AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
ICHEP – Senior 1 Level – CESS
CSE354: Distributed Computing
Spring 2023

2D Multiplayer Racing Car Game

Submitted to:

Prof. Dr. Ayman Bahaa

Eng. Mostafa Ashraf

Submitted by:

Omar Mohamed Yousef (19P4953)

Omar Helmy Ibrahim Elbanna (19P3904)

GitHub Repo: mourra950/Distrubited-System-Racing-Cars (github.com)

YouTube Video: https://youtu.be/XfoahjVGTwg

Serene Cars Website: https://mourra950.github.io/Distributed-Car-Game-Site/

Game Download Link: https://drive.google.com/drive/folders/1frWMftkLvu9jjcvbRjIKaBYx2iYEl2MM?usp=sharing

Table of Contents

List of Figures

ABSTRACT

This is a documentation of the Distributed Computing initial project submission. This project aims to build a multi-player distributed 2D Car Racing Game along with chatting feature. The system must support multiple, autonomous agents (either human or automated) contending for shared resources and performing real-time updates to some form of shared state. The state of the system should be distributed across multiple client or server nodes. The system should be robust. The system should be able to continue operation even if one of the participant nodes crashes. This is our objective with this project, without delving into the technical details, as they will be thoroughly explained and assessed throughout the final submission.

The methods towards achieving such a goal will be clearly detailed as well as thoroughly described in terms of implementation throughout our development process. That is the purpose of this documentation, to know how to plan such a project and implement it as efficiently and accurately as possible.

1.0 INTRODUCTION

In this exciting project, we delved into the field of distributed systems to create a thrilling and immersive gaming experience. The objective of our project is to design and develop a cutting-edge 2D multiplayer racing car game that leverages the power of distributed systems to provide seamless gameplay, real-time interactions, and exhilarating racing competitions.

With the rapid advancement of technology and the increasing popularity of multiplayer gaming, the demand for immersive and engaging experiences has grown exponentially. Traditional single-player games have given way to multiplayer games that allow players from around the globe to compete, collaborate, and challenge each other. However, designing and implementing a multiplayer game presents a unique set of challenges, especially when it comes to ensuring low latency, synchronization, and scalability across a distributed network.

In our project, we aim to tackle these challenges head-on by harnessing the capabilities of distributed systems. By leveraging the power of multiple interconnected nodes, we can distribute the game logic, handle player interactions, and synchronize game states in several nodes. This not only enhances the overall performance and responsiveness of the game but also allows for a scalable architecture that can accommodate a reasonable number of concurrent players.

The key components of our 2D multiplayer racing car game will include a game server implemented using node.js, python proxy, and unity client game, and a communication protocol that facilitates real-time communication between the players and the proxy, and between proxy and the server. The game server will be responsible for managing player connections and handling game events. The client applications will provide the graphical user interface and render the game elements, while also interacting with the server through our python proxy that acts as a client to the server and a server to unity game to receive game updates and transmit player actions and chat messages.

Throughout the development process, we will focus on key aspects such as minimizing network latency, implementing efficient synchronization mechanisms, and designing a robust fault-tolerant system.

2.0 PROJECT DESCRIPTION

The aim of this project is to develop a distributed system for a 2D car racing game that incorporates interesting features from a systems perspective. The system will support multiple agents contending for shared resources and performing real-time updates to a shared state. By leveraging distributed systems principles, the project will ensure robustness, fault tolerance, and the ability to recover from node crashes.

The system will be designed to meet several important properties. Firstly, it should support multiple, autonomous agents participating in the 2D car racing game. These agents can be human players or automated entities that compete for shared resources and dynamically update the game's state in real-time.

Robustness is a crucial aspect of the system. It should be designed to handle failures gracefully and continue operation even if one of the participant nodes crashes. This fault tolerance ensures uninterrupted gameplay and prevents any loss of progress or data. Additionally, the system should support the recovery of a crashed player's state, allowing player to resume playing seamlessly.

The system will offer real-time playing and viewing capabilities, allowing participants to actively race against each other and observe the progress of other players simultaneously. This real-time interaction enhances the multiplayer experience and immerses participants in an engaging environment.

To facilitate communication between participants, chat functionality will be implemented. This feature enables players to exchange messages during, before, and after playing the game. It enriches the overall multiplayer experience.

Caching and copy migration techniques will be employed to optimize application response time. By caching frequently accessed data or game resources, the system can reduce latency and provide faster response times to participants. Additionally, copy migration can dynamically move game resources closer to the participants, reducing network latency and enhancing the overall gameplay experience.

3.0 BENEFICIARIES

Our project is developed for academic purposes with the intention of further enhancements for publication as a real game, can benefit several groups of individuals.

Gaming enthusiasts who have an interest in multiplayer games can benefit from experiencing the 2D car racing system. While not yet ready for commercial release, the project offers an engaging gaming environment where participants can compete against each other in real-time. The chat functionality enables communication between players, fostering social interaction and collaboration. Gaming enthusiasts can enjoy the game for entertainment purposes and gain an understanding of the potential features and challenges involved in developing distributed multiplayer games.

Students and researchers in the field of distributed systems are among the primary beneficiaries. The project provides them with practical learning experience, allowing them to apply the theoretical knowledge gained in their studies. By actively developing the 2D car racing system with distributed architecture, students can gain insights into the challenges and complexities of building robust and fault-tolerant systems. It serves as a valuable educational tool, enhancing their understanding of distributed systems principles, real-time updates, fault tolerance, and state management.

Although the project requires additional enhancements to be published as a real game, it offers valuable insights for game developers. Developers interested in creating multiplayer racing games with distributed architecture can leverage the project as a foundation. It provides a starting point, demonstrating the fundamental components and functionality required for a multiplayer game. Developers can build upon the project's structure, integrate their own gameplay mechanics, and enhance the user experience to create a fully-fledged, commercially viable game.

Finally, the project can contribute to the open-source community. By sharing the project's source code and documentation, it becomes a valuable resource for developers worldwide. The open-source community can study, modify, and improve upon the project, fostering collaboration and innovation in the fields of distributed systems and game development. Developers can learn from the project's insights and implementations, potentially leading to the creation of more robust and feature-rich distributed game systems.

4.0 DETAILED ANALYSIS

4.1 Components Structure

In the 2D multiplayer distributed systems racing car game project, a 3-tier layered architecture is employed, consisting of a server, an intermediate proxy, and a Unity front-end game. This section will provide a detailed analysis of the code structure, logic, and explanation of each component, focusing on the sockets used for communication.

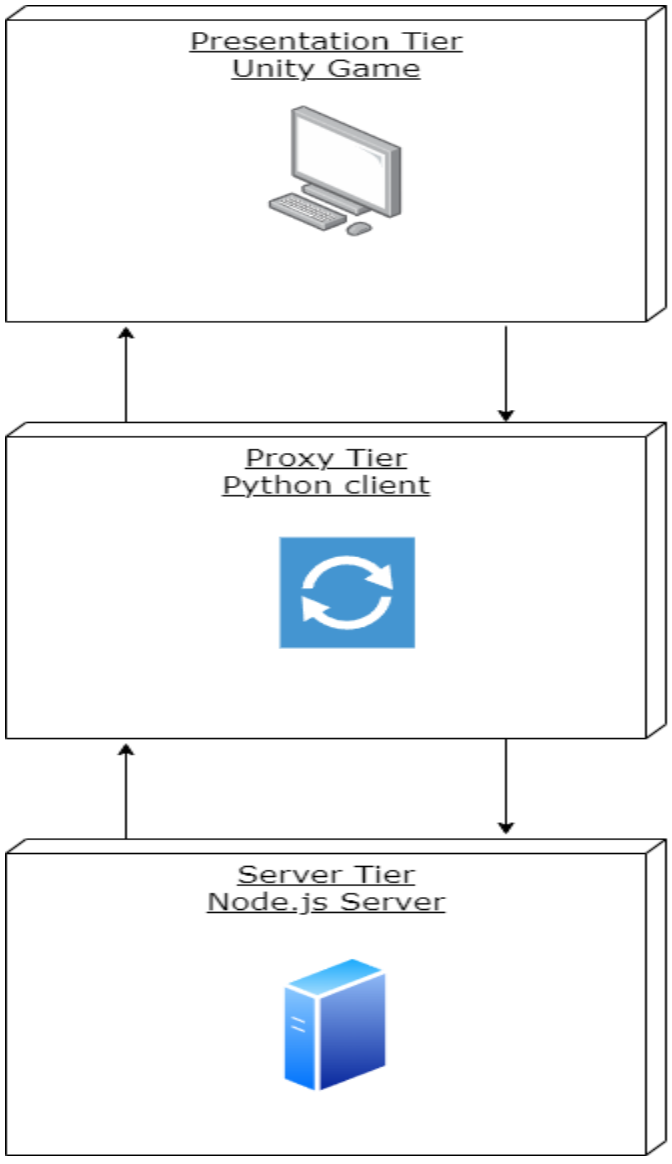


Figure 1 System Layered Architecture

4.2 Components Explanation

4.2.1 Node.js server

The server component is implemented using Node.js, Express, and Socket.io. It serves as the central hub for communication between the clients and manages the game state. Socket.io is used as the primary socket library for real-time bidirectional communication between the server and clients. The server's code structure follows an event-driven model, utilizing Socket.io's event-based system. It listens for various events, such as client connections, disconnections, and custom game events. Upon a client's connection, a socket connection is established, enabling the server to receive and send data to the client.

