Computer Science NEA 2022 Armin Raad

# Current method

The current methods of getting faster at typing include typing tutors or playing typing racing games against other players. The way this works is the player goes on a website and logs in using a username and password (optional) and then finds a match to play against other players. They are then shown a long sentence which can range from song lyrics to randomly generated phrases which they need to then type out as fast as possible without making any errors. Making an error will result in not being able to progress and in different websites it is treated differently whether you have to manually delete the incorrect letters or to just type the correct letter.

The player who finishes the phrase first will win the race and there are usually more than 2 players total.

# The problem

The problem with the current method is that in the case of typing tutors it is boring and in the case of the racing games it is against very slow typers due to the lack of matchmaking. Oscar would like a ranked mode in which you are put with players of similar skill level to make a competitive environment. This makes a lot of sense as it encourages improvement and prevents players from saying “I’m good enough now” when they start winning most of their matches against players far slower than them. By competing with other fast typers, it makes the player want to improve, either to increase their rank or to simply be better than more opponents.

This is a solution seen in many competitive games today, players will get to a level where casual matches are far too easy as their skill level is so much higher than the average player’s. An example would be a first person shooter, where over time a player’s aim gets so good that they can make mistakes such as bad positioning without getting punished for it simply because they kill the other player faster. This then means when they compete against other good players who have the same aim as them, they will lose. In cases like these playing against players worse than you could make you worse at the game and in the context of typing you would get away with making more mistakes or just not typing as fast but still winning. This would mean you would stay at around the same skill level with no improvement.

By providing a competitive environment, improving is not only encouraged, it is often necessary to keep climbing the ranks (ranks are divisions given to players who have a certain number of points, usually just as a milestone for the player to feel like they have accomplished something by the time they achieve it).

# End user

The end user will include Oscar himself and the other players who will be playing the game. Players could use this game as fun, to measure their WPM or to improve their typing speed. This is achievable because there is a competitive aspect to the game where you go head to head against other players in real time.

# Proposed solution

A ranked mode is a solution seen in many competitive games today, players will get to a level where casual matches are far too easy as their skill level is so much higher than the average player’s. An example would be a first person shooter, where over time a player’s aim gets so good that they can make mistakes such as bad positioning without getting punished for it simply because they kill the other player faster. This then means when they compete against other good players who have the same aim as them, they will lose. In cases like these playing against players worse than you could make you worse at the game and in the context of typing you would get away with making more mistakes or just not typing as fast but still winning. This would mean you would stay at around the same skill level with no improvement.

By providing a competitive environment, improving is not only encouraged, it is often necessary to keep climbing the ranks (ranks are divisions given to players who have a certain number of points, usually just as a milestone for the player to feel like they have accomplished something by the time they achieve it).

My proposed solution is a game made with python, which will use an SQL database to store user information and a server to store this on. The game itself will use a peer to peer model so that players can play together without a server so that server usage would not be an issue in the case of lots of users playing at the same time, and a client server model for the matchmaking and storing of user information. The game will let the user log in to an account that they have made the first time they play the game, they can stay logged in if they wish, and they will then be able to play a ranked game, where they will be put against other players of similar rank, and they will have a race to see who can type their letters the fastest. They will be able to see their opponent’s progress in real time and the winner is decided by factoring in the time they finished typing everything in and how many mistakes they made during that time. This will then decide a winner and the winner will get a certain amount of points, depending on how big the skill gap was, and the margin they won by. The same number of points will be deducted from the loser and they will be sent back to the main menu. The main menu could include more than just matchmaking, such as a “recently played” tab which shows the outcome of recent games with certain information, but such features I will have asked Oscar about in the interview.



# Survey with client

I made a google form and gave it to Oscar to fill out with the following questions followed by his responses.

A = Armin, O = Oscar

A: What information should be stored and kept about a user?

O: Information such as games played, games won, games lost, highest rank, longest win streak, biggest win margin, are all cool things that would be nice. Of course, these would have to be account specific and so should have a section in the main menu where you can check your account statistics for information such as this.

A: Should the game have separated ranks to prevent players of different ranks playing together or should it be based purely off a point system where players will be put against other players with a similar number of points. If so, how long should it take to get up a single rank in games won.

O: Simple an ELo system where its only points would be the simplest solution to not require a lot of balancing for points gained and lost, but also need to only matchmake players within a certain range that could be defined after players have played the game for a while so it would be fine tuned to the player base. If I had to estimate, I’d say match players up against each other if they are within 5 games of points of each other.

A: How strict should the matchmaking be? For example, should they be within 3 good games’ worth of points of each other or a different metric?

