



Racing Car Game Project

Malak Mohamed Metwalli 19p7787

Mario Remon Guirguis 19P6892

Mai Hossam Hassan Serageldin 19P6129

Supervisor: Dr Ayman Bahaa,

Eng Mostafa Ashraf

Department of Computer Systems Engineering

Faculty of Engineering at Ain Shams University

Spring 2023

Table of Contents

Table of Figures	- 2 -
Introduction	- 3 -
Detailed project description	- 4 -
Overall design of software	- 5 -
Detailed project Analysis	- 7 -
Beneficiaries of our project	- 11 -
Benefits of using a client/server model	- 11 -
Benefits of using Mongo Database	- 12 -
Task Breakdown Structure	- 13 -
Time plan	- 14 -
System Architecture	- 15 -
Model	- 15 -
View	- 16 -
Controller	- 16 -
Testing scenarios and results	- 17 -
Single player scenarios:	- 17 -
Database scenarios:	- 25 -
Multiple players scenario:	- 28 -
User guide	- 32 -
How to play:	- 32 -
Conclusion	- 34 -

Table of Figures

Figure 1- Main page	- 4 -
Figure 2- Database Mongo DB	- 6 -
Figure 3-server.py file	- 7 -
Figure 4-Car Server.py file	- 8 -
Figure 5-Chatting server.py file.	- 9 -
Figure 6- MVC model	- 15 -
Figure 7- Model architecture	- 15 -
Figure 8- view architecture	- 16 -
Figure 9-Controller architecture	- 16 -

Links

Github:

<https://github.com/MarioRemon/Distributed-Computing-Final-Project.git>

Video:

https://drive.google.com/drive/folders/1_gEYzNwCmegLNet77O2l9K1qoXqIKV2T?usp=sharing

Introduction

Our project features a multi-player 2D car racing game with a chatting feature. The game support multiple autonomous agents contending for shared resources and performing real-time updates to some form of shared state. Our game setting is created to entertain, challenge, and engage players.

We used a distributed architecture offering a dynamic platform for players to enjoy our game. We implemented the multiple players using client/server model and the system is distributed among different clients and servers.

The game supports real time playing where all players can see each other and can chat and exchange messages during the game.

Detailed project description

The Car Race Game is a simple desktop application made using the python programming language. For this Car Racing Game, we designed and accomplished a car game that is shown in Figure 1 with a projective view.

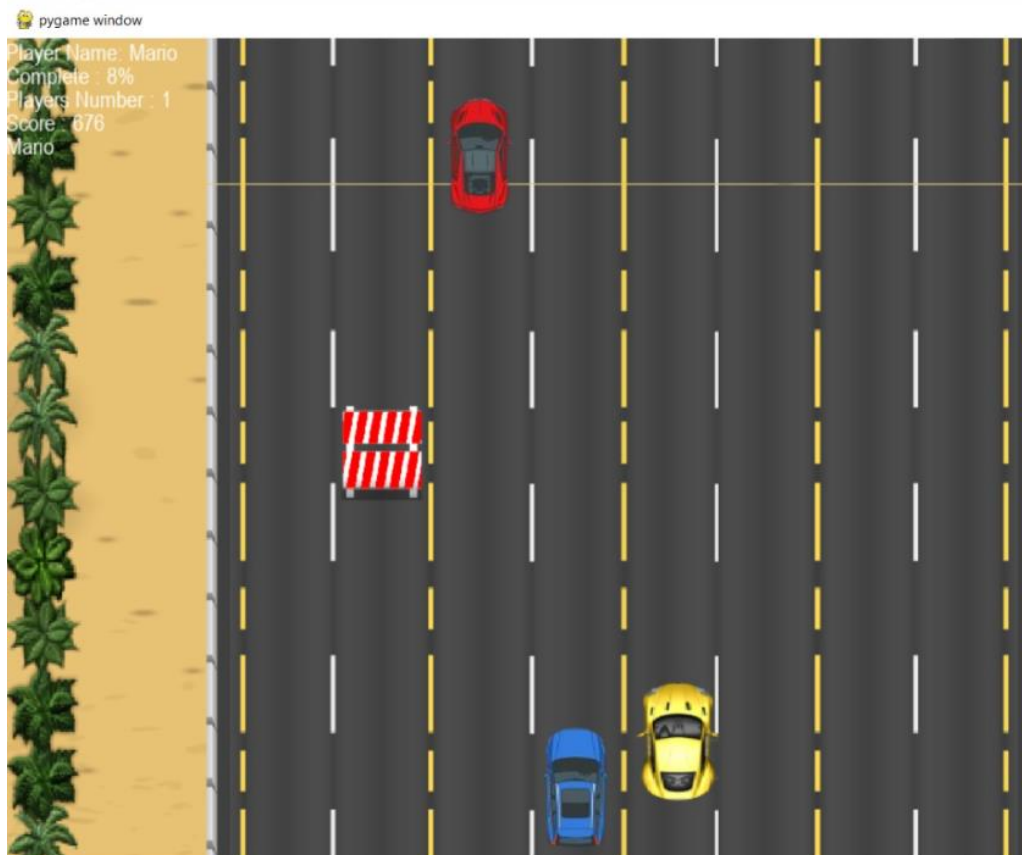


Figure 1- Main page

In this car game, the player control one of the cars to compete with other opponent computer players. The major elements that this game consists of are accelerate, score, obstacles, rank, and crash. The player uses keyboard to control the car to turn left or right, to accelerate while the top 3 players are shown in the top left of screen depending on the score.

The theme of our game is to compete with the other opponents that are controlled by other players on a different computer in a race, the player's goal is to get to the destination as soon as possible while trying to avoid crashing to other cars or road obstacles that make the player lose.

The difficulty level of this game is determined by the number of players, obstacles on the road and reaching the finish line.

Overall design of software

1. Servers

The game consists of the main server (server.py) listening for clients incoming requests. When a client request is accepted, it is sent to one of two servers: carserver.py and carserver1.py to start the game. Every game server can handle up to 8 players. When the first server is full (exceeds the 8-player limit), the clients is sent to the second server carserver1.py. if the second server is also full, it stops accepting clients' request.

2. Game

When a game first begins, all game information is then sent to database and stored with game ID. every client that connects is associated with the player id and is stored in database with all player information (player id, username, score).

Throughout the game, the X and Y coordinate updates while the car moves and is also stored in database.

3. Chatting feature

Another feature we have is the chatting feature, players can send and receive messages during the game. this feature is implemented using the chatting server. when first running the application, we run the chatting server (Chatthingserver.py) that listens for clients and wait for requests. when player firsts connect with the main server it also connects with the chatting server corresponding to which car server the clients are connected to. The chatting server is responsible for the chatting operation during the game. The messages are stored in database to be retrieved in case of failure.

4. Obstacles

Throughout the road, the file obstacles.py generates obstacles randomly for cars to avoid.

5. Crash

In case of crash, when a car hits another car or an obstacle, the player losses and exist the game.

6. Rain mode

The rain mode is an additional feature that we added. when key"" is pressed on the keyboard rain is generated in the background to make the effect of rain on the road.

7. Database

The application has an online database that gets updated regularly. Multiple database replicas are implemented to help the program to recover in case of failure.

```
▼ 1: Object
  id: 4
  userName: "momo"
  position: "(202, 550)"
  mapComplete: "0.7374999999999999"
  active: false
  score: "360"
```

Figure 2- Database Mongo DB

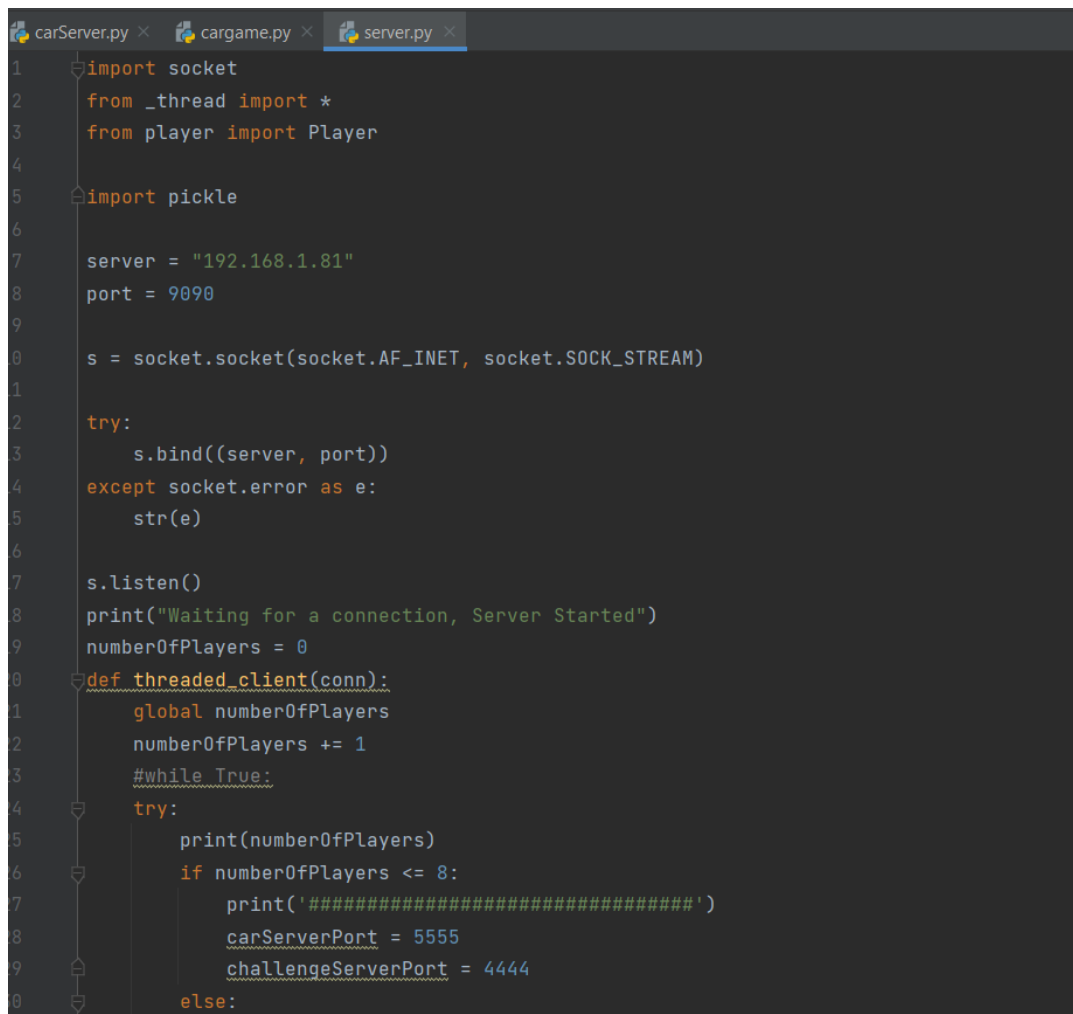
Detailed project Analysis

Our project is implemented using a Client/Server model that divides the tasks between multiples servers and clients. The client-server model is good for real-time games because it allows for centralized control, scalability, reduced cheating, synchronization, and network efficiency. The server acts as a central authority, ensuring fair gameplay and handling game logic. It can scale to support many players, reduces cheating risks, synchronizes game state, and optimizes network traffic. However, challenges like server load management, latency, and increased network traffic need to be managed effectively.

