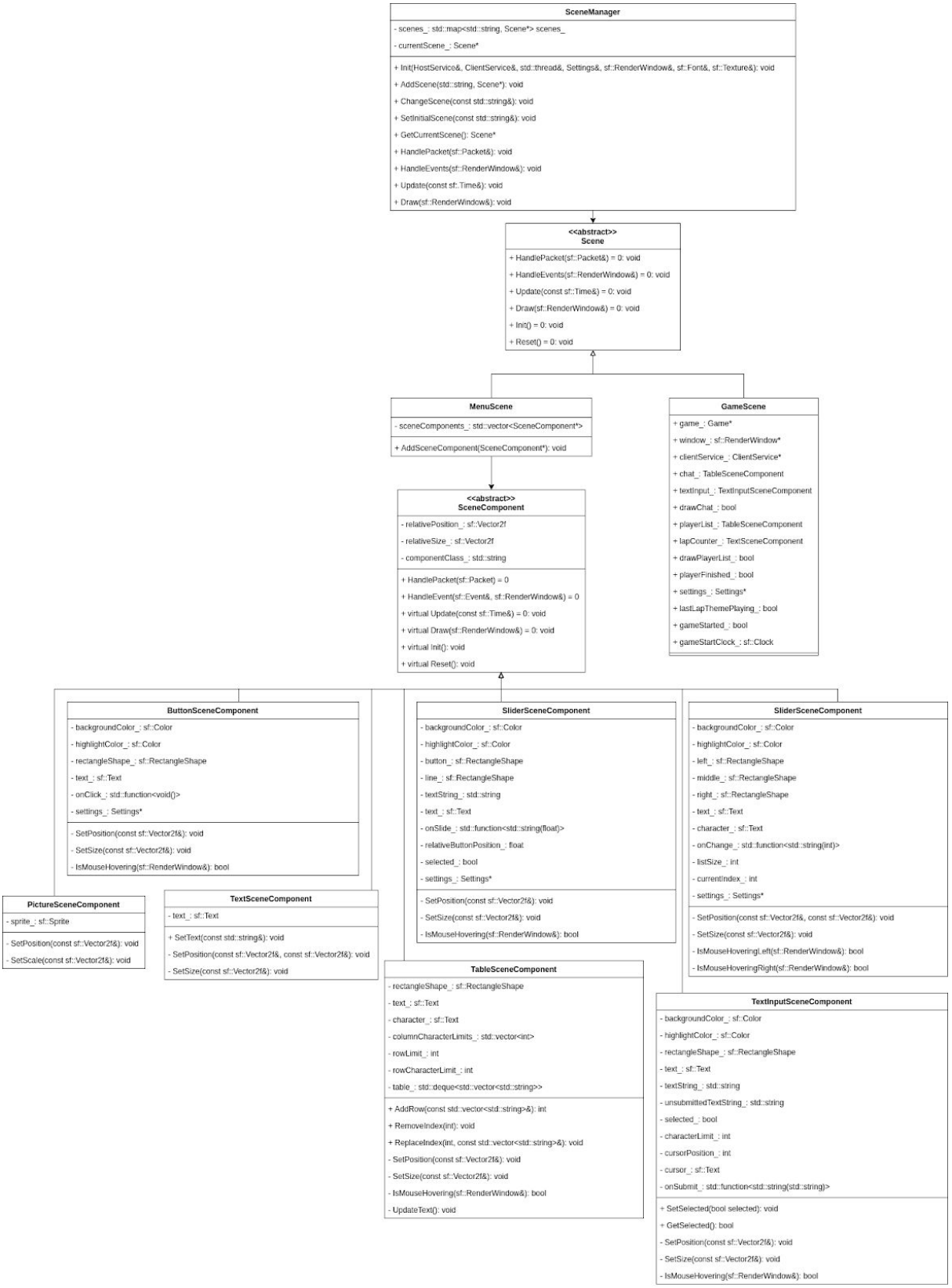
After loading the track the players are placed in their starting positions. When the starting sound effect plays, the race starts. The players must go around the track, and through invisible checkpoints which are specified in the map file. When a player completes their last lap, their car is despawned and their view is changed to full screen mode of the minimap.