O: We do not want very good typers going against quite new typers as that would just make them want to quit and this is meant to be a fun way of improving your typing skills. The idea that they get less points for winning against a much lower skilled opponent is important, as it helps find out what ELo players belong, but that does not mean there would be a boring match for both sides, one having no chance of winning and one not having to try to win. I think it would be a good idea to have them not be more than 5 games away from each other in MMR, but this could be adjusted to reduce queue times if they are too long.

A: Should players be able to see in real time how well the opponent is doing? Should there be any audio or visual effects for it?

O: Yes, they should be able to see what letters are being typed by the opponent, it is not too important for there to be too little latency but the general idea of what your opponent is doing and that they are an actual player is something that is important. Some audio effects would be appropriate, for when the opponent has reached a certain threshold.

A: Should they lose points for mistyping? Or be delayed in some way?

O: No but the person who wins should be the one who gets the most letters correct.

A: What factors should be considered to decide the winner?

O: It should depend on who types the most correct letters.

A: How should the game deal with players typing in the wrong letter?

O: The game should show that they typed the wrong letter/make a noise that it was wrong and let them continue.

A: What kind of text should be generated for the actual race? Should both players get the same phrase? How long should the phrase be? Should it include numbers?

O: It can be random from a list of already existing sentences, or it could be randomly generated as it does not matter. The phrase should be long enough that it should take about 30 seconds to type, maybe make it so that it scales with the players’ average WPM so that it would not be a 10 second game at high levels.

A: Should the user be asked to make an account the first time they play?

O: Yes, and they should login with a username that is unique to them, the email should only be used to verify they are human.

A: What should there be in the main menu?

O: A statistics section, matchmaking and logging out.

A: Should ranked points have diminishing returns so that at high levels there are less points being earned?

O: This would not be required as the skill level of players should balance things out.

A: Should there be penalties for leaving games early?

O: I would say just a loss and losing the maximum amount of ELo, the same being given to the other person.

A: Any other requests?

O: field left empty

# Objectives

1. User should be able to login and save their progress automatically:
   1. User information such as username and password will need to be collected and stored.
      1. The player should be able to create a new account.
      2. The username must be unique per player.
      3. The password will be stored encrypted and hashed with a salt
   2. Users should be able to see their statistics.
      1. Information will be recorded where appropriate client side and new information will be calculated before being uploaded to the server where the information will be checked for formatting.
   3. Users should be able to log out of their account.
2. Once logged in they should be presented the main menu.
   1. A play button which will queue the player to find a game.
   2. A leaderboards button which will show the leaderboards.
   3. An options button which will let players change certain aspects of the game.
      1. Sound volume.
      2. Window resolution
   4. Statistics button.
      1. Like the leaderboards button it will show you your account’s statistics.
         1. WPM (Words Per Minute)
         2. Games played.
         3. Games won.
         4. Longest win streak.
         5. Highest ELo.
         6. Largest win margin (In seconds).
   5. An exit button.
      1. This will open a smaller menu which will let the player logout or stay logged in.
3. Users should be able to queue into a match against someone of similar skill:
   1. The main menu should display their current ELo.
   2. The players they get matched up with must not be outside a reasonable range of their own ELo.
   3. The player will be kept in queue until an opponent is found unless they press the cancel button on screen.
4. The game should last 30 seconds
5. If a player leaves, then they should lose maximum ELo and that should be given to the player who did not leave.
6. The game should have real time updates for the opponent’s progress:
   1. The text should get highlighted a different colour as their opponent types it out.
   2. The opponent’s text background should change colours to reflect if they are ahead or behind.
   3. Sound volume should be able to be changed in the settings.
7. The player should be given the option to go back to the main menu after a game is finished.
   1. A new menu should be opened when the escape key is pressed.
   2. The buttons resume, settings and main menu should be presented.
      1. The main menu key should be locked when in a game and should unlock when the game is finished.
         1. Players will still be able to leave games through other methods outside the program.
8. The game’s ELo system should be similar to other ELo systems in games e.g. chess, League of legends (LoL) or Counter Strike: Global Offensive (CS:GO).

# ELo system

The ELo system was originally made by Arpad Elo. It is a system used to rank players based off their skill level for zero sum games. A zero-sum game in short is a game where an advantage gained by one player will mean that an equal disadvantage is given to the other player. This works in games such as this because there are only 2 players and specifically this one because one player is ahead, and the other is behind. In this case, a player getting ahead will give them the advantage of being able to make more mistakes, and a player being behind will give them the disadvantage of being able to make less mistakes.