A. Servers

i. Server.py

It is a controller server. it plays a crucial role in managing and coordinating the activities of multiple servers within our distributed system and network the Server class is the main server that receive all client's requests at first. then, depending on the system state, the server replies with the appropriate IP address and Port Number that the client can connect to in order to join the game.

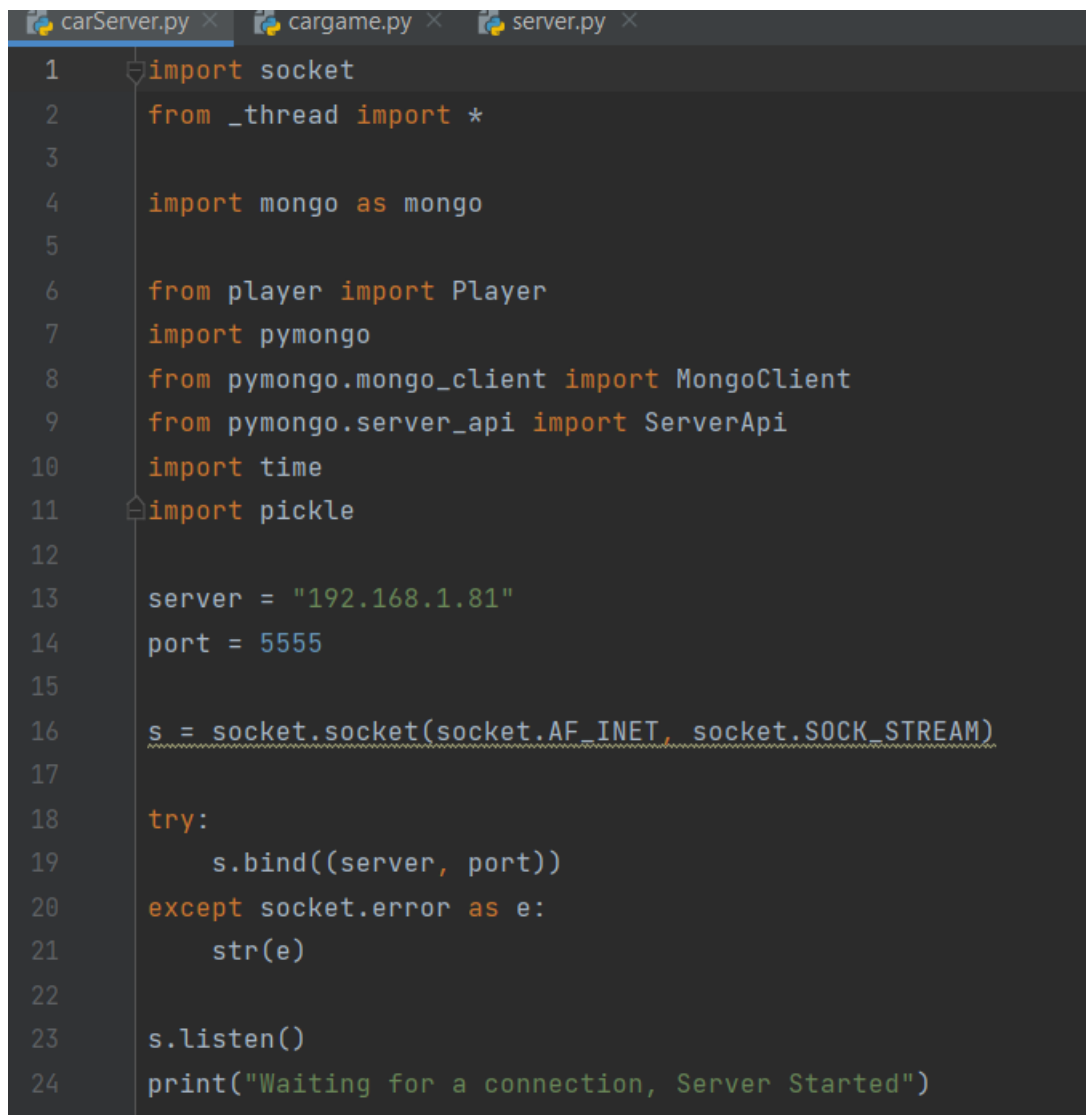


```
1 import socket
2 from _thread import *
3 from player import Player
4
5 import pickle
6
7 server = "192.168.1.81"
8 port = 9090
9
10 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11
12 try:
13     s.bind((server, port))
14 except socket.error as e:
15     str(e)
16
17 s.listen()
18 print("Waiting for a connection, Server Started")
19 numberOfPlayers = 0
20
21 def threaded_client(conn):
22     global numberOfPlayers
23     numberOfPlayers += 1
24     #while True:
25     try:
26         print(numberOfPlayers)
27         if numberOfPlayers <= 8:
28             print('#####')
29             carServerPort = 5555
30             challengeServerPort = 4444
31         else:
```

Figure 3-server.py fil

ii. Car server.py

it is the server responsible for starting the game and streaming it across multiple clients that joined. The game can only support eight players. if the number of players exceeds that number the car server starts a new game for new players. There is two Car server's classes to support two games: carServer.py and carServer1.py. when the first game begins it continues to accept clients' connections and let them join the game. when it reaches the eight players limit, the game is full and the carServer stops accepting request in this case,we run carServer1 to accept new players in a new game.

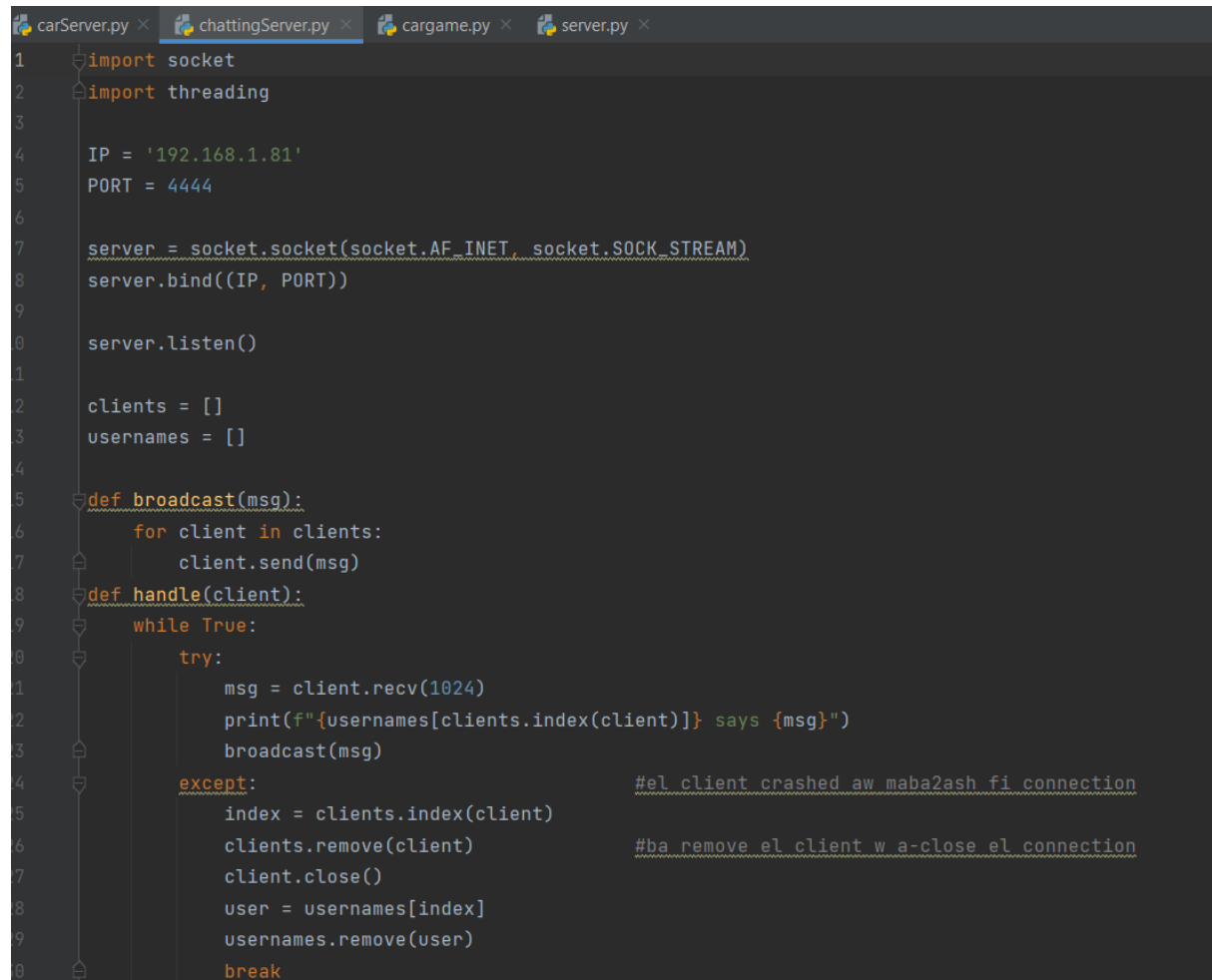


```
1 import socket
2 from _thread import *
3
4 import mongo as mongo
5
6 from player import Player
7 import pymongo
8 from pymongo.mongo_client import MongoClient
9 from pymongo.server_api import ServerApi
10 import time
11 import pickle
12
13 server = "192.168.1.81"
14 port = 5555
15
16 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17
18 try:
19     s.bind((server, port))
20 except socket.error as e:
21     str(e)
22
23 s.listen()
24 print("Waiting for a connection, Server Started")
```

Figure 4-Car Server.py file

iii. Chatting Server.py

The Chatting Server is responsible for the chatting feature in the game. it passes messages between players during the game that is placed at the right of the screen. The chatting has its own network for messages to flow over it.



```
1 import socket
2 import threading
3
4 IP = '192.168.1.81'
5 PORT = 4444
6
7 server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8 server.bind((IP, PORT))
9
10 server.listen()
11
12 clients = []
13 usernames = []
14
15 def broadcast(msg):
16     for client in clients:
17         client.send(msg)
18
19 def handle(client):
20     while True:
21         try:
22             msg = client.recv(1024)
23             print(f'{usernames[clients.index(client)]} says {msg}')
24             broadcast(msg)
25         except:
26             index = clients.index(client)
27             clients.remove(client)
28             client.close()
29             user = usernames[index]
30             usernames.remove(user)
31             break
```

Figure 5-Chatting server.py file.