The server layer plays a critical role in coordinating the game logic and managing the shared state. It receives updates from the clients, processes them, and broadcasts the changes to all connected clients. By maintaining the authoritative game state, the server ensures consistency across all clients and facilitates a synchronized multiplayer experience.

The server consists of a single file app.js which handles all events.

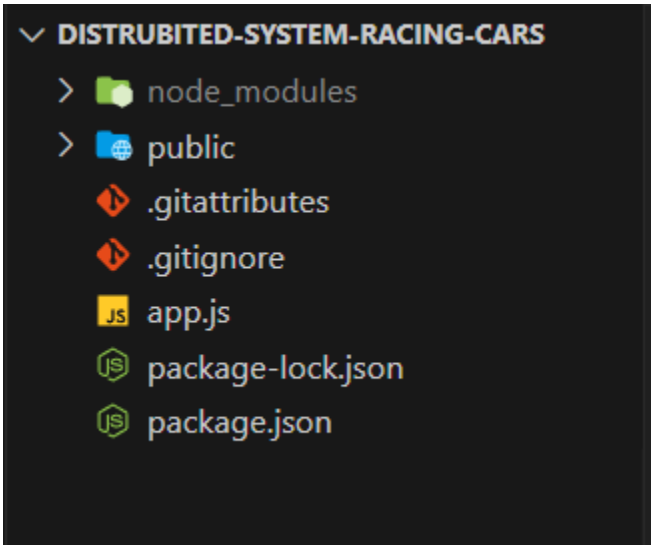


Figure 2 Server Code Structure

The code begins by importing the necessary dependencies, including Express, the HTTP module, and Socket.io. These libraries enable the server to handle web requests and establish WebSocket connections for real-time communication.

Next, the Express application is set up, creating an instance of the app. This app will handle the HTTP requests made to the server. The HTTP server is created using the Express app, allowing the server to listen for incoming HTTP connections.

The Socket.io server is then initialized by passing the HTTP server to the Server class constructor. This establishes the WebSocket connection and enables bidirectional communication between the server and connected clients. The io object represents the Socket.io server instance, which will handle the events and manage the communication with clients.

The code defines an event handler for the "connection" event. This event is triggered whenever a client establishes a connection with the server. Inside the event handler, a callback function is executed, which logs a message to indicate that a user has connected.

The code also includes several event handlers that listen for specific events emitted by the connected clients. These events include "my message", "CreateRoom", "joinRoom", "refreshplayers", "StartGame", "disconnect", "Coord", and "ChatRoom". Each event handler performs different actions based on the received event and data from the client.

For example, the "my message" event handler broadcasts the received message to all connected clients except the sender. This is achieved using the socket.broadcast.emit() function, which emits the event to all connected clients except the current socket.

The code also handles room management functionalities, such as creating and joining rooms, refreshing player lists, starting games, and managing chat messages within specific rooms. It utilizes the io.of("/").adapter.rooms object to retrieve information about the available rooms and their participants. Based on the received event and data, the server performs actions like joining a room (socket.join()), emitting events to specific rooms (io.in().emit()), and broadcasting messages to clients within a room (socket.to().emit()).

Furthermore, the code includes a disconnect event handler that logs a message when a client disconnects from the server. This allows for proper handling of client disconnections and cleanup of any associated resources.

To run the server, the code calls httpServer.listen(3000), instructing the server to start listening for incoming connections on port 3000.

4.2.2 Python proxy

The intermediate proxy, implemented in Python, acts as a bridge between the server and the Unity front-end game. It connects to the server using Socket.io, establishing a bidirectional communication channel. The proxy layer receives updates from the server and relays them to the Unity front-end game. Likewise, it receives commands and data from the Unity game and relays them to the server. This communication allows for real-time synchronization of the game state and actions between the server and Unity.

Python's socket.io library facilitates the bidirectional communication between the proxy and the server. It provides the necessary functions to handle socket connections, emit and receive events, and transfer data efficiently. By leveraging the proxy layer, the Unity front-end game can interact with the server seamlessly and stay synchronized with the authoritative game state.

The proxy consists of a single file client.py that contains 3 threads for receiving, sending data to unity, and for chatting.

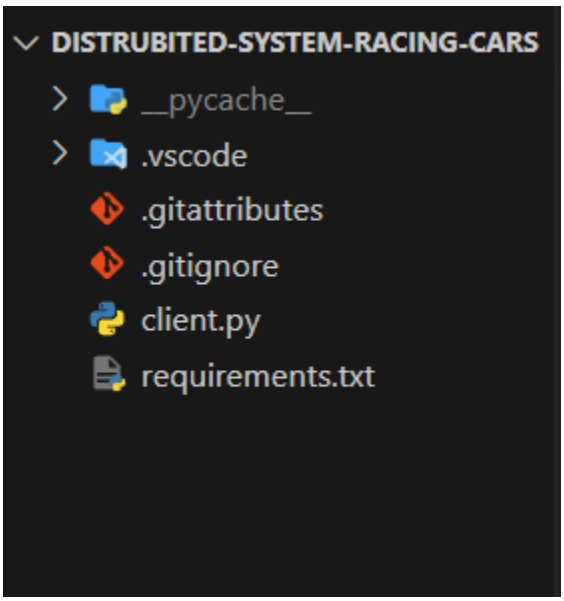


Figure 3 Proxy Code Structure

The code begins by importing the necessary libraries, including socketio and socket, which are used for handling real-time communication and low-level socket operations.

Next, a socketio client is created using the Client class, named sio. This client handles the WebSocket connection and communication with the Node.js server.

Several global variables are declared to store important information during gameplay, such as sendServer to handle low-level socket communication with Unity, unityChatSocket to handle chat messages, UserID to cache the user ID during the game, and RoomID to cache the room ID during the game.

The code defines event handlers using decorators (@sio.on and @sio.event) to handle various events received from the Node.js server via Socket.io. These events include "connect", "roomStatus", "GameStarted", "CoordBroadcast", "refresh", and "ChatBroadcast". Each event handler performs different actions based on the received event and data.

The code also includes utility functions such as unityReceive, unitySend, and Chat, which are executed in separate threads. These functions handle the low-level socket communication with Unity and manage chat messages between clients during gameplay.

In the main program entry point, the code establishes a connection to the Node.js server using the sio.connect method. This connects the client to the server's WebSocket endpoint.

Three threads are then created to handle different aspects of the client functionality: unityReceive for receiving messages from Unity, unitySend for sending messages to Unity, and Chat for managing chat messages.

Finally, the threads are started using the start method, and the client begins its operation, facilitating communication between the Node.js server and the Unity front-end game.

4.2.3 Unity Game

The Unity front-end game, developed in Unity using C# and .NET TCP sockets, is responsible for rendering the 2D car racing game and providing the user interface for players. It connects to the intermediate proxy using a TCP socket implementation in C# with the .NET framework. This TCP socket connection allows for reliable and efficient data transmission between the Unity game and the proxy.

The code structure in Unity follows an object-oriented approach, with classes and scripts managing various game elements, including player controls, car movements, rendering, and user interface components. The TCP socket implementation handles the exchange of data with the proxy layer, ensuring synchronized gameplay and communication between the Unity game and the distributed system.

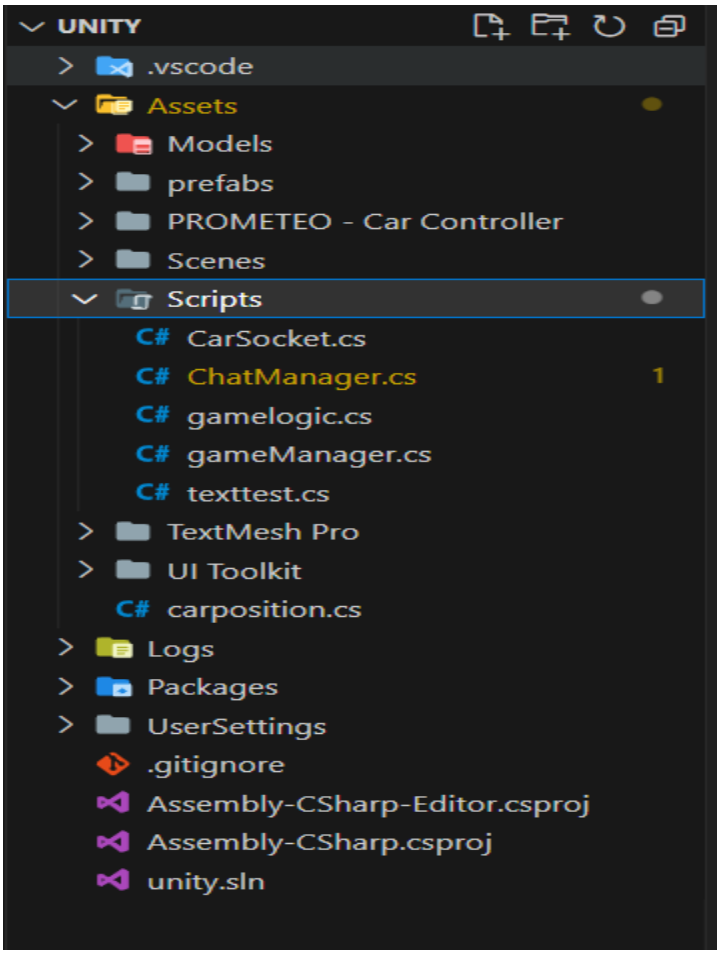


Figure 4 Unity Code Structure

The chat manager file is responsible for managing the chat functionality within the game. It facilitates the sending and receiving of chat messages between players and updates the chat box UI to display the messages in real-time.