With an ELo system, it is expected that two players with equal ELo rating should have an equal chance of winning. This however turns out to not be realistic because it is unlikely someone is completely correctly represented by their ELo rating, and that there are many factors that go into how well someone performs in a certain game.

An advantage of this system is that the winning player will take points from the losing player, so the ratings will always balance outw. This means that players will have to start with a certain number of points. The points are comparative and do not matter, as the general skill level of the playerbase will always balance out the ELo ratings of the game. It also is specific to the context where it was established, so having a high ELo in LoL will not mean the player is a very good CS:GO players.

The ELO system mainly gained traction in chess, due to the game being very fitting for this rating system. It was originally implemented in by the United States Chess Federation (USCF) in 1960 and later by the World Chess Federation in 1970. This is important as chess has certain similarities to the typing game required, for example chess is a 1v1 game where winning is what determines if you lose or gain points.

# Similar websites

There are websites that do things similar to this already that do not have a ranked mode, most notably Typeracer. Typeracer is a website that lets you race others in typing a certain phrase out. Usually the phrases are song lyrics, and it lets you see how others do in real time. The reason this isn’t the same thing is that Typeracer’s real time is based on regular intervals instead of a smooth highlighting of the words the opponent types. Typeracer also has no MMR (Matchmaking rating) system and puts you against random players, so it is very casual and not competitive. This still however does leave space for us to take inspiration, for example the leaderboards are phrase specific.

One issue I had with Typeracer was I was going up against players which were very slow, so I didn’t have to try to win. This is what the ranked mode is going to address.

The date the record is set is also a good statistic to store.

Graphical user interface

Description automatically generated

Another similar application is typing academy which is advertised as a way of learning to touch type, and is very good for that purpose, however I like the way they deal with errors.

Table

Description automatically generated with medium confidence

An error does not carry forward, while it takes a while to get used to not trying to delete the mistake, it is a good way of dealing with errors and puts a focus on speed over accuracy. When you type a letter incorrectly, the letter is added where your cursor was, and you can try to type it again.

# Connecting a client and a server

A vital part of making a multiplayer game is that two players can connect to each other through a server. This can be achieved through python with the socket library.

It is possible to make a listening server by having a while loop that is always checking for new connections, while this would normally cause the whole program to stop until a connection is made, this issue is solved by the threading library in python. This library allows you to have a function running in parallel, which allows us to have the server running all clients in parallel waiting for their responses. This does not come with its drawbacks; if the client does not end the connection using the message it needs to then the client will not be able to connect the next time it tries to, as the connection will still be open. This is dealt with by disconnecting from the server as the client whenever the game either crashes or finishes.

While programming this, I went through the following implementations of socket programming.

I tried synchronous, asynchronous and parallel with objects, with the 3rd being the most effective.

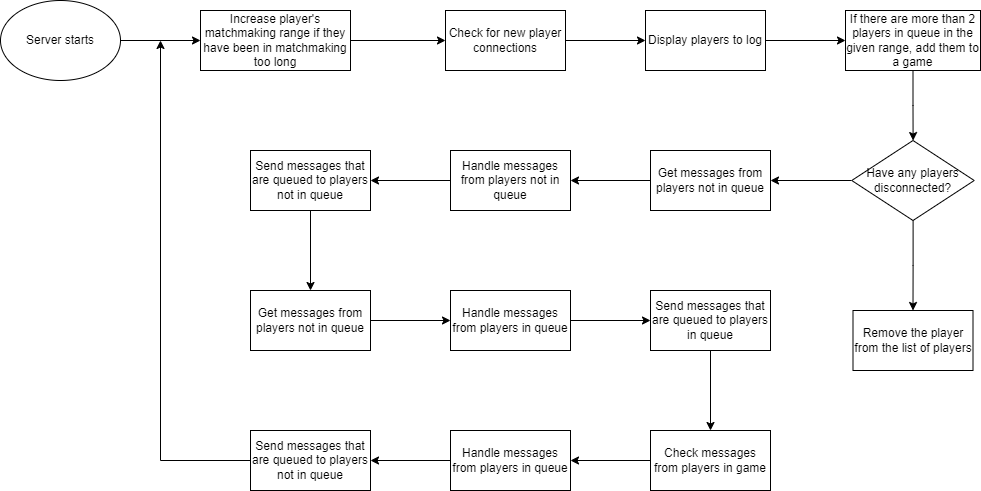
I initially tried to make them run in separate threads, but this quickly brought up issues with synchronising multiple threads. I then tried synchronous, which was because I learned of non-blocking sockets. This was a lot more successful and is in the final implementation of the final design for the server.