After every player has completed their laps, they are moved to a scoreboard which shows the results of the game. They then can go back to the lobby and start a new game.

In the settings tab it is possible to change volume level, size of the window, player username, and toggle between fullscreen and windowed mode.

Software structure

Scene system



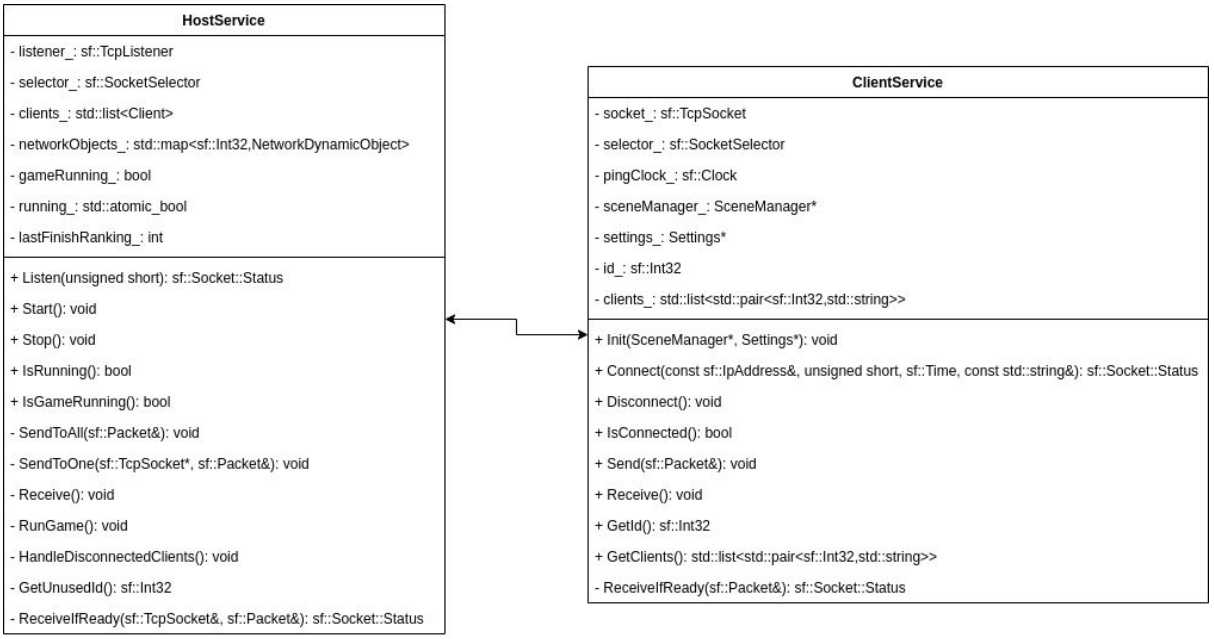
The scene system of the game allows for transitions between different scenes of the game, such as the game scene and menu scenes.

MenuScenes consist of SceneComponents which are drawable objects with their own functionality. For example TextInputSceneComponents allow for user text input and PictureSceneComponents allow for the drawing of pictures. SceneComponents are also used in the GameScene to a limited extent.

The SceneManager class manages, which scene is currently selected. The scene system as a whole makes sure that all events and network packets reach the currently selected scene and its components.

The GameScene is its own class which represents the scene of the game. Game functionality is further divided into a Game class which handles the non-drawing side of the game.