The chat manager script has a reference to the UI elements, such as the chat input field and chat box, to interact with them and update their content.

At the start of the script, the necessary variables and references are defined. This includes a reference to the socket connection to communicate with the server, as well as variables to store the player's username, room ID, and other relevant information.

The script will have methods to handle user input, such as sending a chat message when the left control button is pressed. These methods will retrieve the text from the chat input field, package it appropriately and send it to the python proxy using the socket connection.

The script also checks for received data from proxy. When a chat message is received it will extract the message content, format it if necessary, and update the chat box UI accordingly. This is done by appending the new message to the existing chat log, scrolling to the bottom of the chat box to display the latest message, and applying desired styling to differentiate between different players' messages.

The game logic file is responsible for implementing the core gameplay mechanics, including the instantiation of players' car objects and handling the transmission of each player's car coordinates to be broadcasted to all other users.

The game logic file in Unity serves as the central component that manages the gameplay mechanics for the multiplayer racing game. It handles various aspects of the game, including the creation and management of players' car objects and the synchronization of their positions across all connected clients.

Upon initialization, the game logic file sets up the necessary variables and references to control the gameplay. This includes defining the car prefabs, spawn points, and other relevant parameters. It also establishes the connection to the server to exchange data and receive updates about other players.

As the game progresses, the game logic file continuously updates the position and movement of the player's car based on user input and other gameplay mechanics. It tracks the changes in the car's position and sends this information to the server for broadcasting to all other connected players.

To transmit the player's car coordinates, the game logic file communicates with the server through socket connections or other networking protocols. It packages the car's position data, including the position vector and rotation, into a message format that can be sent to the server. This data is then broadcasted to all other connected clients, allowing them to update the position of the corresponding player's car on their own game instances.

The gameManager file in Unity is responsible for managing the creation and joining of game sessions in the multiplayer racing game. It provides functionalities that allow players to create new game rooms or join existing ones.

The gameManager file serves as a central component that handles the game's overall flow and facilitates the creation and joining of game sessions. It provides the necessary functions and interfaces to enable players to initiate and participate in multiplayer races.

When the game starts or the player navigates to the game lobby, the gameManager initializes and displays the available options for creating or joining game sessions. It provides a user interface that allows players to interact with these options.

To create a new game, the gameManager provides a "Create Game" functionality. When selected, it prompts the player to enter specific details for the new game session, such as the room name or ID and any additional settings or parameters. The gameManager then communicates with the server to create the new game room, establishing a unique identifier for the session.

Alternatively, the gameManager provides a "Join Game" functionality to allow players to join existing game sessions. When selected, it prompts the player to enter a specific room ID. The gameManager communicates with the server to check the availability of the chosen game room and handles the necessary actions to join the selected session.

Upon successful joining of a game room, the gameManager manages the transition from the lobby to the actual gameplay scene. It coordinates with the server to synchronize the player's state, including their car position, with other players in the game session.

4.3 Layers Interaction

The interaction between the layers is facilitated through the defined socket connections. The server layer serves as the central point for communication and coordination. It interacts with the intermediate proxy layer through Socket.io, relaying game updates, commands, and other relevant events. The intermediate proxy, implemented in Python, establishes a connection with both the server and the Unity front-end game, acting as a mediator for data transmission.

The Unity front-end game layer communicates with the proxy layer using the TCP socket implementation in C#. It exchanges player commands, receives game updates, and chat messages, ensuring a seamless multiplayer experience. The TCP socket connection enables real-time data transfer and synchronization between the Unity game and the authoritative game state maintained by the server.

The 3-tier layered architecture and the use of different socket libraries (Socket.io for server-proxy communication and TCP sockets for proxy-Unity communication) facilitate efficient and reliable communication among the server, intermediate proxy, and Unity front-end game. This architecture ensures synchronized gameplay, real-time updates, and interactive multiplayer experiences in the 2D racing car game project.

5.0 TASK BREAKDOWN STRUCTURE

The task breakdown structure for our project involved several key stages and collaborative efforts between the two team members. The project began with thorough project planning and setup, where we defined the project scope, requirements, and objectives. We set up the development environment and selected the necessary tools to support the project's implementation.

During the design and architecture phase, we carefully analyzed the requirements and designed the overall system architecture. We identified the main components of the system, including the server, proxy, and Unity frontend, and determined how these components would interact with each other. Communication protocols and data formats were established to facilitate seamless data exchange between the components.

The server implementation was carried out incrementally, using a test-driven development approach. We started by setting up the basic server infrastructure using Node.js and Express. We integrated Socket.io to enable real-time communication between the server and clients. Gradually, we implemented the server's functionalities, such as handling room creation, joining, and management. Fault tolerance mechanisms were also put in place to ensure the server's reliability and resilience in the face of crashes or other issues. Throughout the development process, we wrote unit tests to validate the server's functionality and ensure its stability.

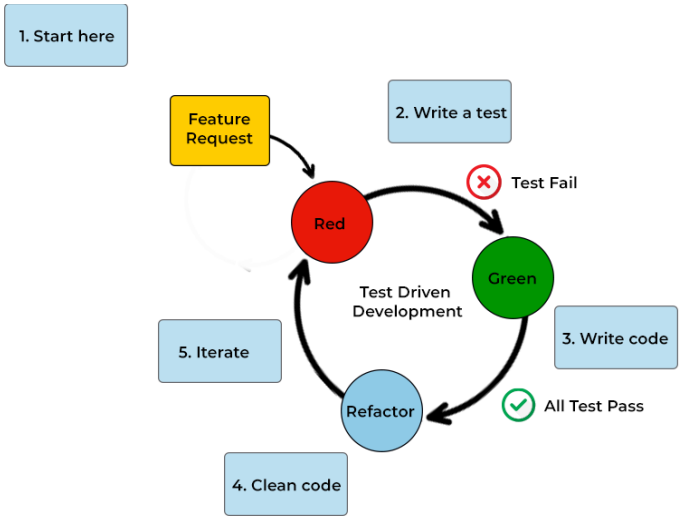


Figure 5 Test Driven Development

Simultaneously, we implemented the intermediate proxy component, which acted as a bridge between the Node.js server and the Unity frontend. We set up a Python environment and established a connection between the proxy and the server using Socket.io. The proxy was responsible for forwarding messages between the server and the Unity frontend, ensuring smooth communication between the components. Extensive testing was performed to validate the proxy's functionality and guarantee seamless integration with the other parts of the system.

The Unity frontend implementation involved creating the necessary game scenes, user interfaces, and visual assets. The team implemented the logic for rendering the racing track and cars, allowing players to control their vehicles. Integration of the C# .NET TCP socket communication enabled the exchange of commands and the reception of car coordinates and chat messages. We also implemented the display of real-time chat messages in the game's chat box, enhancing the multiplayer experience. Extensive testing was conducted to ensure the Unity frontend's functionality and its seamless integration with the server and proxy.

Integration and testing were crucial stages in the project. We brought together the server, proxy, and Unity frontend components to ensure smooth communication and overall functionality. Integration testing was performed to verify the end-to-end behavior of the multiplayer racing game. Any issues or bugs discovered during testing were promptly addressed to maintain the system's reliability and stability. Performance testing was also conducted to evaluate the system's scalability and response time, ensuring that it could handle multiple players and deliver a smooth gaming experience.

In the finalization phase, we conducted a final review of the project, ensuring that all requirements were met. We addressed any code quality issues, optimized performance, and applied best practices to finalize the codebase. The project was prepared for submission or publication, depending on the project's requirements or goals.

Regular collaboration and communication between the team members were essential for the successful completion of the project. By implementing the server component incrementally using a test-driven approach, the team ensured early detection of issues, maintained code quality, and iteratively improved the system's functionality. The task breakdown structure provided a clear roadmap for the project, guiding the team through the various stages and facilitating effective task completion.