B. Game Network class

This class manages the communication and networking between all different components of the system. It is responsible for the sending and receiving of messages through the network between the Car game server and the clients. It is responsible for setting up a network connection, establishing communication with a server, and initializing components for a game and chat functionality.

C. Main.py

The main class initiates the “HomePage” class which is a key component of our game. When “HomePage” is called it begins the execution of the application and provide a starting point for the Graphical User Interface which is the home page of the game.

D. Button.py

This file is used when homepage is running. it is called for all buttons created.

E. Car game.py

This file opens the game itself and start running it. it calls all GUI features, show the cars and lanes and start the Game loop so that players can start playing.

F. Chat.py and Chatty.py

Those two files are responsible for chatting feature. When servers accept clients' connections, all clients connected to the server can exchange messages. The messages are shown in the right side of the screen. players can enter any text in the text field and then press send or enter to send the text. It is then showed above with the players username beside the text.

G. Network.py

The network file is one of the essential files in the project. The network class is an like an intermediate layer between the car game and the servers. All communications between them is done through the Network.py file. when car game wants any information from server, he sends an request to network. Network.py communicate with the server to get information needed and then send it back to car game.

H. Obstacles.py

This file is responsible for obstacles generated. Its setup all obstacles features and objects that are shown during the game.

I. Player.py

The player class is responsible for initializing players objects .it takes multiples parameters: players id, username, score, rank, and other player information. It tracks the buttons pressed by player and updates its X and Y-positions and update the game map.

J. Homepage.py

The homepage file is responsible for the game homepage including all buttons, text fields, images and events that are shown to the user.

K. Mute Button.py

The mute button class is responsible for the mute features during the game. When mute is pressed, the background music is muted, and the mute button image is shown. When it is unmuted, the image of unmuted button is showed, and background music is back on.

Beneficiaries of our project

This project combines the thrill of car racing with the power of networked gameplay, data storage, and efficient programming. The primary beneficiaries of the car racing game are the players themselves. They experience the excitement and adrenaline rush of racing against friends or online opponents. The multiplayer capability facilitated by the client-server model allows players to compete in real-time, enhancing the overall gaming experience. The game provides entertainment, competition, and a platform for social interaction among players.

The car racing game project contributes to the gaming community and industry as a whole. It adds to the diverse range of gaming experiences available, catering to car racing enthusiasts and attracting new players. The project showcases the capabilities of Python as a game development language and highlights the potential of the client-server model for multiplayer gaming. It inspires creativity and innovation within the gaming industry, encouraging the development of new and exciting game projects.

This project exemplifies the positive impact that technology and gaming can have on various individuals and industries, creating an engaging and immersive gaming experience for all involved.

The multiplayer capabilities facilitated by the client-server model enable players to connect and race with friends or players from around the world. This fosters social interaction, allowing players to form communities, join teams, and engage in friendly competition. The game becomes a platform for players to connect, communicate, and build relationships with fellow racing enthusiasts.

Benefits of using a client/server model

By implementing a client-server model, players can engage in real-time multiplayer gameplay. The client-server architecture enables synchronized gameplay, ensuring that all players see the same game state and can interact with each other simultaneously.

The client-server model allows for scalability, accommodating a large number of players simultaneously. The server can handle multiple client connections and manage the gameplay for all participants. This scalability ensures that the game can accommodate a growing player base without sacrificing performance or user experience.

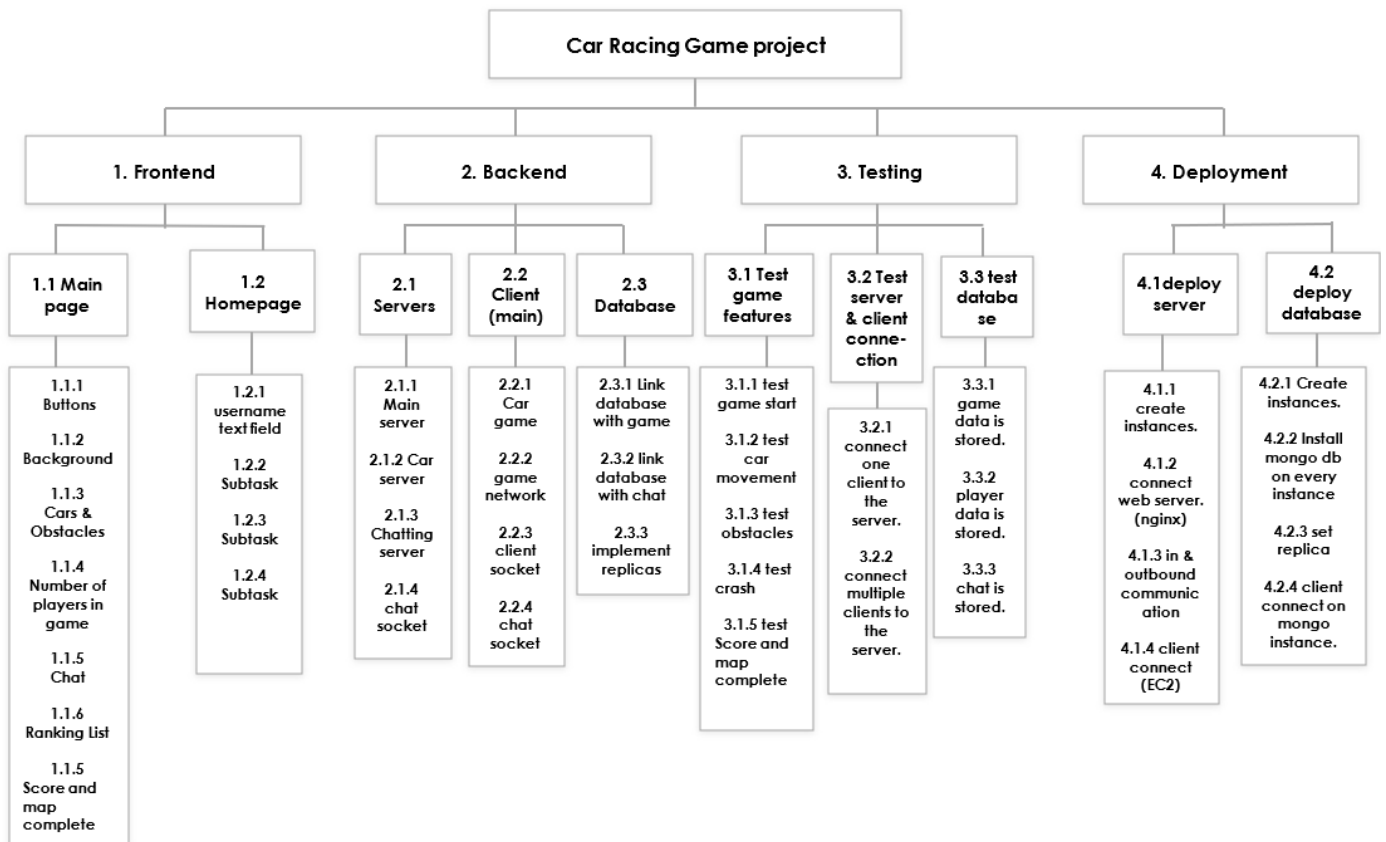
With the client-server model, the game logic resides on the server. This centralization ensures that the gameplay is consistent and secure across all clients. The server validates player actions, handles collision detection, and enforces game rules.

Benefits of using Mongo Database

using MongoDB as a database offers several benefits:

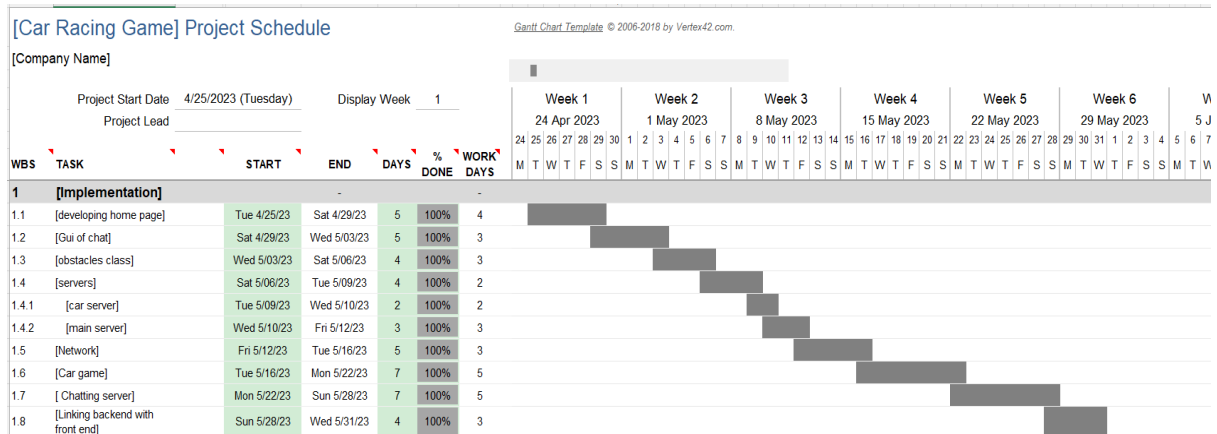
1. **Flexible and Scalable Data Model:** MongoDB's flexible document model allows for easy schema evolution and horizontal scalability as your data grows.
2. **High Performance:** MongoDB offers high-performance read and write operations, indexing, and query optimization techniques.
3. **Replication and High Availability:** MongoDB supports replica sets for automatic failover and continuous availability of data.
4. **Flexible Data Consistency:** MongoDB provides strong consistency within a single document and offers flexible consistency options for distributed scenarios.
5. **Horizontal Scalability:** MongoDB's sharding feature enables horizontal scaling by distributing data across multiple servers.