The final and most effective method I found was to have a thread that would simply check for messages and add them to a list stored in an object. This was a way for threads to share data but not have to wait for each other. This was what was used for the client.

## Planning for server

The server simply needs to be able to get messages and handle them. This is simple in itself but the hard part is that the client will be in different stages of the game and the server won’t be able to know which without a player object. I will go into more detail on the server later on. I will be going through each scene the player can see and explain how the client and the server interact with each other. If using word, go to view -> show -> navigation pane. This will let you navigate the word document much more easily as everything is organised using them.

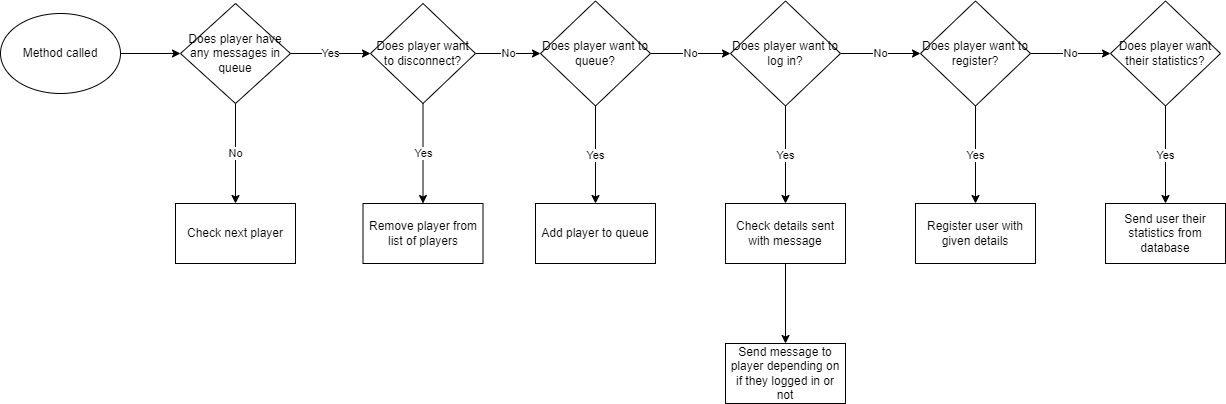
Flowchart for server finding connections to clients:



Flowchart for server sending messages to the client:

Diagram

Description automatically generated

Flowchart for server handling the client:

## Planning for ClientSocket

Flowchart for client socket connecting to the server:



Flowchart for client socket sending a message to the server:

Diagram

Description automatically generated

Flowchart for client socket getting a message from the server:

Diagram

Description automatically generated

## Python code for server

So far this doesn’t have all the functionality that it needs, but it will be expanded on as needed while working on the client.

import socket

from Game import Game

from Player import Player

class Server:

    def \_\_init\_\_(self):

        self.\_\_HEADER = 8

        self.\_\_PORT = 5000

        self.\_\_SERVER = socket.gethostbyname(socket.gethostname()) *#Gets the local IP address*

        self.\_\_ADDRESS = (self.\_\_SERVER, self.\_\_PORT) *#Makes a tuple for the address*

        self.\_\_FORMAT = 'utf-8'

        self.players = []

        self.playersInMatchmaking = []

        self.running = True *#Boolean used to close other threads once the program ends*

        self.\_\_server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) *#AF\_INET is for ipv4. SOCK\_STREAM is for TCP, SOCK\_DGRAM is UDP*

        self.\_\_server.bind(self.\_\_ADDRESS)

        print("[SERVER STARTED]")

*#Made to be used in a seperate thread*

*#Checks each player for a message being sent*

*#If a message is received it is appended to the list player.msgsReceived*

    def GetMsgs(self, players):

        for player in players:

            player.connection.setblocking(False)

            try:

                msgLen = int(player.connection.recv(self.\_\_HEADER).decode(self.\_\_FORMAT)) *#Waits for message with length 8 bytes to be received from the client and then decodes it*

            except:

                player.connection.setblocking(True)

                return 0

            player.connection.setblocking(True)

            if msgLen > 0:  *#First message will always be empty*

                msg = player.connection.recv(msgLen).decode(self.\_\_FORMAT) *#Waits for a message with length msgLen to be received*

                player.msgsReceived.append(msg)

*#Made to be used in a seperate thread*

*#Checks each player for a message that needs to be sent from player.msgsToSend list*

*#If a message needs to be sent it will send it and remove it from the list*

    def SendMsgs(self):