6.0 SYSTEM ARCHITECTURE AND DESIGN

As mentioned in the previous section, three-tier layered architecture is adopted in our project. Adopting a three-tier layered architecture in the project brings several benefits and helps in organizing and managing the application effectively.

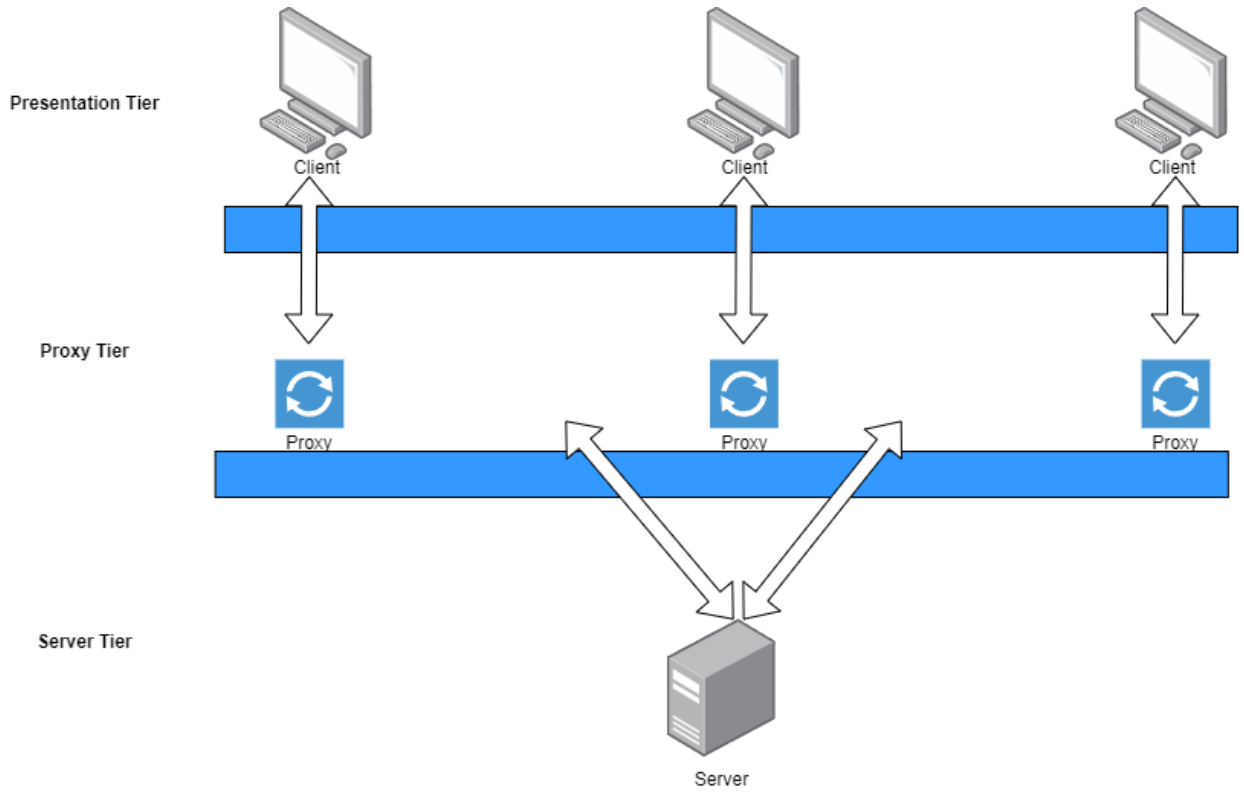


Figure 6 Three-Tier Layered Architecture with Multiple Clients

The presentation layer, also known as the front-end, is responsible for providing the user interface and capturing user inputs. In this project, the front-end is developed using the Unity framework. Unity handles the rendering of game graphics, user interactions, and input capturing. It provides a powerful and flexible platform for creating immersive game experiences. By separating the presentation layer, developers can focus on creating an engaging user interface and implementing the game's visual elements without being tightly coupled to the back-end logic.

The application layer, which serves as the middle layer, plays a crucial role in the architecture. In this project, the Python proxy server acts as the application layer. It acts as an intermediary between the front-end and the back-end. The proxy server receives user inputs from the front-end, processes them if necessary, and forwards them to the back-end server. It also receives game updates from the back-end and relays them to the front-end for visualization. By introducing an application layer, the architecture achieves a separation of concerns and promotes modularity and flexibility.

The back-end layer is responsible for managing the game logic, game state, and multiplayer synchronization. In this project, the back-end is implemented using Node.js. It receives user inputs and game updates from the proxy server, processes them, and applies the necessary game logic to update the game state. The back-end server also manages multiplayer synchronization to ensure that all players have a consistent view of the game world. The use of Node.js, with its event-driven and non-blocking I/O model, allows for efficient real-time communication and scalability.

By adopting a three-tier layered architecture, the project achieves a clear separation of concerns. The front-end focuses on rendering the game graphics and capturing user inputs, the proxy server handles communication and data processing, and the back-end manages the game logic and multiplayer synchronization. This separation allows for modularity, as each layer can be developed and maintained independently. It also promotes scalability, as multiple clients can connect to the back-end server through the proxy server, enabling real-time multiplayer gameplay. The architecture also offers flexibility by allowing the use of different technologies in each layer, enabling developers to leverage the most suitable tools for specific tasks and facilitating code reusability.

7.0 TESTING SCENARIOS

7.1 Room Creation with Existing Room ID

Objective: To verify the behavior of the system when a player attempts to create a room using a room ID that already exists in the game.

Preconditions:

1. The game is running and accessible to players.
2. At least one room with a specific room ID already exists in the game.

Test Steps:

1. Launch the game in the main screen.
2. Enter the room ID that is already in use by an existing room.
3. Click on the "Create Room" button to initiate the room creation process.

Expected Results:

1. The system should detect that the entered room ID already exists in the game.
2. The player should not be able to proceed with the room creation process.
3. The player should be able to enter a different room ID or join the existing room if desired.

Test case Passed.

7.2 Join a Non-existent Room.

Objective: To verify the behavior of the system when a player attempts to join a room that does not exist in the game.

Preconditions:

1. The game is running and accessible to players.
2. No room with the specified room ID exists in the game.

Test Steps:

1. Launch the game in the main screen.
2. Enter a non-existent room ID that is not currently in use by any room.
3. Click on the "Join Room" button to initiate the room joining process.

Expected Results:

1. The system should detect that the entered room ID does not exist in the game.
2. The player should not be able to proceed with the room joining process.
3. The player should be able to enter a different room ID or create a new room if desired.

Test case Passed.

7.3 Spectate a Non-existent Room.

Objective: To verify the behavior of the system when a user attempts to spectate a room that does not exist in the game.

Preconditions:

1. The game is running and accessible to players.
2. No room with the specified room ID exists in the game.

Test Steps:

1. Launch the game in the main screen.
2. Enter a non-existent room ID that is not currently in use by any room.
3. Click on the spectate button to initiate the room joining process.

Expected Results:

1. The system should detect that the entered room ID does not exist in the game.
2. The player should not be able to proceed with the room joining process.
3. The player should be able to enter a different room ID.

Test case Passed.

7.4 Creating a Room with a Unique ID Successfully and Scene Transition to Lobby

Objective: To verify the behavior of the system when a player successfully creates a room with a unique ID and ensures a smooth transition to the lobby scene.

Preconditions:

1. The game is running and accessible to players.
2. The player is on the room creation screen.

Test Steps:

1. Launch the game in the main screen.
2. Enter a unique room ID that is not currently in use by any existing room.
3. Click on the "Create Room" button to initiate the room creation process.
4. Verify that the room is successfully created with the entered room ID.
5. Observe the system for a smooth transition from the room creation screen to the lobby scene.

Expected Results:

1. The system should successfully create a room with the entered unique room ID.
2. The system should automatically transition from the room creation screen to the lobby scene.
3. The player should be able to start the game.

Test case Passed.

7.5 Joining an Existing Room Successfully and Scene Transition to Lobby

Objective: To verify the behavior of the system when a player successfully joins an existing room and ensures a smooth transition to the lobby scene.

Preconditions:

1. The game is running and accessible to players.
2. An existing room with the specified room ID is available for joining.

Test Steps:

1. Launch the game in the main screen.
2. Enter the room ID of an existing room that is open for new players to join.
3. Click on the "Join Room" button to initiate the room joining process.
4. Verify that the player is successfully joined to the specified room.
5. Observe the system for a smooth transition from the room joining screen to the lobby scene.

Expected Results:

1. The system should successfully identify and join the existing room with the entered room ID.
2. The system should automatically transition from the room joining screen to the lobby scene.
3. The player should be redirected to the lobby.
4. The player can’t start the game.

Test case Passed.

7.6 Spectating an Existing Room Successfully and Scene Transition to Lobby

Objective: To verify the behavior of the system when a player successfully joins an existing room and ensures a smooth transition to the lobby scene.