Task Breakdown Structure



Time plan

Project Grant Chart



Roles:

1. Frontend

1.1 Main Page - Mai Malak Mario

1.2 Homepage - Mai

2. Backend

2.1 Servers - Mario

2.2 Client – Mario Malak

2.3 Database – Mario Mai

3. Testing

3.1 Test game features - Mai

3.2 Test server and client connection - Malak

3.3 test database – Mario

4. Deployment - Mai , Mario and Malak

System Architecture

The Architecture used in our game is the MVC architecture. the MVC separates the game's data and logic (Model), visual presentation (View), and user interaction (Controller), resulting in a modular and maintainable codebase. it allows for code reusability, enabling components to be reused across different tracks, cars, or game modes.

The MVC architecture helps in separating concerns and maintaining a clear separation between data, presentation, and user interaction. It enhances modularity, reusability, and ease of maintenance in the development of the car racing game.

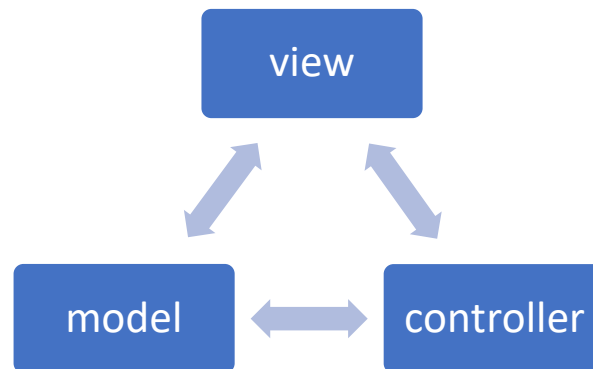


Figure 6- MVC model

Model

The Model represents the underlying data and logic of the game. It includes components such as the game state, players data, the servers, and any other game-specific data.

The Model is responsible for managing the communication between servers and clients and updates to the game state.

The server is first called and start listening for incoming clients' requests then start sending each client to its corresponding server: Carserver.py and Car server1.py.

The client starts a connection with either Carserver.py or Car server1.py and chattingserver.py or chattingserver1 until client is disconnected from the game. The communication with the database is done through the Car and Chatting server. At the beginning of the game, each player data is inserted into the database table and will keep updating throughout the game. The chat between players is also saved in database.

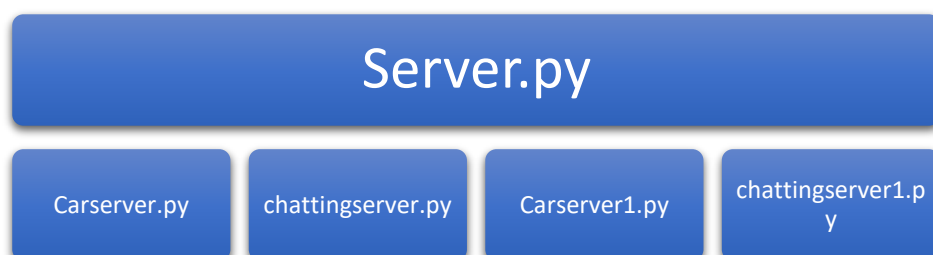


Figure 7- Model architecture

View

the "View" component is responsible for presenting data to the user and handling user interactions. It represents the graphical user interface (GUI) of an application. the "View" component is responsible for presenting data to the user and handling user interactions. It represents the graphical user interface (GUI) of an application.

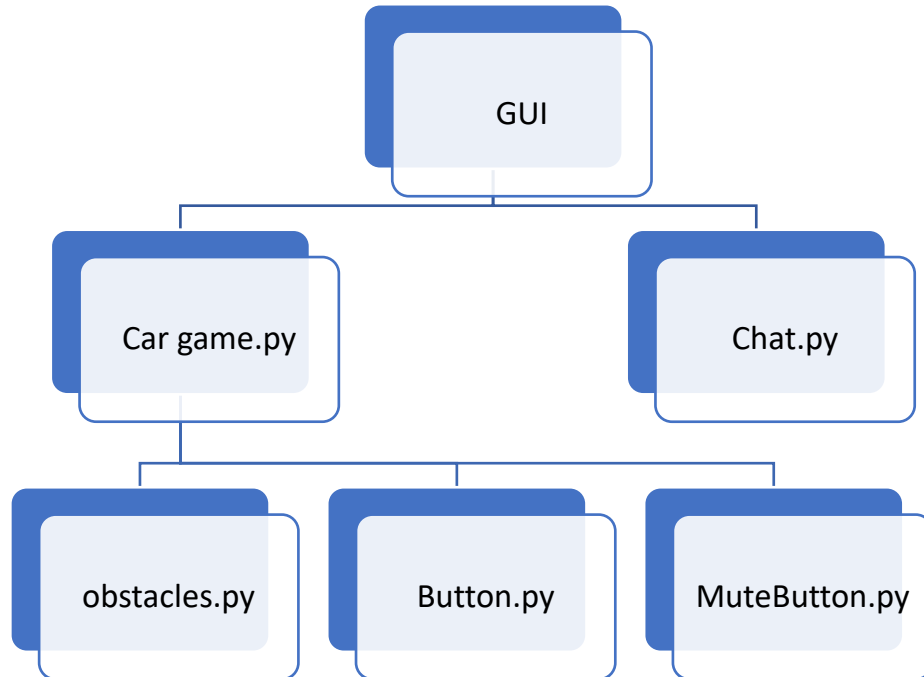


Figure 8- view architecture

Controller

The Controller acts as an intermediary between the Model and the View. It receives input from the player (such as keystrokes or touch events) and translates it into commands for the Model or View.

In the car game, the Controller would handle player input for car controls, such as acceleration, moving between lanes and even pausing the game.

The Controller updates the Model based on the player's input and triggers the necessary actions or changes in the game state.

The Controller also updates the View to reflect any changes in the game state, ensuring that the player sees the updated visuals and feedback.



Figure 9-Controller architecture

Testing scenarios and results

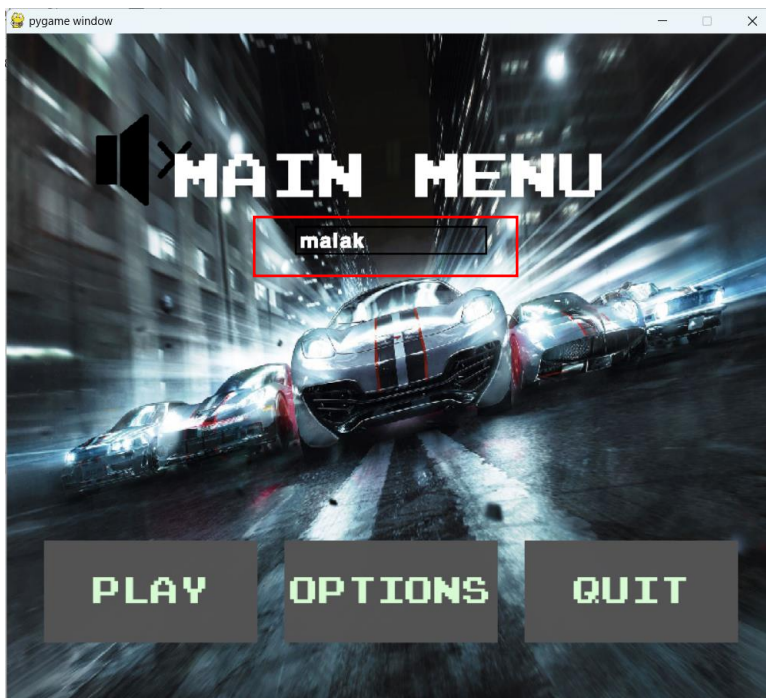
Single player scenarios:

Scenario 1: open application.



Test result: pass

Scenario 2: open application >> enter username.



Test result: pass

Scenario 3: open application >> enter username>>press play.

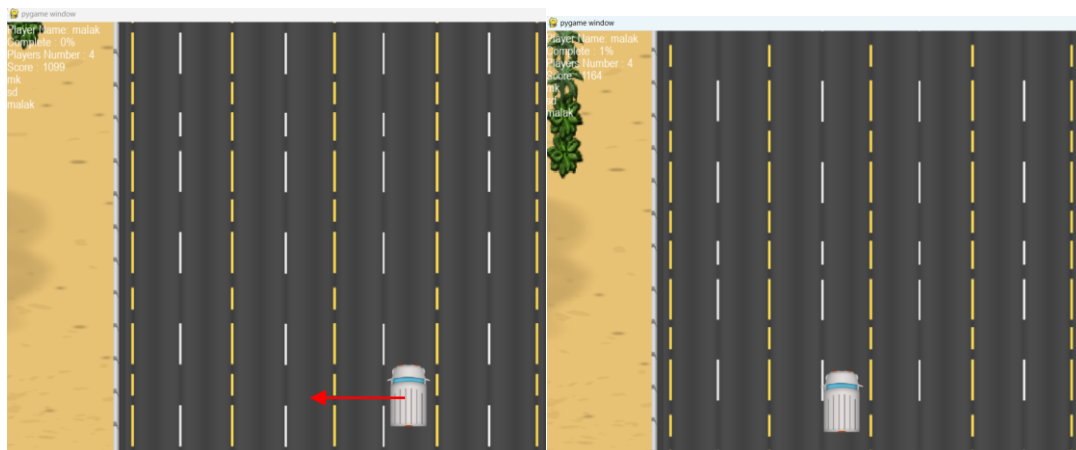


Test result: **pass**

Scenario 4: open application >> press play>> music background is on.

Test result: **pass**

Scenario 5: open application >> enter username>>press play>> Try arrows.