*#Checks queue for each player to send messages that need to be sent.*

        for player in self.players:

            player.connection.setblocking(False)

            try:

                for msg in player.msgsToSend:

                    conn = player.connection

                    encMessage = msg.encode(self.\_\_FORMAT) *#encodes msg with utf-8*

                    msgLen = len(encMessage)

                    msgLen = str(msgLen).encode(self.\_\_FORMAT)

                    msgLen += b' ' \* (self.\_\_HEADER - len(msgLen)) *#makes the message length be 8 bytes long so the server recognises it*

*#b' ' means the byte representation of a space*

                    conn.send(msgLen)

                    conn.send(encMessage)

                player.msgsToSend = []

            except socket.error:

                pass

            player.connection.setblocking(True)

*#Function to be used in the run function to look for new players and not block everything else that needs to happen*

*#Ran in parallel by using threading module*

    def \_\_CheckForNewPlayers(self):

        self.\_\_server.setblocking(False)

        try:

            conn, addr = self.\_\_server.accept() *#When connection occurs*

            thisPlayer = Player(addr, conn)

            self.players.append(thisPlayer)

        except:

            pass

        self.\_\_server.setblocking(True)

    def \_\_PrintPlayersInMatchmaking(self):*#*

        print(f"Players in matchmaking:{len(self.players) + len(self.playersInMatchmaking)}", end="\r")

    def \_\_CreateNewGame(self):

        self.currentGames.append(Game(self, self.playersInMatchmaking[0], self.playersInMatchmaking[1]))

        self.playersInMatchmaking.pop(0)

        self.playersInMatchmaking.pop(0)

        self.currentGames[-1].StartThread()

*#Goes through every message for every player in players parameter*

    def \_\_HandleMessages(self):

        disconnectedPlayers = []

        if len(self.players) > 0:   *#Check if there are any players*

*#For player in players*

            for i in range(len(self.players)):

                j = 0

                if len(self.players[i].msgsReceived) > 0:   *#Check if there are any messages*

                    while j < len(self.players[i].msgsReceived):

                        msg = self.players[i].msgsReceived[j]

                        if msg == "!DISCONNECT":

                            disconnectedPlayers.append(i)

                            self.players[i].msgsReceived = []

                        elif msg == "!QUEUE":

                            self.playersInMatchmaking.append(self.players[i])

                            disconnectedPlayers.append(i)

                            self.players[i].msgsReceived.pop(j)

*#More messages can be handled here*

                        else:

                            self.players[i].msgsReceived.pop(j)

            while len(disconnectedPlayers) > 0:

                self.players.pop(disconnectedPlayers[-1])

                disconnectedPlayers.pop(-1)

*#Does same thing for players in matchmaking*

        disconnectedPlayers = []

        if len(self.playersInMatchmaking) > 0:

*#For player in players*

            for i in range(len(self.playersInMatchmaking)):

                j = 0

                if len(self.playersInMatchmaking[i].msgsReceived) > 0:

                    while j < len(self.playersInMatchmaking[i].msgsReceived):

                        msg = self.playersInMatchmaking[i].msgsReceived[j]

                        if msg == "!DISCONNECT":

                            disconnectedPlayers.append(i)

                            self.hasPrintedNewPlayers = False

                            self.playersInMatchmaking[i].msgsReceived = []

                        elif msg == "!DEQUEUE":

                            self.players.append(self.playersInMatchmaking[i])

                            disconnectedPlayers.append(i)

                            self.hasPrintedNewPlayers = False

                            self.players[i].msgsReceived.pop(j)

*#More messages can be handled here*

                        else:

                            self.playersInMatchmaking[i].msgsReceived.pop(j)

            while len(disconnectedPlayers) > 0:

                self.playersInMatchmaking.pop(disconnectedPlayers[-1])

                disconnectedPlayers.pop(-1)

    def Run(self):

        self.\_\_server.listen() *#Looks for connections*

        self.currentGames = []

        while self.running:

            self.\_\_CheckForNewPlayers()

*#Prints players in matchmaking*

            self.\_\_PrintPlayersInMatchmaking()

*#Creates new game object with 2 players in it*

            while len(self.playersInMatchmaking) >= 2:

                self.\_\_CreateNewGame()

            self.GetMsgs(self.players)

            self.GetMsgs(self.playersInMatchmaking)

            self.SendMsgs()

            self.\_\_HandleMessages()

server = Server()

server.Run()

## Python code for client

This just gets messages and sends them using the lists.