Preconditions:

3. The game is running and accessible to players.
4. An existing room with the specified room ID is available for joining.

Test Steps:

6. Launch the game in the main screen.
7. Enter the room ID of an existing room that is open for new players to join.
8. Click on the "spectate" button to initiate the room joining process.
9. Verify that the spectator is successfully joined to the specified room.
10. Observe the system for a smooth transition from the room joining screen to the lobby scene.

Expected Results:

5. The system should successfully identify and join the existing room with the entered room ID.
6. The system should automatically transition from the room joining screen to the lobby scene.
7. The spectator should be redirected to the lobby.
8. The spectator can’t start the game.

Test case Passed.

7.7 Chat Functionality Between Players and Spectators in the Lobby

Objective: To verify the behavior of the system when players and spectators can communicate with each other through chat functionality in the lobby.

Preconditions:

1. The game is running and accessible to players.
2. Players and spectators are present in the lobby.
3. The chat functionality is enabled and available in the lobby.

Test Steps:

1. Launch the game and navigate to the lobby scene.
2. Observe the presence of players and spectators in the lobby.
3. Verify that a chat interface or chat window is available for communication.
4. As a player, type and send a chat message in the lobby chat.
5. Observe the message being displayed in the lobby chat for all players and spectators.
6. As a spectator, type and send a chat message in the lobby chat.
7. Observe the message being displayed in the lobby chat for all players and spectators.
8. Ensure that the chat messages are visible to all participants in the lobby.
9. Verify that the chat messages include the name or identifier of the sender.
10. Test the ability to send and receive multiple chat messages consecutively.

Expected Results:

1. The lobby should display the presence of players and spectators, indicating their availability for chat.
2. The chat interface or chat window should be visible and accessible in the lobby.
3. As a player, the chat message sent should be displayed in the lobby chat for all players and spectators.
4. As a spectator, the chat message sent should also be displayed in the lobby chat for all players and spectators.
5. All participants in the lobby should be able to see and read the chat messages in real-time.
6. The chat messages should include the name or identifier of the sender, allowing easy identification.

7. Participants should be able to send and receive multiple chat messages consecutively.

Test case Passed.

7.8 Chat Functionality Between Players and Spectators During the Game

Objective: To verify the behavior of the system when players and spectators can communicate with each other through chat functionality during the game.

Preconditions:

1. The game is running and accessible to players.
2. Players and spectators are present in the game.
3. The chat functionality is enabled and available during the game.

Test Steps:

1. Launch the game and join a game session as a player or spectator.
2. Verify that the game interface includes a chat interface or chat window.
3. Ensure that players and spectators are visible within the game.
4. As a player, type and send a chat message during the game.
5. Observe the message being displayed in the game chat for all players and spectators.
6. As a spectator, type and send a chat message during the game.
7. Observe the message being displayed in the game chat for all players and spectators.
8. Ensure that the chat messages are visible to all participants during the game.
9. Verify that the chat messages include the name or identifier of the sender.
10. Test the ability to send and receive multiple chat messages consecutively during the game.
11. Test the ability to scroll through chat history and view previously sent messages.

Expected Results:

1. The game interface should include a visible and accessible chat interface or chat window.
2. Players and spectators within the game should be visible and identifiable.
3. As a player, the chat message sent during the game should be displayed in the game chat for all players and spectators.

4. As a spectator, the chat message sent during the game should also be displayed in the game chat for all players and spectators.
5. All participants during the game should be able to see and read the chat messages in real-time.
6. The chat messages should include the name or identifier of the sender, allowing easy identification.
7. Participants should be able to send and receive multiple chat messages consecutively during the game.
8. The chat interface should allow scrolling through chat history to view previously sent messages.

Test case Passed.

7.9 Excluding Car Instantiation for Spectators

Objective: To verify that the game does not instantiate cars for spectators, ensuring they do not occupy a car object within the game world.

Preconditions:

1. The game is running and accessible to players and spectators.
2. Both players and spectators have joined the game session.

Test Steps:

1. Launch the game and join a game session as a spectator.
2. Observe the game world and verify that no car objects are instantiated for the spectator.
3. Verify that the spectator does not have control or interaction capabilities with any in-game cars.
4. Observe the behavior of the game when cars are instantiated for the players.
5. Verify that the cars instantiated for players are visible and controllable by the respective players.

Expected Results:

1. The game should not instantiate car objects for spectators.
2. The spectator should not have control or interaction capabilities with any in-game cars.
3. The game should behave normally and continue to function without any errors or inconsistencies, even without car instantiation for spectators.
4. The cars instantiated for players should be visible and controllable by the respective players.
5. Spectators should be able to observe the game world and the actions of the players without occupying a car object.

Test case Passed.

7.10 Sending and Receiving Coordinates Among All Players

Objective: To verify that all players in the game can send and receive coordinate data, enabling real-time location tracking and synchronization among the players.

Preconditions:

1. The game is running and accessible to multiple players.
2. The coordinate sending and receiving functionality is implemented and enabled for all players.

Test Steps:

1. Launch the game and ensure multiple players have joined the game session.
2. Verify that each player's coordinate data is sent and updated in real-time.
3. As a player, send your current coordinates to other players in the game.
4. Observe and verify that the other players receive and display your coordinates correctly.
5. Repeat the process for other players, where each player sends their coordinates to others.
6. Observe the behavior when multiple players send their coordinates simultaneously.
7. Verify that all players can receive and display the coordinates of other players in real-time.
8. Test scenarios where players move or change positions and verify that their updated coordinates are correctly sent and received by other players.
9. Test scenarios where players stop or pause their movement and verify that their stationary coordinates are correctly shared with others.
10. Verify the accuracy and precision of the coordinate data being sent and received, ensuring that the positions of players are synchronized correctly.

Expected Results:

1. All players in the game should have access to the coordinate sending and receiving functionality.
2. Each player's coordinate data should be sent and updated in real-time.
3. Players should be able to send their current coordinates to other players.
4. The coordinates sent by each player should be received and displayed correctly by other players.
5. Multiple players should be able to send and receive coordinate data simultaneously without any conflicts or errors.

6. The received coordinates should reflect the real-time movements and positions of other players accurately.
7. Changes in player positions, including movement and stationary positions, should be accurately reflected and synchronized among all players.

Test case Passed.

8.0 END-USER GUIDE

Before starting the game, ensure that you have Python 3.10 installed on your PC. You can check the installed version by running the following command in the command prompt (CMD): "python --version".

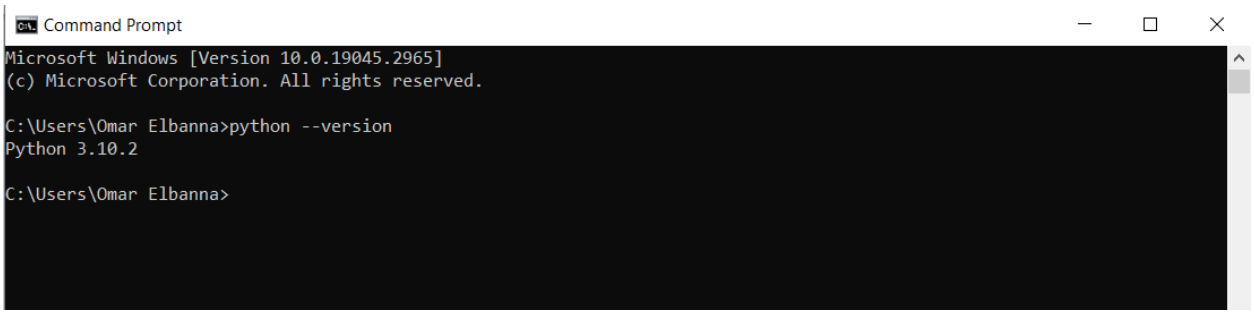


Figure 7 Checking Python Version

Download the game from the given link. You will find the following directory.