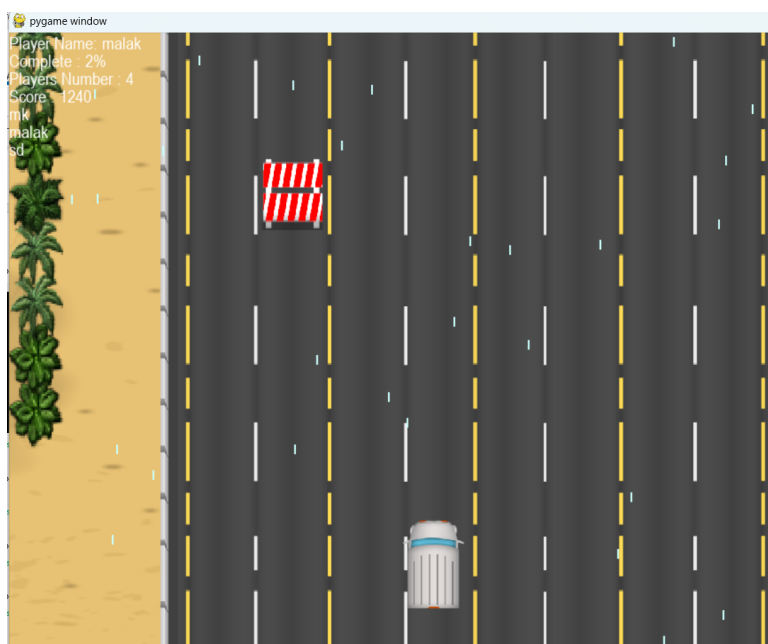
Test result: **pass**

Scenario 6: open application >> enter username>>press play>> press pause.



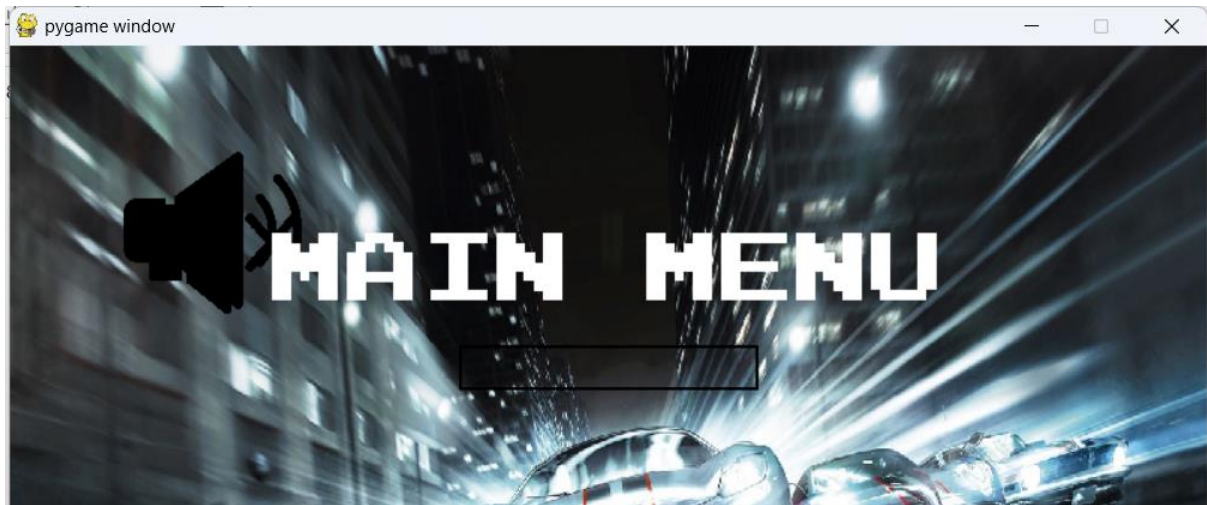
Test result: **pass**

Scenario 7: open application >> enter username>>press play>> press "R" for rain mode.



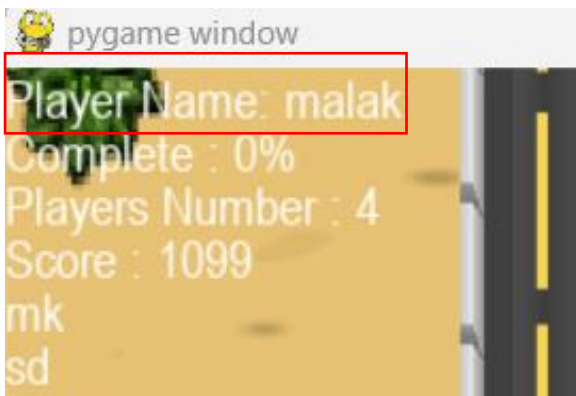
Test result: **pass**

Scenario 8: open application >> enter username>>press play>> press mute.



Test result: **failed**.

Scenario 9: open application >> enter username>>press play>> check username.



Test result: **pass**

Scenario 10: open application >> enter username>>press play>> check map complete =0.



Test result: **pass**

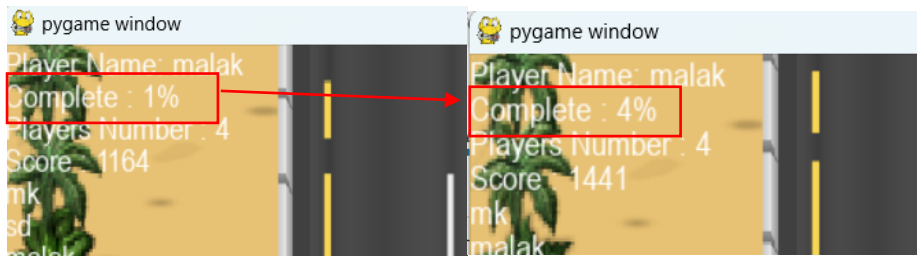
Scenario 11: open application >> enter username>>press play>> check score =0.



When player first joins, score begins counting immediately.

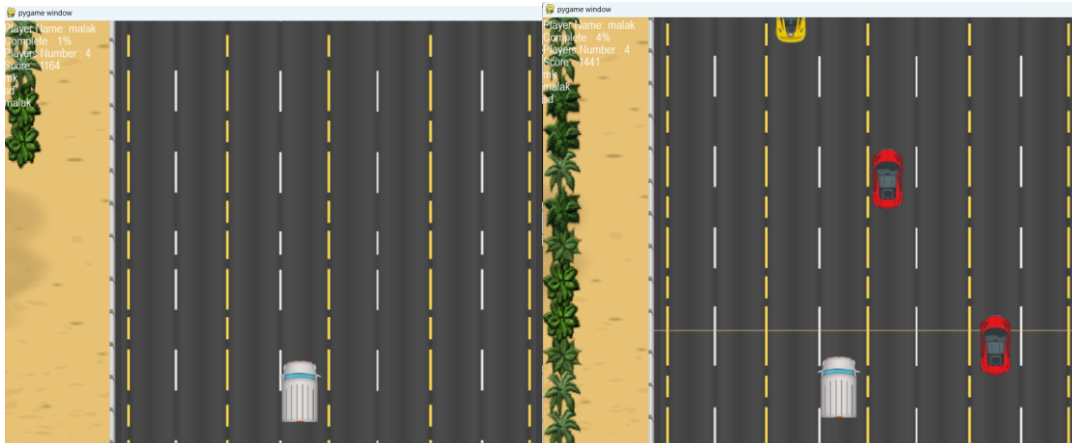
Test result: **pass**

Scenario 12: open application >> enter username>>press play>> check map complete percentage increases while playing.



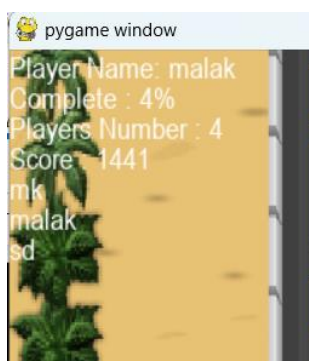
Test result: **pass**

Scenario 13: open application >> enter username>>press play>> check score increases while playing.



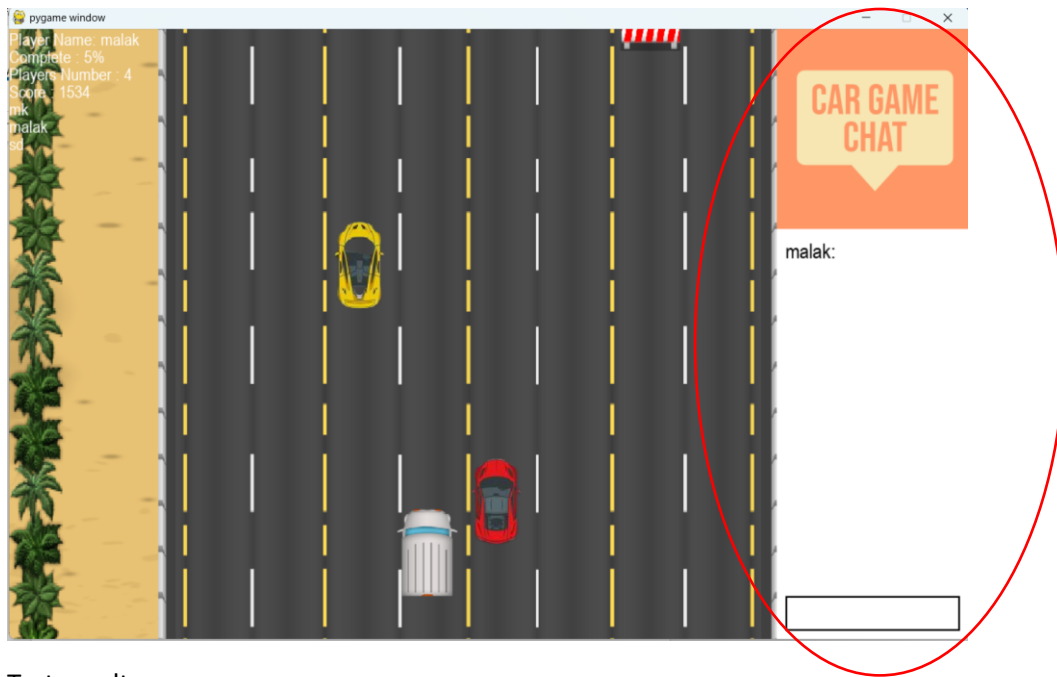
Test result: **pass**

Scenario 14: open application >> enter username>>press play>> check rankers list exists.