Network system



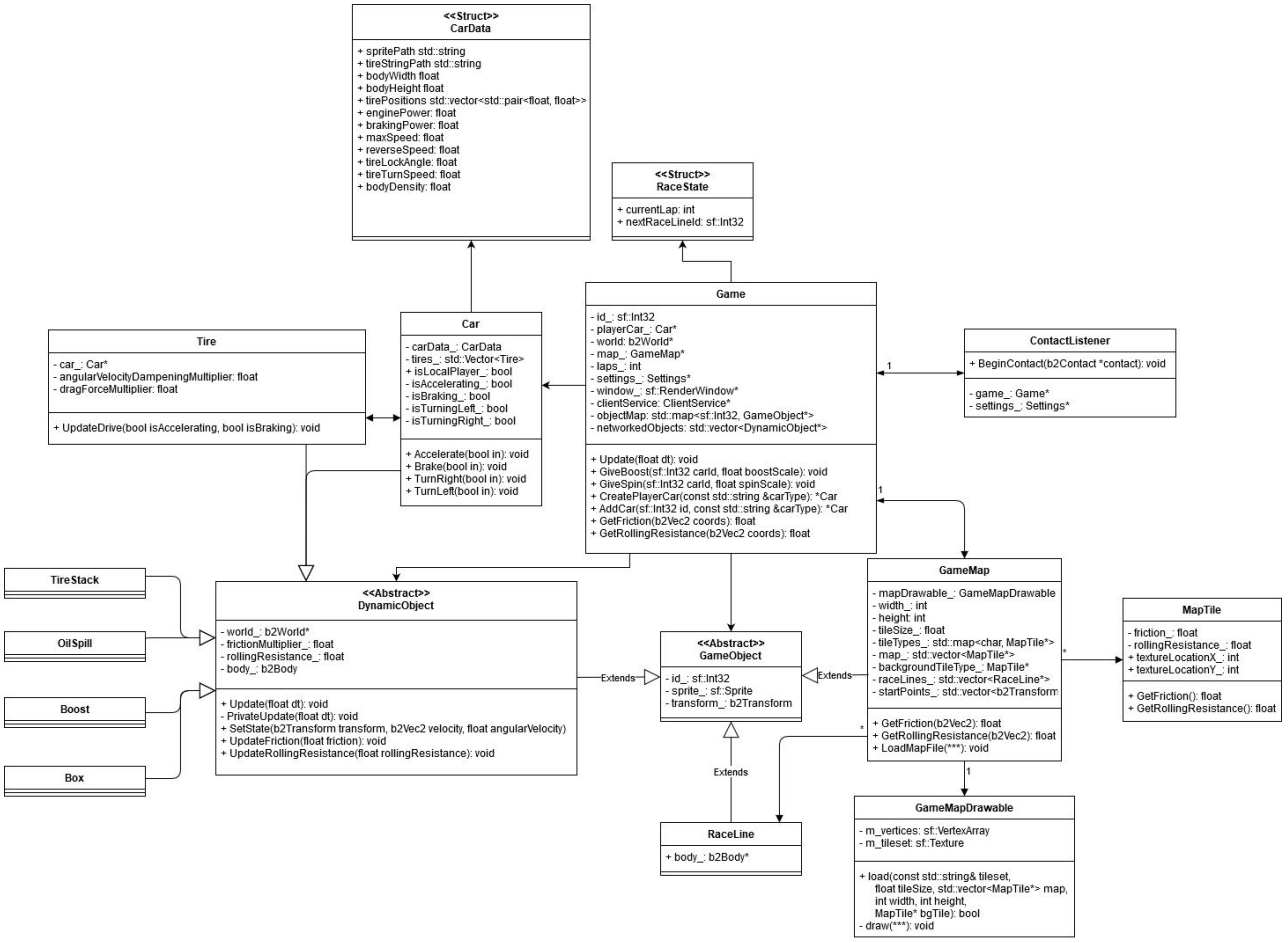
The network system works by allowing one client to host on their computer. The hosting client acts as a master version of the game, which provides data on moving objects and other clients to other clients. Using a master client as the absolute truth, no major desyncing will happen.

The HostService runs in a separate thread to the main program. The only way the host thread and the main thread of the host communicate are through an atomic boolean variable indicating whether the host thread is running and through sockets.

Messages are sent with a network message type system. The type of the message indicates to the handler of the message how the message is to be handled.

The network system does not provide much cheating prevention. Using the game normally shouldn't allow for anything wrong to happen, but in case of malicious use, the network system could be abused.