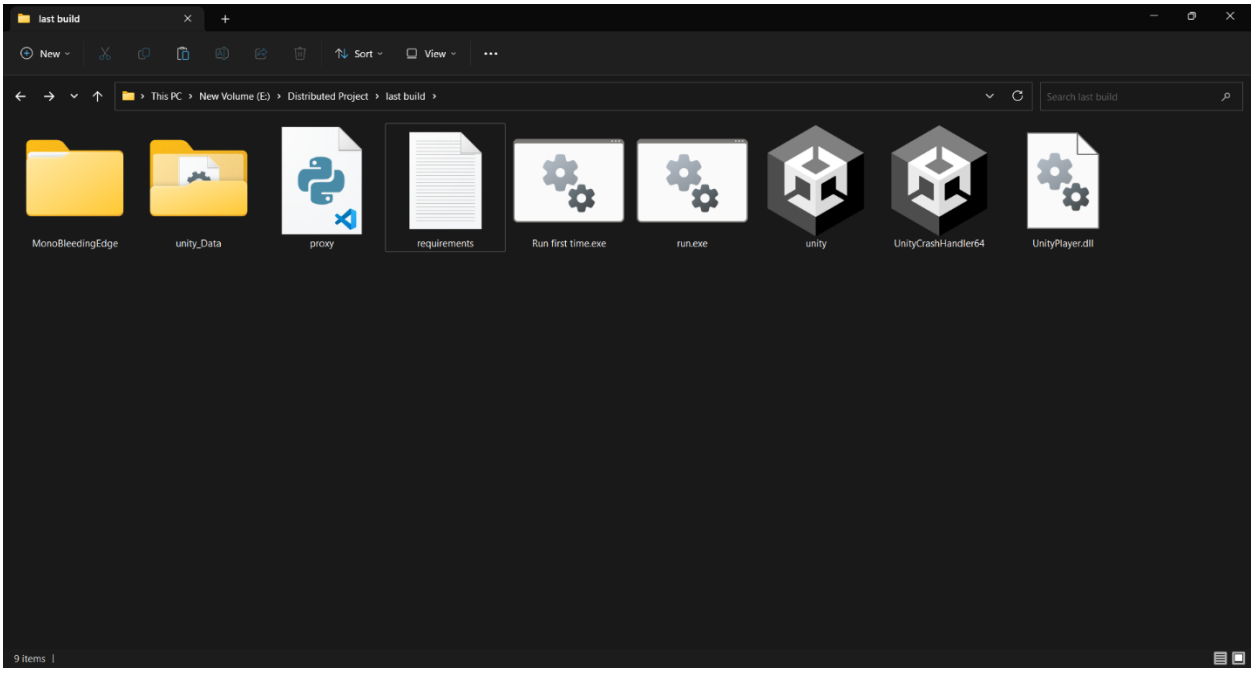


Figure 8 Game Directory

If this is your first time running the game, follow these steps:

1. Double click on "run first time.exe". This will open a CMD window and automatically install all Python dependencies and requirements needed for the game.

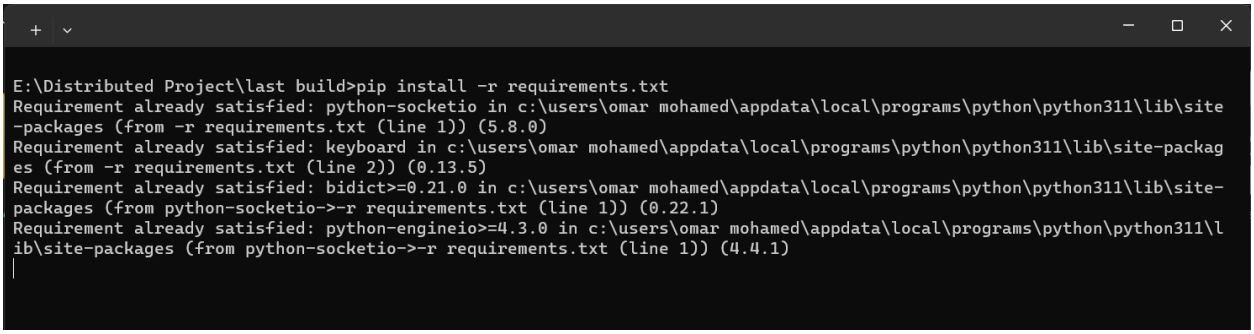


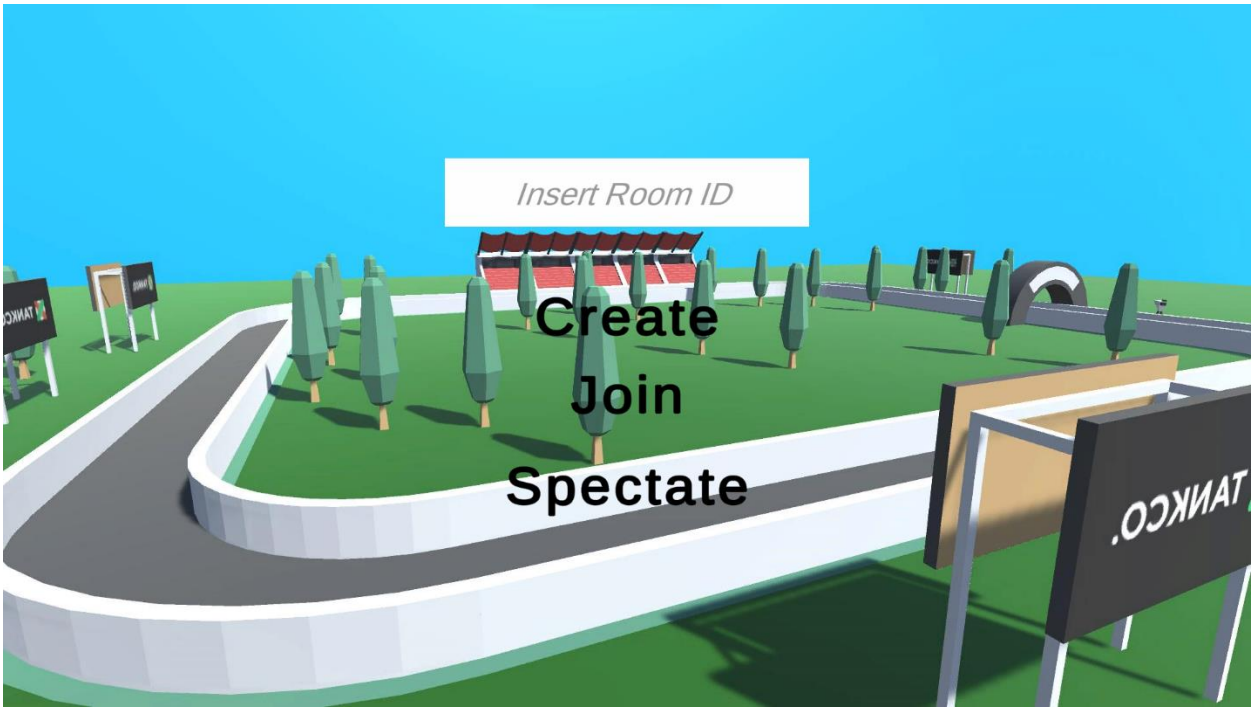
Figure 9 Installing Python Requirements

To launch the game:

1. Double click on "run.exe". This will open the game window.

Main Window:

1. The game will open in the main window, where you have various options.



To create a room, enter your room ID then click on “Create” button.

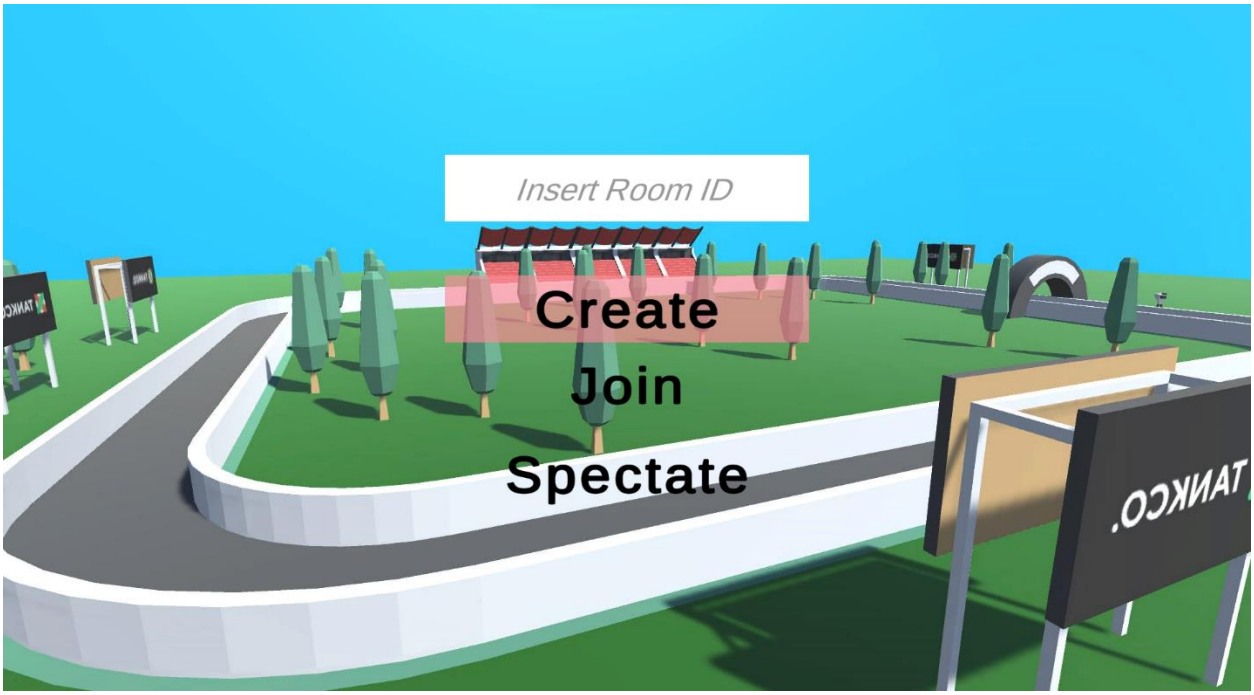


Figure 10 Create Room Button

To join a room, enter room ID then click on “Join” button.