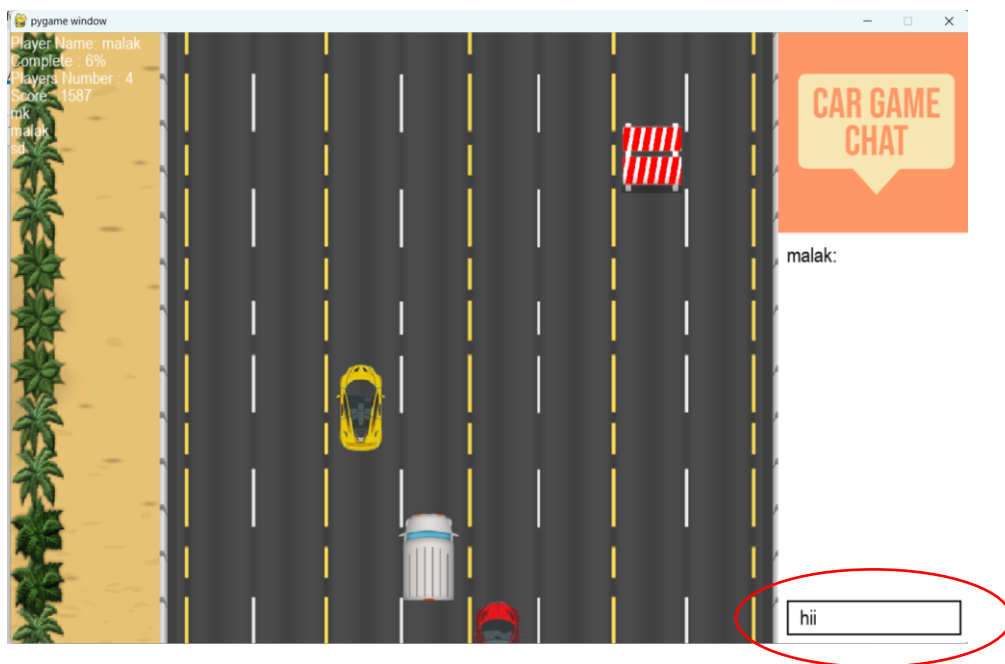
Test result: **pass**

Scenario 15: open application >> enter username>>press play>> check chat window appears.



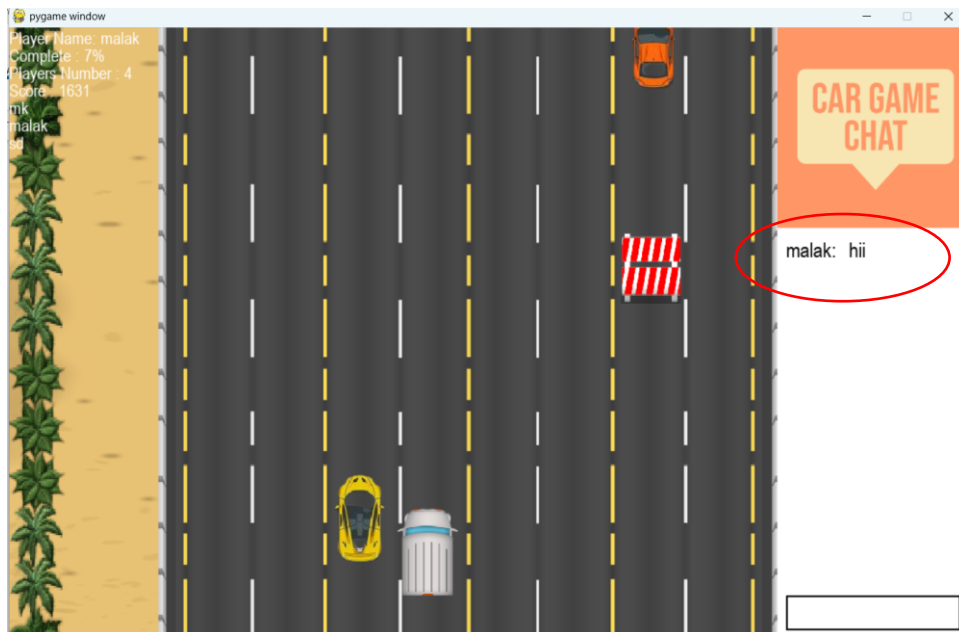
Test result: **pass**

Scenario 16: open application >> enter username>>press play>> try typing in text filed and send.



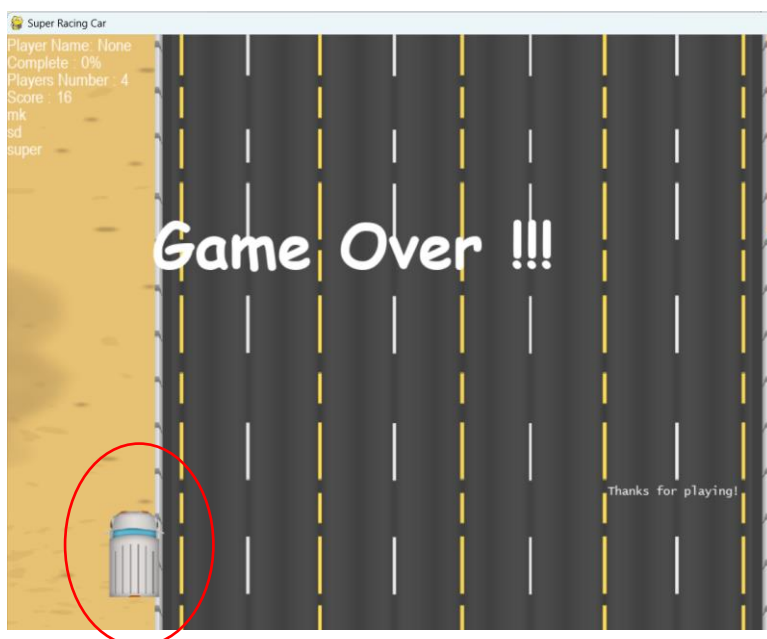
Test result: **pass**

Scenario 17: open application >> enter username>>press play>> try typing in text filed and send>>check text appears above.



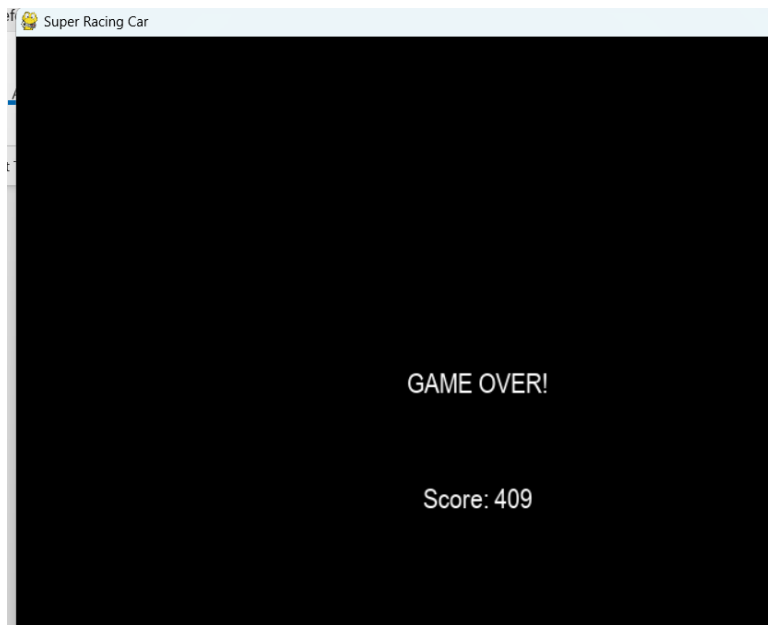
Test result: **pass**

Scenario 18: open application >> enter username>>press play>> try moving outside of lanes>>game over.



Test result: **pass**

Scenario 19: open application >> enter username>>press play>> player crash>> game is over.



Test result: **pass**

Scenario 20: open application >> enter username>>press play>> player crash>>crash sound is on>> game is over.

Test result: **pass**

Database scenarios:

Scenario 1: test database connection.

```
Pinged your deployment. You successfully connected to MongoDB!
```

Test result: **pass**

Scenario 2: test if there exist a document in database>>document exists>> retrieve last document according to time.

Test result: **pass**

Scenario 3: test if there exist a document in database>>document does not exist>> create new document.

Test result: **pass**

Scenario 4: check players number in game>> equal 8 >>create new game and adds player.

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('648c6a15fad024ae4a4af9aa')
GameId: 1
server: 1
time: "Fri Jun 16 16:56:37 2023"
numberOfPlayers: 8
chatOfTheGame: "A user connected to the server!A user connected to the server!malak:ho..."
▶ players: Array
```

```
_id: ObjectId('648f5f35916d2d294091c779')
GameId: 1
server: 1
time: "Fri Jun 16 16:56:37 2023"
numberOfPlayers: 1
chatOfTheGame: ""
▼ players: Array
  ▶ 0: Object
```

Test result: **pass**

Scenario 5: check players number in game>> below 8>> add player in last document.

```
_id: ObjectId('648c6a15fad024ae4a4af9aa')
GameId: 1
server: 1
time: "Fri Jun 16 16:56:37 2023"
numberOfPlayers: 2
chatOfTheGame: "A user connected to the server!A user connected to the server!malak:ho..."
▼ players: Array
  ▶ 0: Object
  ▶ 1: Object
```

```
▶ _id: ObjectId('648c6a15fad024ae4a4af9aa')
GameId: 1
server: 1
time: "Fri Jun 16 16:56:37 2023"
numberOfPlayers: 1
chatOfTheGame: "A user connected to the server!A user connected to the server!malak:ho..."
▶ players: Array
```

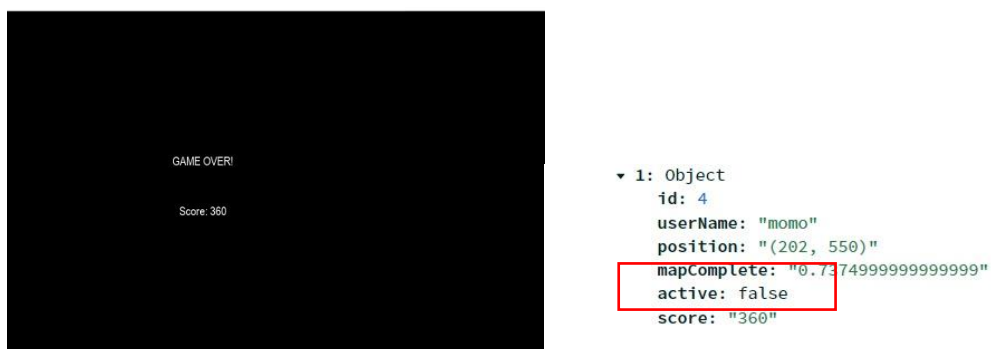
Test result: **pass**

Scenario 6: open application >> enter username>>press play>> pause game>> check game data in database if it is updated.



Test result: **pass**

Scenario 7: open application >> enter username>>press play>> connection is lost>> check game data in database if it is updated.



Test result: **pass**

Scenario 8: check chat is stored in database.