Game structure



Game logic

Game logic works completely separately from the GUI. An interface is implemented to class Game to call updates in the world and to set object states to be in sync with the network.

Game class
The Game class is the main manager of the game state. It holds information about the map and all objects in it, including player cars. Box2D's b2World is used to simulate the physics of the game. Game's Update method and other game-state affecting methods are mostly called by the GUI class GameScene.

RaceLine
Invisible "checkpoint" in the world. All of the RaceLines need to be crossed in order for the lap to count.

RaceState struct
Holds information of the current lap and the id of the next race line in order. Each player gets assigned one RaceState in Game.

ContactListener class
Extends Box2D's b2ContactListener. Listens to collisions in the world, and filters them based on the objects that collided. Certain collisions trigger different behaviours, such as Car and Raceline updates the car's RaceState, Car and Boost calls for a forward impulse on the car etc.

GameObject class
The abstract GameObject class represents an entity in the game. It has a sprite and a transform (position and rotation). All entities in the game world are derived from GameObject. Every GameObject has a unique ID which is used to access them easily. This is useful when syncing the states of different objects through the network.

DynamicObject class
The abstract DynamicObject class represents a physics entity in the game world. It is derived from GameObject, and in addition has a Box2D body (including fixture and shape for collisions). Its Update() method is the same for all DynamicObjects in the active Game. PrivateUpdate() method is also called similarly, but is modified by each child class to fit for their specific needs. DynamicObjects also keep track of their current friction and rolling resistance in the simulated Box2D world.

Car and Tire classes
Car and Tire are further derived from DynamicObject. Car holds references to its Tires, where the actual physics simulation is done. Tires apply force forward or backward depending on the player's input. Tires also apply sideways impulses to kill lateral velocity while still allowing a little bit of skidding, depending on the surface characteristics. The tires are attached to the Car body with Box2D Joints, which also control the rotation of the front tires, again depending on the player input.

CarData struct
The CarData struct holds information about the car it represents, for example size, sprite location, engine power, top speed and steering lock angle. The parametrization of the car characteristics into a struct allows easily reading the data from a file, adding new cars and simplification of the code.

Box, TireStack, Boost, Oilspill
These classes are derived from DynamicObject-class. Box, TireStack, Boost and Oilspill are objects that affect the gameplay. It is possible to collide with boxes and a tire stacks, boxes are movable and tire stacks static. Boost and Oilspill affect car handling: Boost gives the car a small nudge forward, and Oilspill throws the car sideways.

Map system

The maps in the game are saved in JSON format. Maps consist of different tiles that have properties such as the texture, friction and rolling resistance. These are also defined and

saved in a separate JSON file. The tiles are arranged in a rectangular grid with a defined width and height. The areas outside the grid fall back to a defined default tile.

GameMap Class
The class GameMap implements all of the main functionality that the map of a game needs. The main job of the class is to load the map and provide an interface to the data that the map contains.

The loading process loads the following into the game:
- Different tiletypes from the tiletype JSON file
- The tile layout of the map
- The size of a tile in Box2D units
- Starting points for cars in a race
- Racelines, which contain the finish line and checkpoint lines
- Other objects such as boosts, tirestacks and oil spills
- Networked physics objects such as boxes

GameMapDrawable Class
The map is rendered using a VertexArray from SFML. The vertices make up squares, which are used to render single tiles. The textures of the tiles are saved to a single texture image, which is loaded only once for the whole map. This makes the rendering of many tiles much faster. This functionality was implemented into the class GameMapDrawable, which extends the SFML primitive drawable types.

Settings

The Settings-class manages different settings. The Settings-class takes care of the player's name, volume, number of laps, the type of the car, maps, car index, sounds and the size of the game window.

