Micro Machine Game Plan

C++ course project

Huyen Do, Quan Hoang, Huyen Linh Nguyen, Duy To

October 30, 2023

1 Introduction

This C++ project aims to recreate the legendary game Micro machines, which is a popular old school racing game from the 90s available on many platforms. The following plan covers relevant aspects of the process of developing this project.

This plan consists of 6 main parts. The first section "Scope of work" discusses general game flow, and features and functionalities that will be implemented. The second section "High level structure" gives an overview on the class structure. Then, useful external libraries are discussed in the section "External libraries". The last two sections are "Division of work and responsibility" and "Schedule and milestone".

2 Scope of work

2.1 Game flow

When the player opens the game, a lobby will pop up. This lobby serves as the game menu. First, the player can add their name and choose their character from five default character: Cat, Dog, Buffalo, Dragon and Snake. These characters does not affect game play but makes it more interesting. Next, the player will select a map to race in from our collection. There are six options: Forest, Sea, Mountain, Otaniemi, Desert or a random map. After choosing the map, the player will select their vehicle Car, Boat, Feet or Skateboard. These vehicles have different properties such as maximum speed, power and map compatibility. Map compatibility means that choosing the suitable vehicle for a map will result in better speed and choosing the wrong one decreases the max speed of that vehicle in a race. After completing their character, the player can choose to add more human player or to add bot. There can be maximum four players, including human and bot in a race. If a human player is chosen, the process of constructing a character repeats. If a bot is chosen, a predefined bot will be added as a player. When everything is ready, the player press START and the race begins.

When the race menu, a countdown will appear. When the countdown finishes, the players start racing using their keyboard. The first racer to finish 5 rounds wins. In case no player accomplishes this goal, the player that completes the highest number of rounds in 15 minutes wins. Around the map, there are fixed and random booster that can give or limit a player' power in a certain amount of time. Fixed booster will always be at the same position in the map for any round. Random booster has random location and random appear time. The player can collect and activate boosters by driving through the booster's location. When a player meets the winning condition, the game ends. A scoreboard will pop up. It ranks the results of each player based on the number of rounds completed. The race ends and the players return to the lobby.

2.2 Game features

1. Basic features

    - Basic gameplay with simple driving physics

    - Multiple players on the same computer
    - Multiple tracks loaded from files
    - Game objects which affect gameplay

2. Additional features
    - Different performance of vehicles on different terrains
    - Different kind of vehicles
    - Sound effects

3. Advanced features
    - Random generated maps

2.3 Implementation

To accommodate this game flow and required features, the following functionalities will be implemented:

1. Running/Opening the game using Cmake
2. GUI and sound effects
    - Character customization using AI MidJourney Bot
    - Creating and importing/exporting maps
    - Randomized map: The environment is selected from our collection of map. The obstacles and roads are randomly generated using circle mechanism. There is one big circle and a smaller circle inside the big circle. We randomly generate a certain number of nodes. Connecting these nodes create the track and obstacles are placed on the track in a similar manner.
    - Fixed and random location for booster
    - Changing between different screens (Lobby and race track)
    - Sound effects for movements and interactions.
3. Game play
    - Player interaction with static body i.e map's elements and obstacles
    - Player interaction with dynamic body i.e other player/bot
    - Round counter
    - Time counter
    - Scoreboard data tracking
4. Bots
    - Smart and human-like racing behaviours
5. Multi-player
    - Adding new player
    - Split screen or network multiplier
6. Vehicle
    - Effects on the performance of the character
7. Player
    - Movement with keyboard

8. Booster
   - Functionalities if collected by a player
9. Handling exception
   - If no action in 5min, popup ”Are you there?”. If no response in 30s, the game ends.

3 High level structure

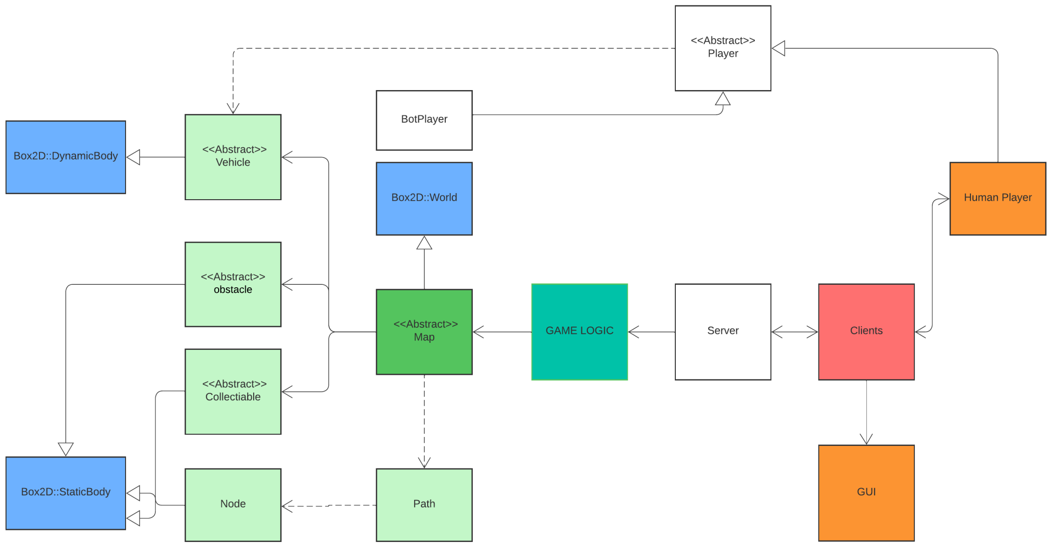

Figure 1: UML Class Diagram of Micro Machine

In this project, we will divide our class structure into three layers of abstraction:

3.1 Lowest Level of Abstraction

In the lowest level of abstraction, we define classes that handle the physics of the racing game:

- **Vehicle**: Extending from the dynamics body of Box2D, this class includes member functions that change the physical properties of the object.
- **Collectible**: Extending from the static body of Box2D, a Collectible will be destroyed upon contact with a vehicle. Taking polymorphism into account, we define a Collectible abstract class to quickly define various items in this game.
- **Obstacle**: Extending from the static body of Box2D, an obstacle can be used to define various barriers and obstacles on the racing track.

3.2 Intermediate Level of Abstraction

The next level of abstraction includes a major class and two helper classes:

- **Map**: Extending from the world of Box2D, generally an instance of a map represents a racing track while hosting and keeping track of various bodies. To achieve this, we will modify the original world from Box2D into a cyclic graph with two companion helper classes: **Node** and **Path**. By representing the map as a cyclic graph, we can create a randomized map creation scheme and extend it to a map editor. Similar to Collectible and Vehicle, Map is defined as an abstract class to accommodate different racing themes.

- **Node**: Extending from the static body of Box2D, each node represents a checkpoint on the track. Each node will have a sensor from Box2D to keep track of all vehicles.
- **Path**: Each path contains two Nodes. Intuitively, a path is a straight line between two nodes, and several lines of obstacles will be used to simulate the boundaries of the racing tracks.

3.3 Multiplayer Considerations

Taking the multiplayer aspect of the project into consideration, we will let the server handle all the calculations and send the results to each client. The clients will then take this minimal data and draw it into the window via SFML. In order to achieve this, we will define two classes for the server-side:

- **Player**: An abstract class that contains member functions that can change the behaviors of the vehicle, via a keyboard binding in the case of humans and a hard-coded moving pattern in the case of bots.
- **BotPlayer**: Extending from Player, this class represents a bot player with automated behavior.

3.4 Client-Side and Server-Side

Specifically on the client-side, we have:

- **HumanPlayer**: Extending from Player, this class represents a human player with keyboard control.

Specifically on the server-side, we have:

- **Game**: This is where we hold the game loop. The game object will have a container to keep track of all the players, both human and bot. Game object received and carry out instruction from each clients and product data so that the game sever.

This structure provides a clear and organized approach to developing a racing game with multiplayer functionality, emphasizing modularity.

4 External libraries

4.1 SFML

In this project, the SFML library will be utilized to construct the GUI, sound effects, as well as renders of maps, players and vehicles. SFML is a suitable choice for these objectives because of its ease to use and its various functionalities. Furthermore, it is compatible with other external libraries used in this project.

4.2 Box2D

The Box2D library is chosen to implement the game logic, the construction of objects and building the game world since the library has many suitable predefined classes. Moreover, as these classes are well-structured, it is possible for us to extend them to add more custom functionalities. The extensions of the predefined classes has been discussed in the section "High level structure".

4.3 ENet

For implementation of networked multiplayer, ENet library will be used. The advantages that make ENet a suitable library are its simplicity and reliability in delivering packets.

4.4 MidJourney AI bot

To generate graphics for characters and vehicles, MidJourney AI will be used. This AI is preferred because it produces consistent and high-quality results.

5 Division of work and responsibility

Team Member	Coding Responsibilities
Linh	Implementing player characters and their interactions. Also responsible for adding sound effects.
Duy	Coding the game's GUI, including menus and scoreboards. Also involved in multiplayer features implementation.
Huyen	Developing map generation, environmental interactions, and GUI design.
Quan	Implementing game physics, vehicle mechanics, and contributing to multiplayer coding.

Table 1: Team division of coding

Additional Responsibilities:

- **Documentation:** All team members are responsible for documenting their own code and contributing to the overall project documentation.
- **Team Lead:** Huyen will coordinate the team's activities, ensure deadlines are met, and act as the main point of contact.
- **Meeting Minutes:** Every team member will take turn to be in charge of writing and distributing the minutes for each team meeting. Keep Git repo in well-organized.

6 Schedule and milestone

Date	Task	Progress
Sat 28-10	Discuss the initial game concept and mechanics.	10%
Mon 30-10	Finish the comprehensive plan.	10%
Fri 3-11	Develop the project structure, create a code template, and outline function descriptions.	20%
Fri 10-11	Build core functions and a basic GUI for a single-player experience.	40%
Fri 17-11	Test the single-player mode and initiate the development of multiplayer features.	60%
Fri 24-11	Further develop multiplayer functionality and add extra features. Documentation craft.	75%
Fri 1-12	Work on final graphics, prepare for deployment. Project demo.	85%
Fri 8-12	Final adjustments, finalization, and preparation for submission.	100%

Table 2: Game Development Schedule