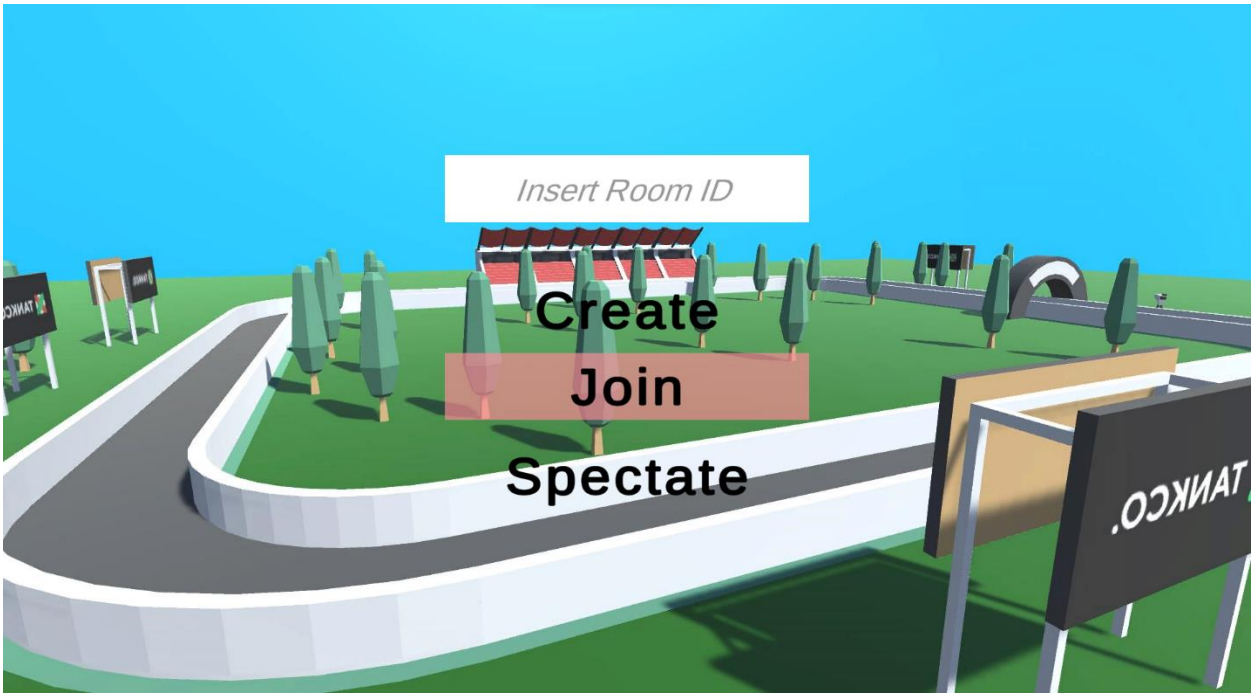


Figure 11 Join Room Button

To spectate a room, enter room ID then click on “Spectate” button.

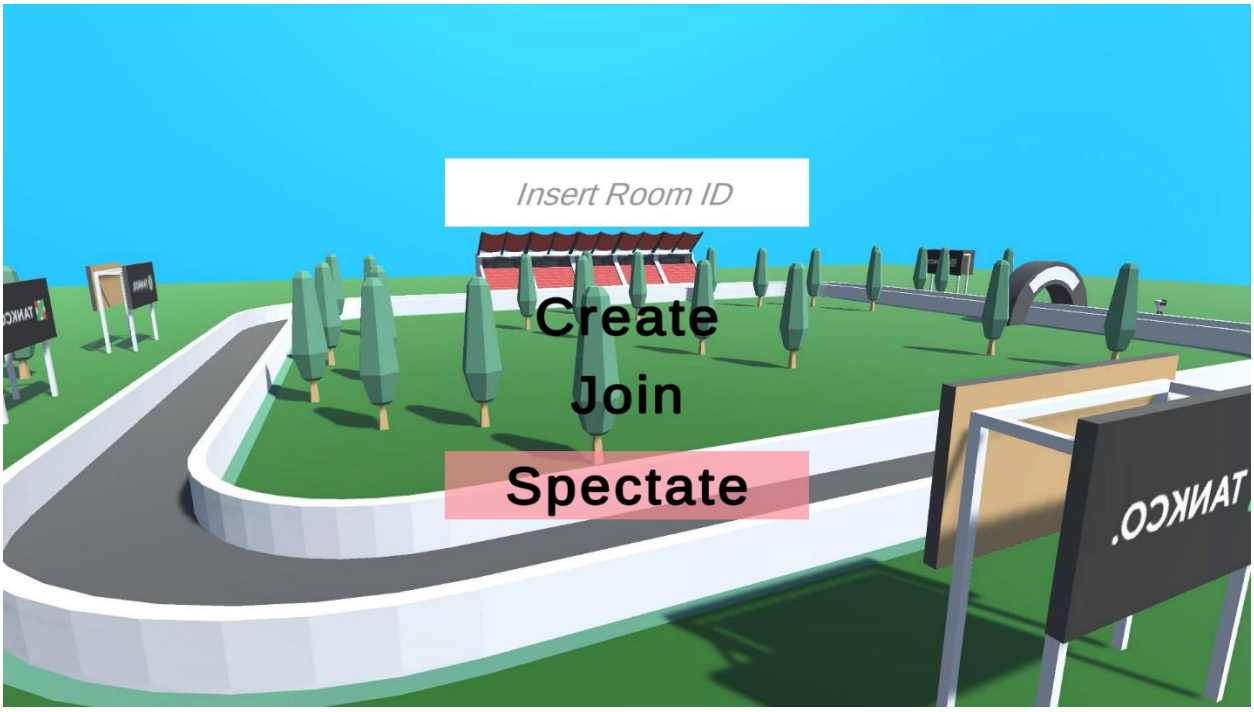


Figure 12 Spectate Room Button

If you have entered a unique room ID and presses create, the room will be created successfully, and you will be directed to the lobby.

You can see the number of players in the room and your room id to share it with your friends.

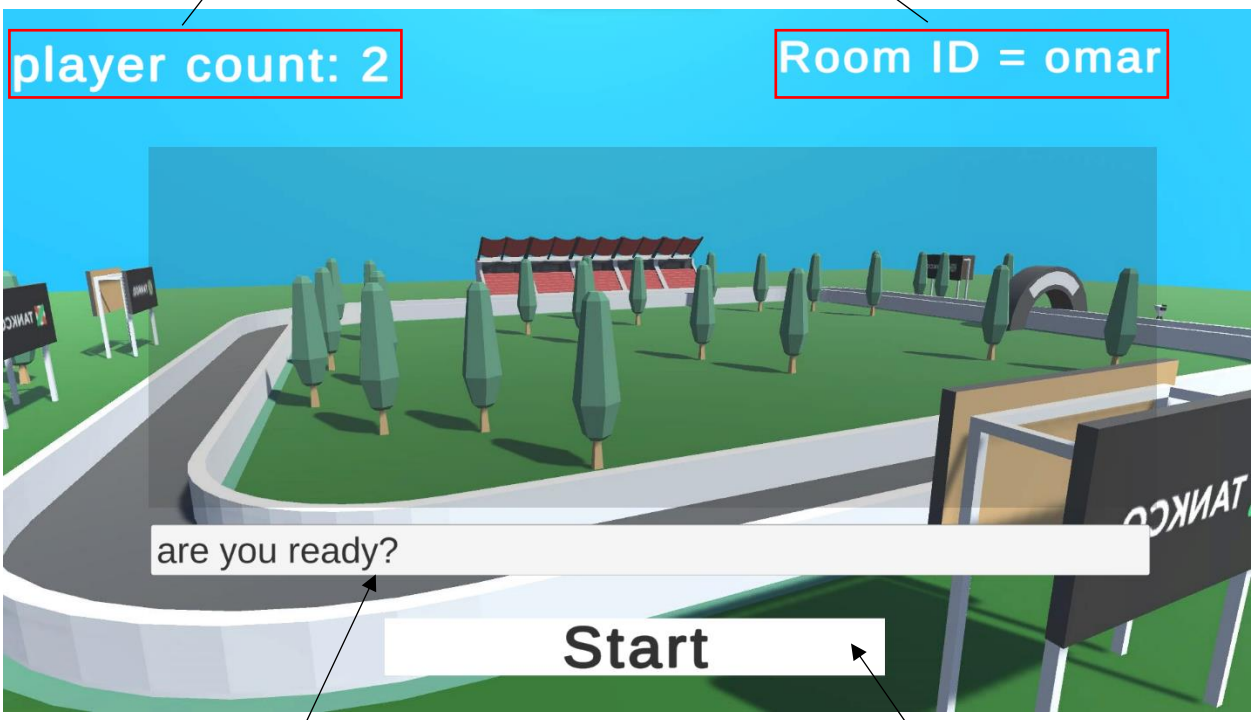


Figure 13 Admin Lobby

Type your Message here!