import socket

import threading

class ClientSocket:

    def \_\_init\_\_(self):

        self.\_\_HEADER = 8

        self.\_\_PORT = 5000

        self.\_\_SERVER = socket.gethostbyname(socket.gethostname()) *#temporary // sets ip of host to client ip, which is same as host ip*

        self.\_\_FORMAT = 'utf-8'

        self.\_\_ADDRESS = (self.\_\_SERVER, self.\_\_PORT)

        self.\_\_client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) *#Same as before, ipv4 and TCP*

        self.\_\_client.connect(self.\_\_ADDRESS) *#Connects to the right local address, in this case its my own pc*

        self.connected = True

        self.msgsToSend = []

        self.receivedMsgs = []

*#Made to be used in a seperate thread*

*#Checks if any messages need to be sent*

*#Sends them to the server and removes them from the list*

    def SendMsgs(self):

        self.\_\_client.setblocking(False)

        try:

            for msg in self.msgsToSend:

                encMessage = msg.encode(self.\_\_FORMAT) *#encodes msg with utf-8*

                msgLen = str(len(encMessage)).encode(self.\_\_FORMAT)

                msgLen += b' ' \* (self.\_\_HEADER - len(msgLen)) *#makes the message length be 8 bytes long so the server recognises it*

*#b' ' means the byte representation of a space*

                self.\_\_client.send(msgLen)

                self.\_\_client.send(encMessage)

        except socket.error:

            pass

        self.\_\_client.setblocking(True)

    def GetMsgs(self):

        self.\_\_client.setblocking(False)

        try:

            msgLen = self.\_\_client.recv(self.\_\_HEADER).decode(self.\_\_FORMAT)

            msgLen = int(msgLen)

        except:

            self.\_\_client.setblocking(True)

            return 0

        self.\_\_client.setblocking(True)

        if msgLen > 0:  *#First message will always be empty*

            msg = self.\_\_client.recv(msgLen).decode(self.\_\_FORMAT) *#Waits for a message with length msgLen to be received*

            self.receivedMsgs.append(msg)

*#This function needs to make sure the message is sent before closing the socket*

    def EndConnection(self):

        self.connected = False

        self.msgsToSend.append("!DISCONNECT")

## Testing client and server

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | Client connects to server | Open server file, open multiple instances of client file. | The hello message should be shown on the server console 3 times, alongside the address of the clients. | Normal | Pass |
| 2 | Client sends message to server | The client script will send the message “Hello” to the server. | The server will print the message alongside the details of the client. | Normal | Pass |
| 3 | Server sends message to the client and the client will print it | Added a line of code in the HandleClient() function to temporarily allow inputs to be entered on the server. This is blocking input so it is inappropriate for the program, but for testing purposes this is good as it shows that the server can send a message. | The messages should be printed to the console on the client script after being typed into the server script. | Normal | Pass |
| 4 | The client sends a message to the server and the server will print that message | Same as above. I have had to add an input line into the client script where newMsg = ‘’ temporarily to test this. | The messages should be printed to the console on the client script after being typed into the server script. | Normal | Pass |

note: this had to be done in command prompt windows due to needing multiple scripts running at the same time.

### 1 and 2

The client needs to be able to connect to the server. This also needs to allow for the client to send a message to the server.   
This will be tested by sending the message “Hello” to the server. The server should then print the message alongside the details of the client.

Text

Description automatically generated

These tests are passed.

### 3

The server needs to be able to send messages to the client. As of right now, the server does not have any way of sending messages as the game has not been coded yet, however I have made it so that it can send messages to the client through an input() statement. This is only temporary and for testing purposed, as later on the server will take input from the client and do something with that to cause an output to the client.

Text

Description automatically generated

The top is the client script and the bottom is the server script. I manually typed the messages.

The messages were successfully received by the client and so this test is passed.

### 4

The server needs to be able to receive messages from the client and display them (only for testing).

To do this I added an input() to the client and allowed myself to send messages to the server.

Text

Description automatically generated

The client message to server text does remain at the top when a message is received from the server (in this case the connection established message), but it is not important as the client won’t be using inputs to send messages to the server in the end product.

# Word generation

As stated in the objectives section:

1. The game should generate a phrase long enough to last 30 seconds based off the players’ average WPM.
   1. Program should average the two players’ WPM and then divide it in half for words per 30 seconds.
   2. WPM should be recorded by the client and then uploaded to the server.

This will be impossible to implement at this stage as the statistics for players is not implemented, however we can make a class that will make an object that generates words of a certain length.

This word generator will use a random word generator to make a string of length number of random words. I originally intended to use music lyrics from an API, using 2 endpoints to get a track and to get the lyrics of that track. I ran into a lot of issues with the API key I was using, so I decided to just use this instead. I also looked into using phrases from books, but I couldn’t find a way to do that.