Instructions for building and using

Required libraries are Box-2D and SFML
Installing Box-2D on Linux:
```
sudo apt-get install libbox2d-dev
```
Installing SFML on Linux:
```
sudo apt-get install libsfml-dev
```

Required build tools are Gcc and CMake
Installing Gcc on Linux:
```
sudo apt-get install build-essential manpages-dev
```
Installing CMake on Linux:
```
sudo apt-get install cmake
```

To run the program
Navigate to the root of the project (where src, res and others are located)

```
mkdir build
cd build
cmake ../src
cmake --build .
./MicroRally
```

How to use the software
The first window opening when running the software is the main menu. From there the user can choose to host a game, join a game, change the settings or to quit the game.

To host the game over the internet, the host must port forward the port 25000. If the game is played on a local network or alone, no port forwarding is needed.

If the user wants to join a game, they need to input the host address (usually their public ip address), and then chooses their car. The host is able to change the map and the number of laps for the race. The game is started by the host when every player has joined the lobby.

From the Settings tab the user can change volume, their name and window size.

Testing

Testing the software as much as possible was a key part of the development and was done after each new feature. A Trello board was used to track issues and planning.

Git feature branches were actively used when managing the development. Only properly working branches were merged to master using pull requests, which were usually reviewed by someone else. This made it easy to track the errors if they surfaced later in the development.

Testing was done mainly manually by running the program and playing the game. Some special testing environments were implemented, like a test map and a Play Now mode, to test features before everything they required was ready. Visual Studio Code's debugging mode was actively used to solve crashing and other issues. Valgrind was used to catch memory errors and fix them as they appeared. This was done before any major pull requests were merged to the master branch. GDB was also used in the most difficult situations.

Known issues

On Linux binding a listener socket to a port repeatedly fails. Windows doesn't have the same problem and it most likely has something to do with how the OS is configured. Even manually trying to use the setsockopt function to set SO_REUSEADDR didn't work. In practice this happens when the host quits the lobby or the game and tries to host again too quickly.

Sometimes when people are in a lobby and a player switches their car model at the same time as the host starts the game, the car types get desynched between clients. This happens because the other clients do not have enough time to receive the message about the changed car model, retaining the one that the player had initially. This does not break the game, but causes undesirable behaviour.

Work log

Week 0
Development environment working for team members
Project plan

Week 1
Markus (5h):
- Initial map loading from json
- Map rendering

Veeti (16h):
- Player input handling
- Basic car simulation with Box2D

Hilda (10h):
- Trying to get development environment work

Ville (20h):
- Plan UI with Elias
- Get initial scene system to work

Elias (9h):
- Plan UI with Ville
- Get development environment work

Week 2
Veeti (3h):
- Game theme composing
- Minor fixes and adjustments to physics

Hilda (7h):
- Settings

Ville (15h):
- Improve scene system
- Initial network system with chat and lobbies

Elias (7h):
- Get sounds work on windows

Week 3

Veeti (8h):

- Tweaks and fixes to physics and network syncing of the game state
    - Car tire position syncing using the body of the car as anchor
    - ID system for all GameObjects to easily access and modify any of them

Hilda (6h):

- Trying to figure out how objects should work
- Box

Ville (10h):

- Minimap
- Views
- Improve network system
- Improve scene system
- Game and network system compatibility

Elias (7h):

- Minimap, views, scenes with Ville

Week 4

Markus (6h):

- Finish and checkpoint line loading from map file
- Initial collision callback function
- GameObject refactoring

Veeti (8h):

- Traction calculation to car tires
- Hunting for a sneaky segfault
    - Uninitialized value was used in Box2D calculations resulting in a massive wave of NaNs
- Implemented easy way of adding new cars
    - New car “Truck”

Hilda (5h):

- Oilspill
- Boost
- Tirestack

Ville (12h):

- Sound effects with Elias
- Gui finishing
- Finish settings scene
- Improve networking with moving objects

Elias (7h):

- Polish sound effects and gui

Week 5

Markus (5h):

- Loading physics objects on level load from map file
- Background tile rendering
- Map scaling and other map file improvements

Veeti (12h):

- Racing logic to track which checkpoints each player has gone through
- Big map, “Ponsa GP”
- New car, “Golf”
- Rolling resistance calculation to car tires and tiles
- Body density parameter to cars
- Minor fixes around the project