```
QUERY RESULTS: 1-1 OF 1

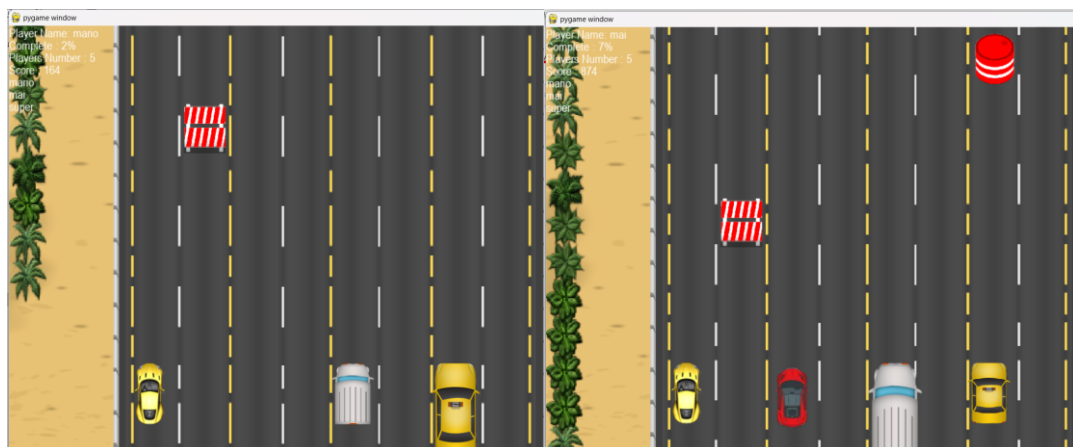
_id: ObjectId('648c6a15fad024ae4a4af9aa')
GameId: 1
server: 1
time: "Fri Jun 16 16:56:37 2023"
numberOfPlayers: 8
chatOfTheGame: "A user connected to the server!A user connected to the server!malak:ho..."
▶ players: Array
```

Test result: **pass**

Multiple players scenario:

In multiplayer mode: open application >> enter username>>press play>> while playing another player joins the game.

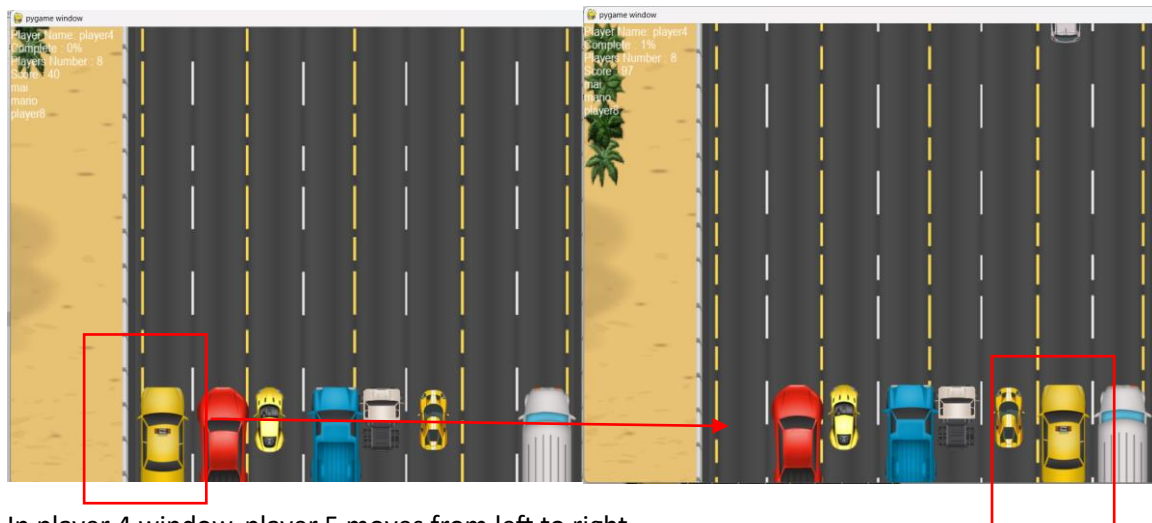
Scenario 1: in every player window, other players cars are seen.



The two cars appear in both players' screen.

Test result: **pass**

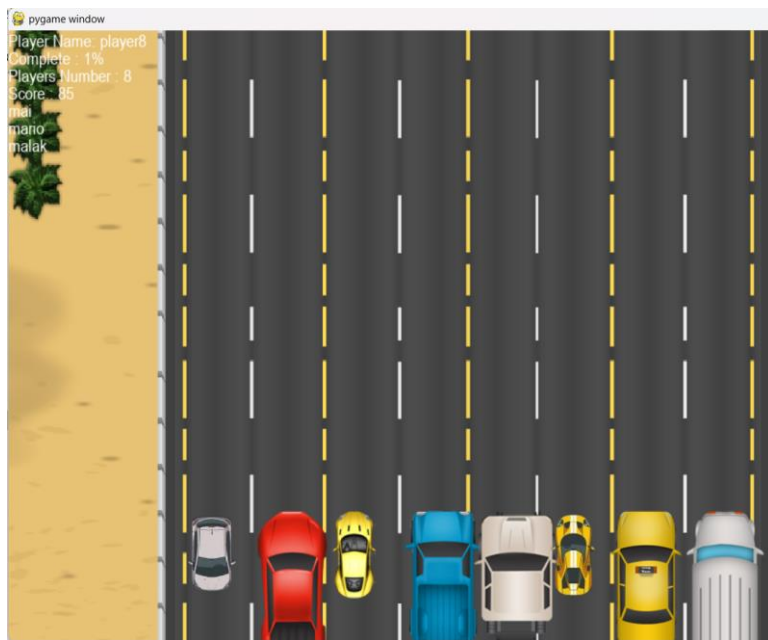
Scenario 2: when one player moves, it appears to others.



In player 4 window, player 5 moves from left to right.

Test result: **pass**

Scenario 3: Cars appears till the maximum limit of 8 players.



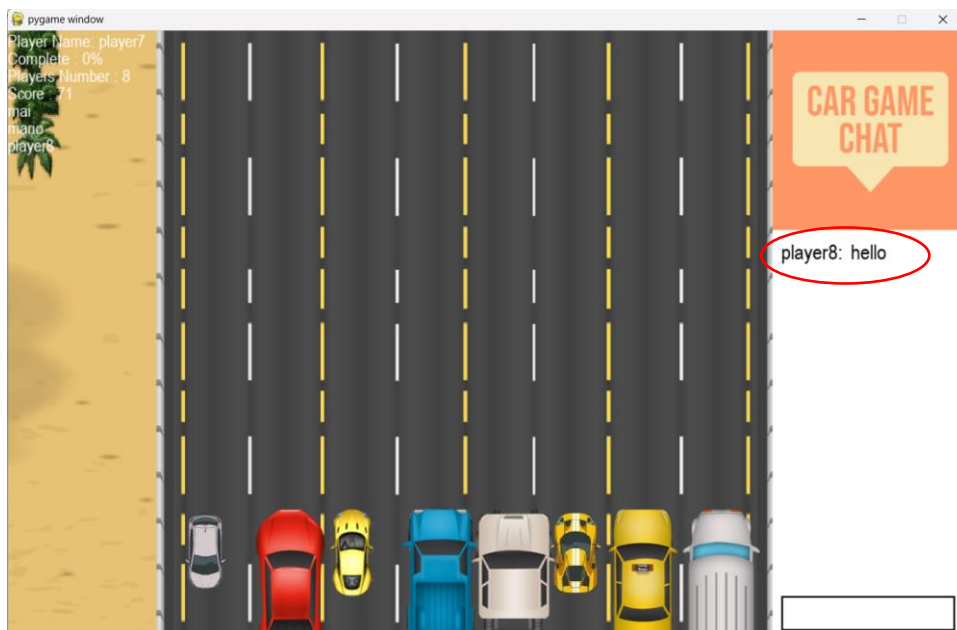
Test result: **pass**

Scenario 4: check players ranking.



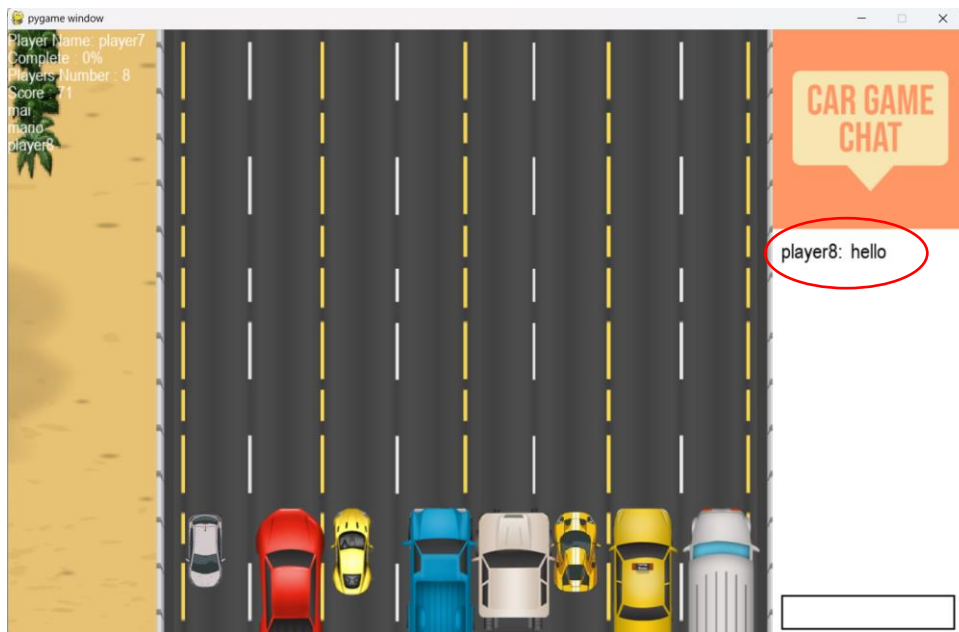
Test result: **pass**

Scenario 5: check when player send a chat, it appears to others.



Test result: **pass**

Scenario 6: check that the right username appears next to the player text.

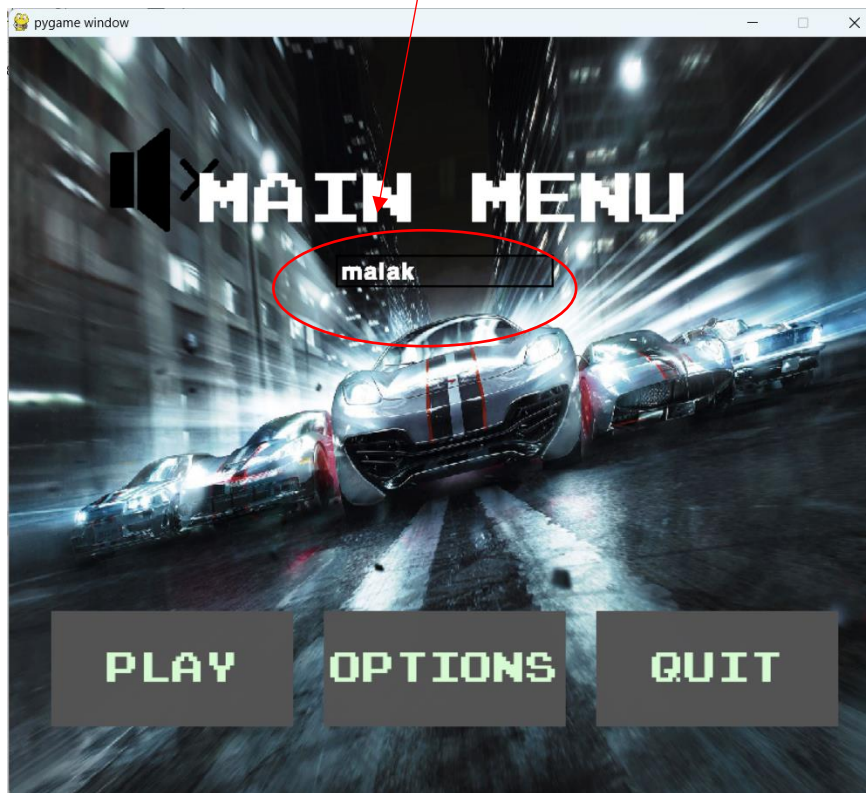


Test result: **pass**

End-User guide

How to play:

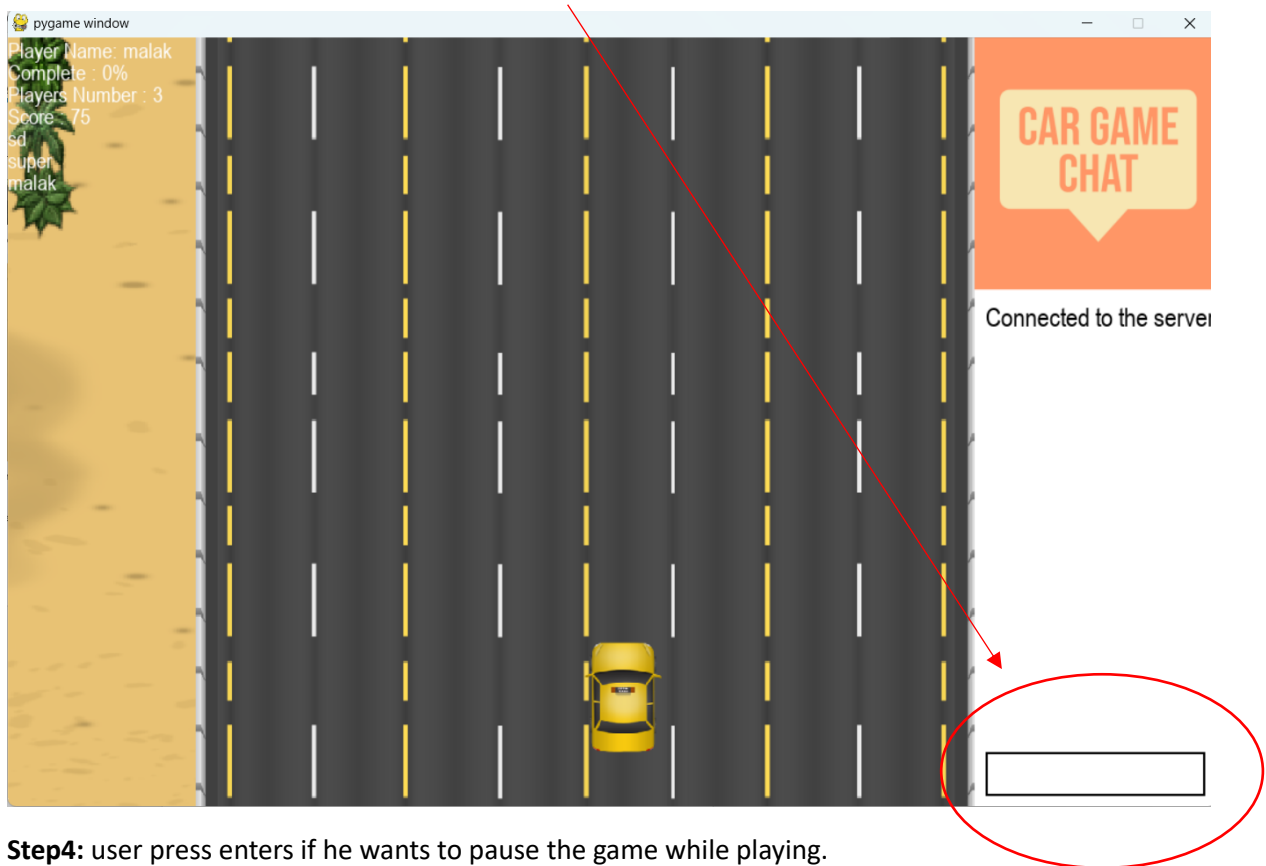
Step1: user enters his username in the text field and press enter and quit when he wants to exit.



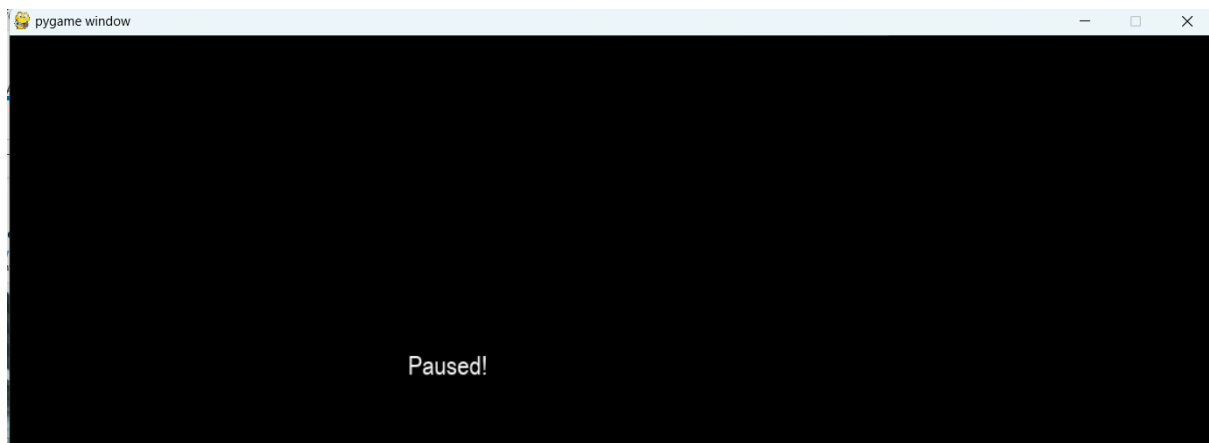
Step2: game begins.



Step3:user can chat with other players from here.



Step4: user press enters if he wants to pause the game while playing.



Conclusion

In our project, a 2D vehicle racing game with multiplayer chat is featured. The game allows for numerous autonomous agents to compete for the same resources and make real-time adjustments to a common state. Our game's environment is designed to amuse, test, and engage players.

In order to provide players with a dynamic platform to experience our game, we chose a distributed architecture. The system is dispersed across several clients and servers, and the client/server approach was used to implement the numerous participants.