This word generator is a library you can install with pip from the following link:

<https://pypi.org/project/Random-Word/>

This also requires PyYaml from the following link:

<https://pypi.org/project/PyYAML/>

## WordGenerator.\_\_GetWords()

This method needs to get words

### Flowchart

Diagram

Description automatically generated

### Pseudocode

SUBROUTINE \_\_GetWords(self)

listOfWords <-- None

WHILE listOfWords = None

listOfWords <-- get random words from library

ENDWHILE

x <-- 0

WHILE x < LEN(listOfWords)

erroneousFound <-- False

check if listOfWords[i] is a letter of the alphabet

if it is then remove it from the list

ENDWHILE

RETURN listOfWords

ENDSUBROUTINE

### Python code

def \_\_GetWords(self):

    listOfWords = None

    while listOfWords == None:

        listOfWords = self.\_\_wordGenerator.get\_random\_words(hasDictionaryDef="true")

    print(listOfWords)

    x = 0

*#Removes instances with numbers*

    while x < len(listOfWords):

        erroneousFound = False

        listOfWords[x] = listOfWords[x].lower()

        for i in range(len(listOfWords[x]) - 1):

            asciiOfLetter = ord(listOfWords[x][i])

            if not (33 <= asciiOfLetter <= 47 or 58 <= asciiOfLetter <= 90 or 97 <= asciiOfLetter <= 122):

                erroneousFound = True

        x += 1

        if erroneousFound:

            listOfWords.pop(x)

    return listOfWords

## WordGenerator.\_\_MakeWordsCorrectLength()

This method needs to take the words and adjust the length. It will either need to cut them down to a shorter length or a longer length depending on how many words are required.

### Flowchart

Diagram

Description automatically generated

### Pseudocode

SUBROUTINE \_\_MakeWordsCorrectLength(self, words, length)

newWords = []

copyOfWords <-- words

WHILE LEN(newWords) < length

newWords.APPEND(copyOfWords.POP())

ENDIF

ENDWHILE

ENDSUBROUTINE

### Python code

*#Cuts lyrics down to certain length and removes newlines*

*#Words is the list of words that could be used*

*#Length is the length that the returned string needs to be*

def \_\_MakeWordsCorrectLength(self, words, length):

    newWords = []

    copyOfWords = words.copy()

*#Appends length number of words to the end of newWords*

    while len(newWords) < length:

        if copyOfWords == []:

            copyOfWords = words.copy()

        newWords.append(copyOfWords.pop())

    wordsString = " ".join(newWords)

    return wordsString

## WordGeneration.py

I will just put the code for the entire file here

from random\_word import RandomWords

class WordGenerator:

    def \_\_init\_\_(self):

        self.\_\_wordGenerator = RandomWords()

*#Gets list of random words*

    def \_\_GetWords(self):

        listOfWords = None

        while listOfWords == None:

            listOfWords = self.\_\_wordGenerator.get\_random\_words(hasDictionaryDef="true")

        print(listOfWords)

        x = 0

*#Removes instances with numbers*

        while x < len(listOfWords):

            erroneousFound = False

            listOfWords[x] = listOfWords[x].lower()

            for i in range(len(listOfWords[x]) - 1):

                asciiOfLetter = ord(listOfWords[x][i])

                if not (33 <= asciiOfLetter <= 47 or 58 <= asciiOfLetter <= 90 or 97 <= asciiOfLetter <= 122):

                    erroneousFound = True

            x += 1

            if erroneousFound:

                listOfWords.pop(x)

        return listOfWords

*#Cuts lyrics down to certain length and removes newlines*

*#Words is the list of words that could be used*

*#Length is the length that the returned string needs to be*

    def \_\_MakeWordsCorrectLength(self, words, length):

        newWords = []

        copyOfWords = words.copy()

*#Appends length number of words to the end of newWords*

        while len(newWords) < length:

            if copyOfWords == []:

                copyOfWords = words.copy()

            newWords.append(copyOfWords.pop())

        wordsString = " ".join(newWords)

        return wordsString

*#main function that is to generate a number of words to be displayed in the game*

    def GetWordsForProgram(self, length):

        return self.\_\_MakeWordsCorrectLength(self.\_\_GetWords(), length)

## Testing for word generation:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | Returns words | I will update the file to create an object and call the method. | It will return random words of given length | Normal | Pass |
| 2 | Returns words with correct word count | I will run the class’s GetWordsForProgram() method with parameter 50 | It will return 50 words | Normal | Pass |
| 3 | Returns words with large word count | I will run the class’s GetWordForProgram() method with parameter 500 | It will loop back to the start of the words list when it runs out of words | Normal | Pass |
| 4 | Returns string with no uppercase letters | I will run the class’s GetWordsForProgram() method with parameter 50 | It will return words that are all lowercase | Normal | Pass |

### Test 1 and 2

Code added:

generator = WordGenerator()

print(generator.GetWordsForProgram(50))

The program should return 50 random words.