Hilda (4h):

- Boost and oilspill affect the car
- Project documentation

Ville (8h):

- Game finishing
- Project documentation

Elias (4h):

- Project documentation

Week 6

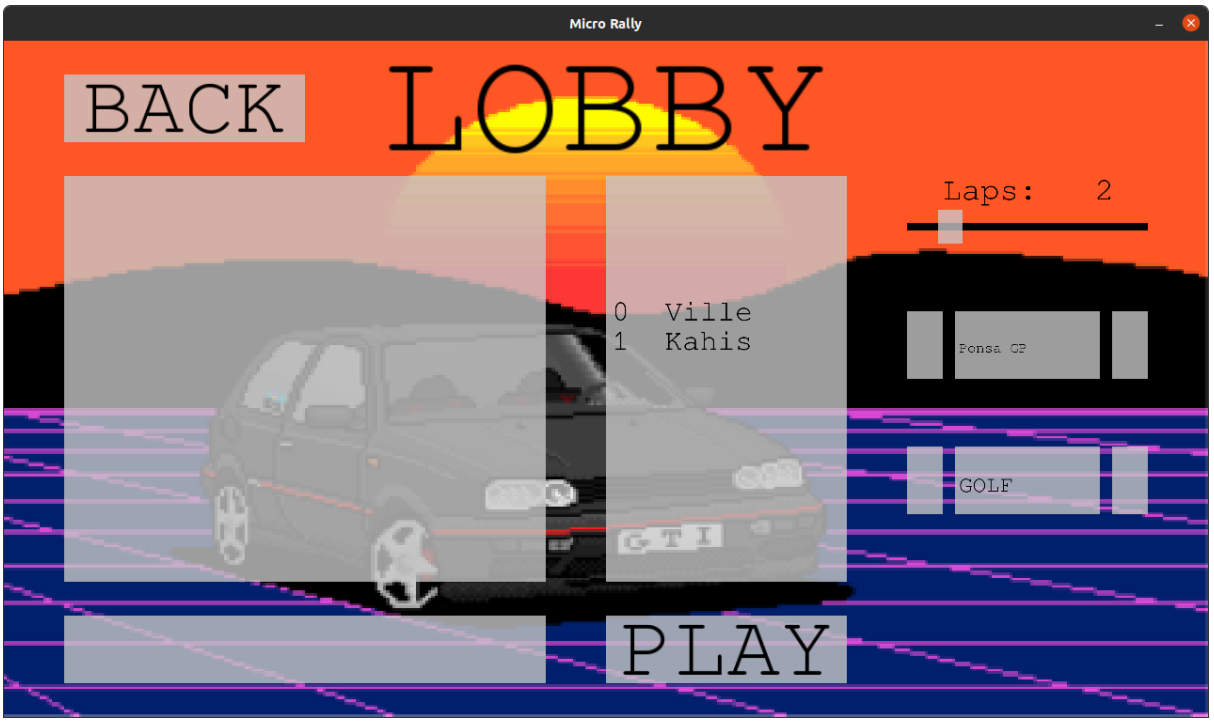
Project documentation

Demo

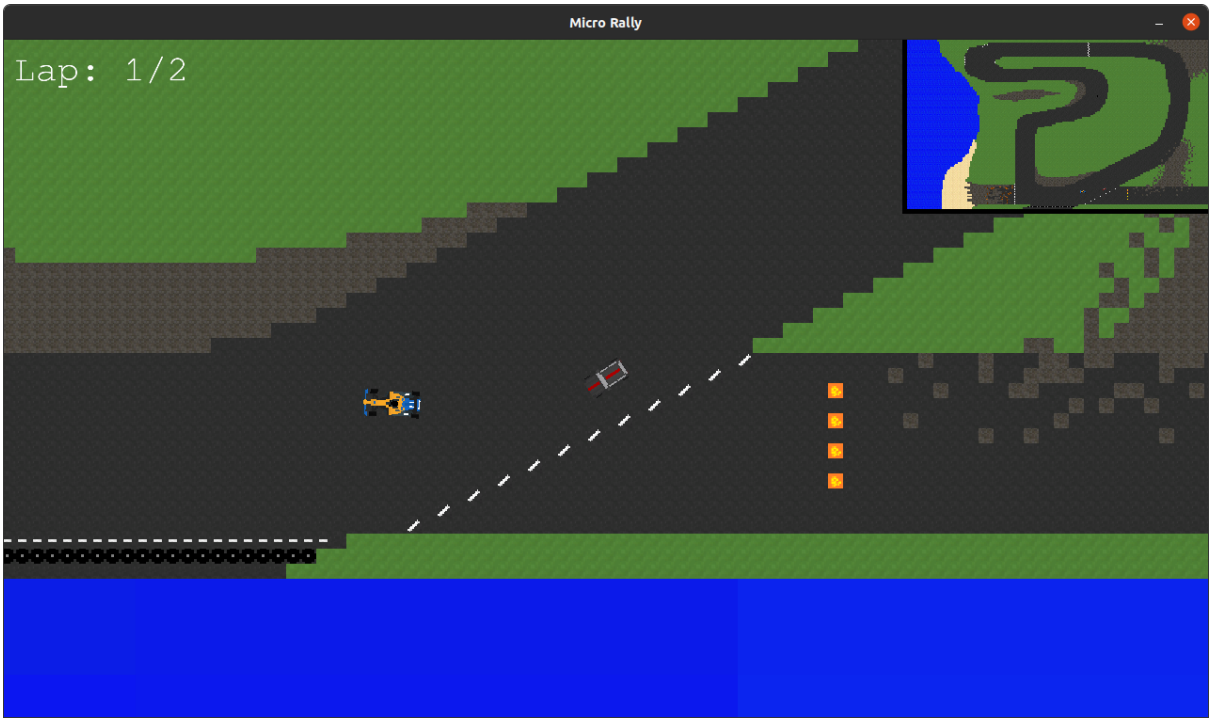
Screenshots

Menu scene

Settings scene

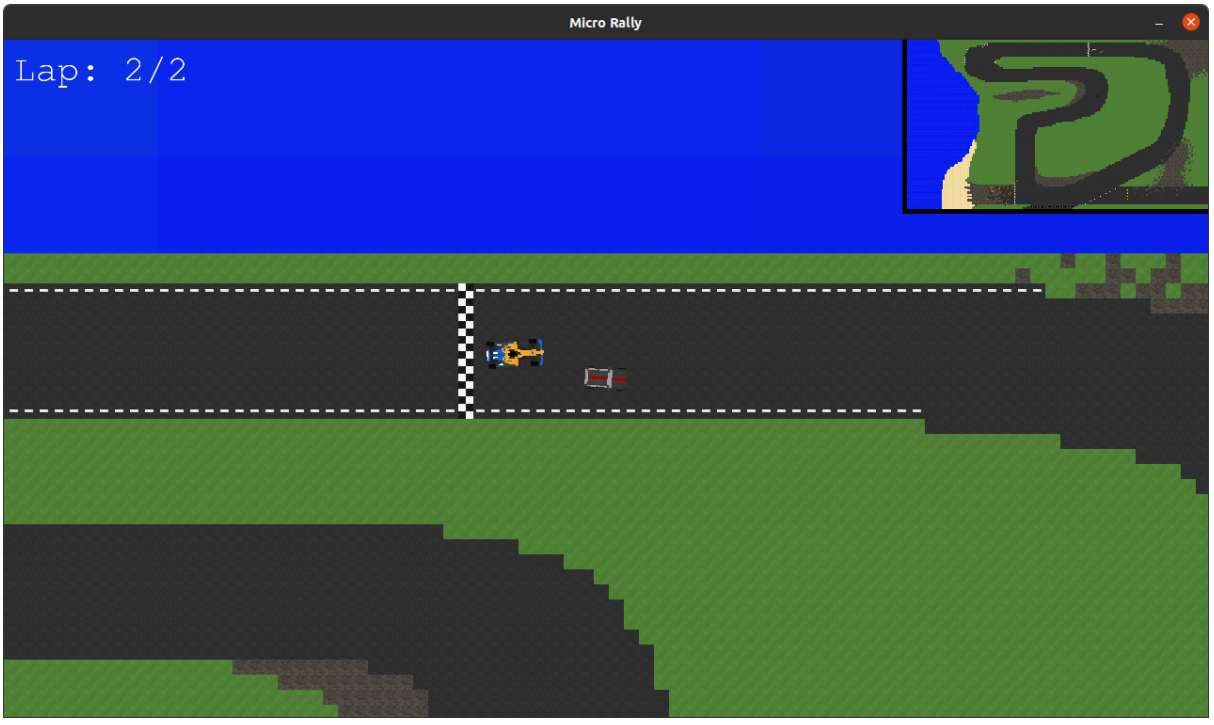