The game allows for real-time play, where all players can see one another and communicate with one another via chat and message exchanges.

The excitement of auto racing is combined in this project with the strength of networked gaming, data storage, and effective programming. The players themselves are the main winners in the automobile racing game.

When they race other people in person or online, they feel the thrill and rush of competition. The client-server model's multiplayer functionality makes it possible for players to engage in real-time competition, which improves the overall game experience. The game offers participants amusement, rivalry, and a venue for social interaction.

This project serves as an example of the beneficial effects that technology and gaming can have on many people and businesses while also providing an immersive and fun gaming environment for all parties.

Real-time multiplayer games are possible by using a client-server structure. Synchronised gaming is made possible by client-server architecture, allowing all players to interact with one another and see the same game state at once.

Scalability is made possible via the client-server approach, which can support several participants at once. The server may oversee the games for everyone involved and handle many client connections. Because of its scalability, the game may grow to accommodate more players without compromising on its functionality or user interface.

The game logic is stored on the server when using the client-server approach. Through centralization, all clients will experience consistent, secure gameplay. The server oversees collision detection, verifies player activities, and upholds game regulations.

References

- 1- *Download MongoDB Community Server*. MongoDB. (n.d.).
<https://www.mongodb.com/try/download/community>
- 2- *Install MongoDB - MongoDB Manual*. (n.d.-b).
<https://www.mongodb.com/docs/manual/installation/>
- 3- Hunt, M. (1994). *Free*. Amazon. <https://aws.amazon.com/free/>
- 4- *Nginx documentation*. nginx. (n.d.). <https://nginx.org/en/docs/>