Start the game from here!

To send a message type your text and press the left CTRL button.

To start the game, click on the start button. You will be directed to the game track.

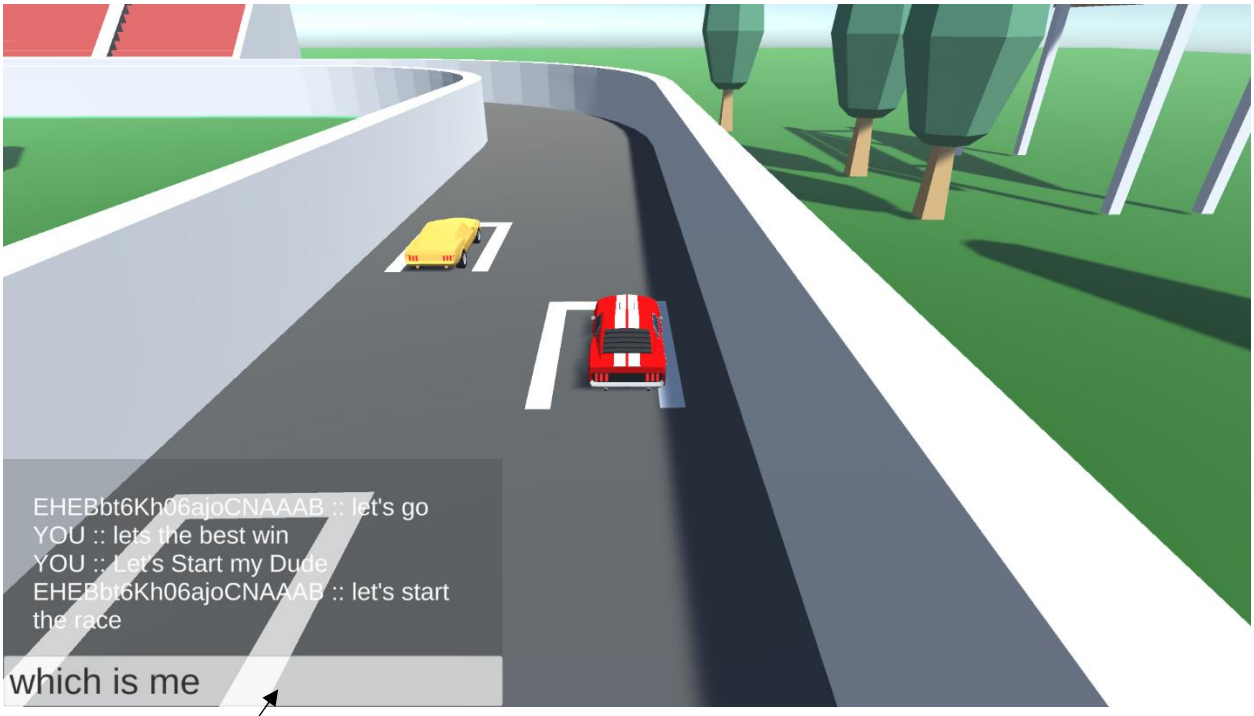


Figure 14 Game Track

Your car will be a red car and the enemy's car will be yellow.

You can also chat while you are in the game by typing the message as shown in the figure and pressing the left CTRL button.

To control your car, use the W, A, S, and D keys for movement.

Figure 15 Car Movement

If you have entered an already existing room ID and clicked on join or spectate button, you will be directed to the guest's lobby. Where you can only read and write messages, but you cannot start the game.

Figure 16 Guest's Lobby

If you are a spectator, you can also participate in the chat, but you cannot control any car. You will see the track from a bird's-eye view.

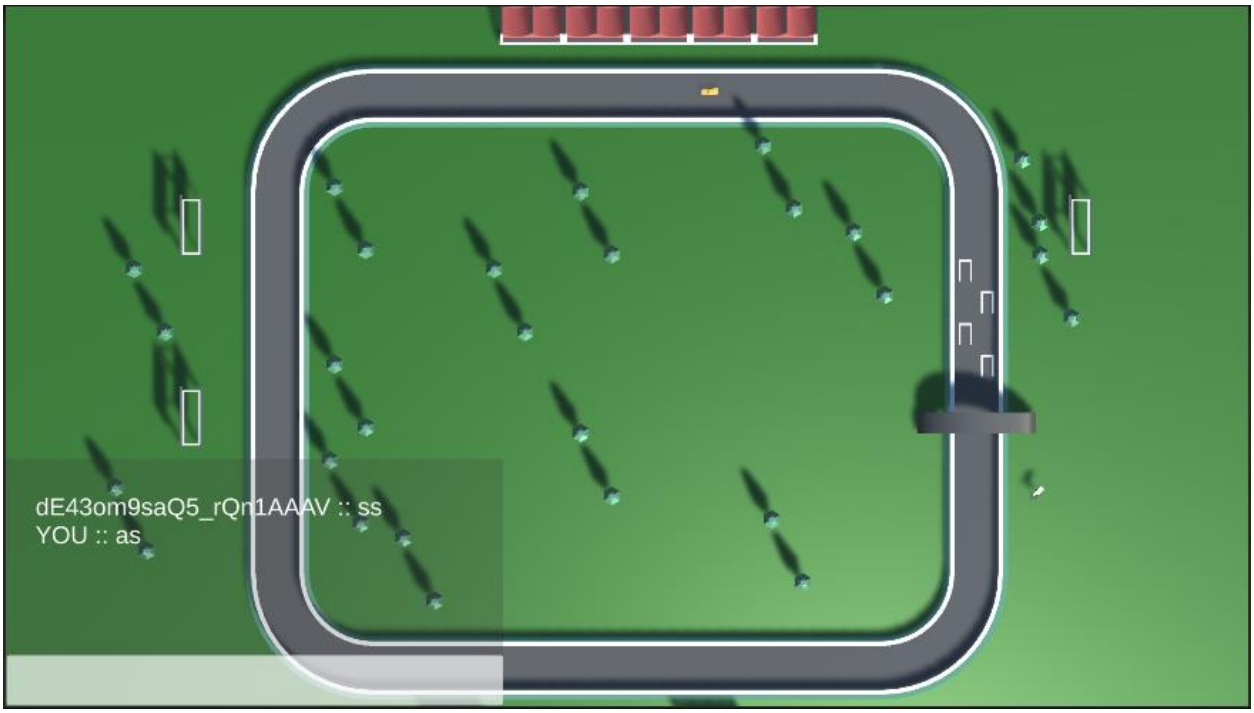


Figure 17 Spectator View

Remember to enjoy the multiplayer distributed racing car game and have fun competing with other players!

9.0 CONCLUSION

In conclusion, this project has been an exciting and challenging endeavor. The development of a distributed multiplayer racing car game, utilizing a three-tier layered architecture, has presented various complexities and required careful consideration of different technologies and components.

Throughout the project, the team has successfully implemented a server developed in Node.js, a Python proxy server, and connected them using Socket.IO for seamless communication. The front-end game was built using the Unity framework, with connectivity to the Python proxy server achieved through .NET C# TCP sockets. This layered architecture has provided a scalable and robust foundation for the game, enabling efficient communication and data exchange between different components.

The project's challenges encompassed areas such as synchronization of game state, real-time multiplayer interactions, seamless scene transitions, and implementing chat functionality for players and spectators. These challenges demanded meticulous planning, coordination, and the integration of various technologies, while maintaining a focus on delivering an engaging and enjoyable gaming experience.

Despite the challenges faced, the project has been an exciting journey that has pushed the team's skills and creativity. The successful implementation of the desired features, including the ability for players and spectators to interact via chat both in the lobby and during the game, has significantly enhanced the multiplayer experience. The exclusion of car instantiation for spectators has further improved the gameplay by providing a clear distinction between players and spectators, ensuring an immersive and realistic environment.

10.0 REFERENCES

[1] Socket.IO. "Socket.IO - Real-time bidirectional event-based communication." Available at: https://socket.io/

[2] Express.js. "Express.js - Fast, unopinionated, minimalist web framework for Node.js." Available at: https://expressjs.com/

[3] Microsoft. "Microsoft - Learn C# | Microsoft Docs." Available at: https://learn.microsoft.com/en-us/dotnet/csharp/

[4] Python SocketIO. "Python SocketIO - Documentation for the Python Socket.IO library." Available at: https://python-socketio.readthedocs.io/en/latest/

[5] Unity. "Unity - Official Documentation for Unity Game Engine." Available at: https://docs.unity.com/

[6] Mozilla Developer Network. "MDN - JavaScript Documentation and Guides." Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[7] M. van Steen and A.S. Tanenbaum, Distributed Systems, 3rd ed., distributed-systems.net, 2017.