Lobby scene

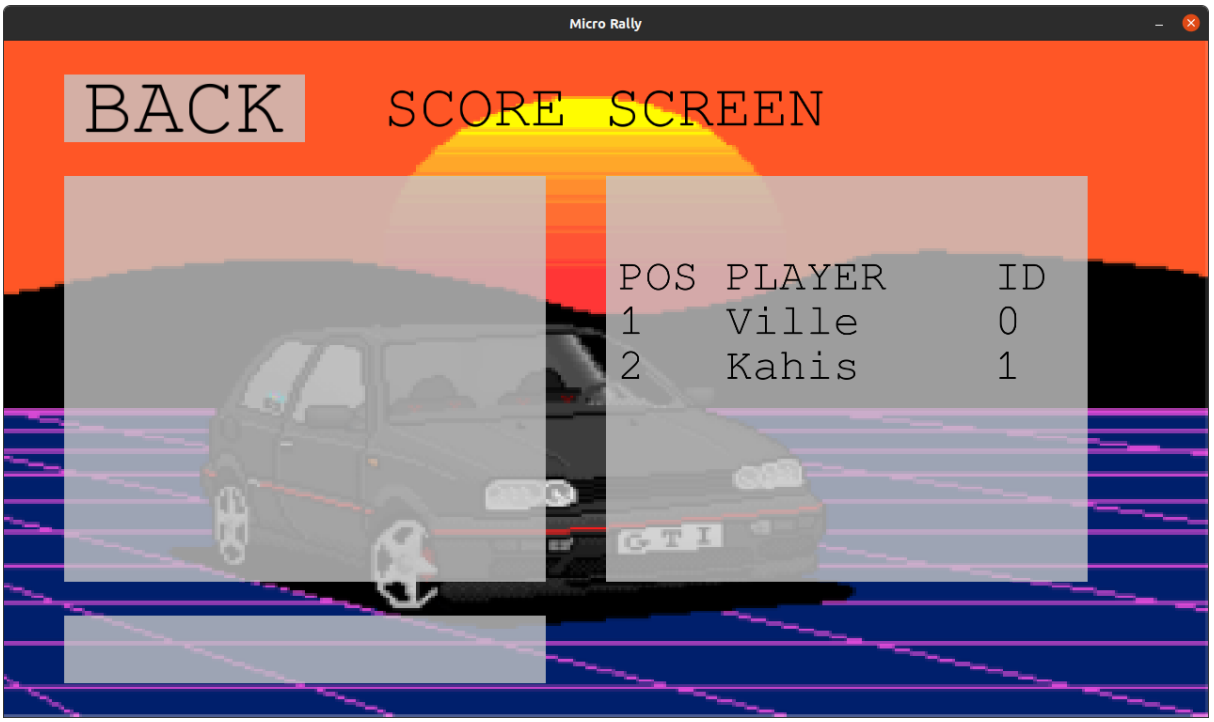


Game scene

Game scene



Game scene


Micro Rally
BACK
SCORE SCREEN
POS PLAYER ID
1 Ville 0
2 Kahis 1

Score scene