The output was:

A screenshot of a computer

Description automatically generated with medium confidence

When pasted into word it is 50 words. Test 1 is passed as words were returned as a string. This also passes test 2 as it is the correct number of words.

### Test 3

The program may potentially need to generate more words than the length of the words list returned by the module, so the chosen solution to this is that the program will add the words to a string and when it runs out of words in the list it will go to the beginning of the words list.

The following code was added to the bottom of the WordGeneration.py file:

generator = WordGenerator()

print(generator.GetWordsForProgram(500))

The output was as follows:

Text

Description automatically generated

When counted this was 500 words so the test passed.

### Test 4

The program will need to ensure that all letters are lowercase as the player will be entering the letters as one large sentence. This is done by using the String.lower() method.

The following code was added to the bottom of the WordGeneration.py file:

generator = WordGenerator()

print(generator.GetWordsForProgram(50))

The output was as follows:



All the letters are lowercase, so this test is passed.

# Client

The client will need to connect, send and receive data to/from the server. The client will have multiple game loops which are Scene objects. The client is made up of many classes, a UML diagram for it is shown below:

## Play.py

This is the main file I decided to have that the player would launch, this file’s python code is below:

import ctypes, pygame, json

from Game import Game

*#Imports settings*

file = open("settings.json", "r")

settings = json.load(file)

file.close()

*# #Volume value is a percentage*

*# settings = {*

*#     "Volume": 100,*

*#     "Resolution": "Fullscreen"*

*# }*

user32 = ctypes.windll.user32

*#Prevents the screen from scaling with windows resolution scale*

*#System -> Display -> Scale and Layout*

user32.SetProcessDPIAware()

if settings["Resolution"] == "Fullscreen":

    window = pygame.display.set\_mode((0,0), pygame.FULLSCREEN)

else:

    res = settings["Resolution"].split("x")

    window = pygame.display.set\_mode((int(res[0]), int(res[1])))

pygame.display.set\_caption("SpeedTyper")

pygame.font.init()

game = Game(window, settings)

game.main()

if game.socket is not None:

    game.socket.EndConnection()

pygame.font.quit()

pygame.quit()

As you can see, whenever the player quits the game for any reason the game will check if the socket isn’t None and if it isn’t it will end the connection from the server.

The EndConnection method of the ClientSocket object is below:

*#This function needs to make sure the message is sent before closing the socket*

def EndConnection(self):

    encMessage = "!DISCONNECT".encode(self.\_\_FORMAT) *#encodes msg with utf-8*

    msgLen = str(len(encMessage)).encode(self.\_\_FORMAT)

    msgLen += b' ' \* (self.\_\_HEADER - len(msgLen)) *#makes the message length be 8 bytes long so the server recognises it*

*#b' ' means the byte representation of a space*

    self.\_\_client.send(msgLen)

    self.\_\_client.send(encMessage)

    self.connected = False

This is handled by the server (more details on the server after the client section), by the following sections of code depending on what stage the client is in(pseudocode is found in server section later):

For when player is in the menus:

def HandleMessagesForPlayersNotInQueue(self):

    playersQuit = []

    for player in self.players:

        while player.msgsReceived.GetLength() != 0:

            message = player.msgsReceived.Dequeue()

            if message == "!DISCONNECT":

                playersQuit.append(player)

For when player is in matchmaking:

def HandleMessagesForPlayersInQueue(self):

    playersQuit = []

    for player in self.playersInMatchmaking:

        while player.msgsReceived.GetLength() != 0:

            message = player.msgsReceived.Dequeue()

            print(message)

            if message == "!DISCONNECT":

                playersQuit.append(player)

For when player is in game:

def CheckIfPlayerFinishedGame(self):

    playersQuit = []

    for player in self.playersInGame:

        if player.gameFinished:

            playersQuit.append(player)

            self.players.append(player)

        elif player.playerQuit:

            playersQuit.append(player)

*#Removes players who quit from the list of players*

    for player in playersQuit:

        listIndex = LinearSearch(player, self.playersInGame)

*#Linear search returns None if item is not in list*

        if listIndex is not None:

            self.playersInGame.pop(listIndex)

## UML diagram

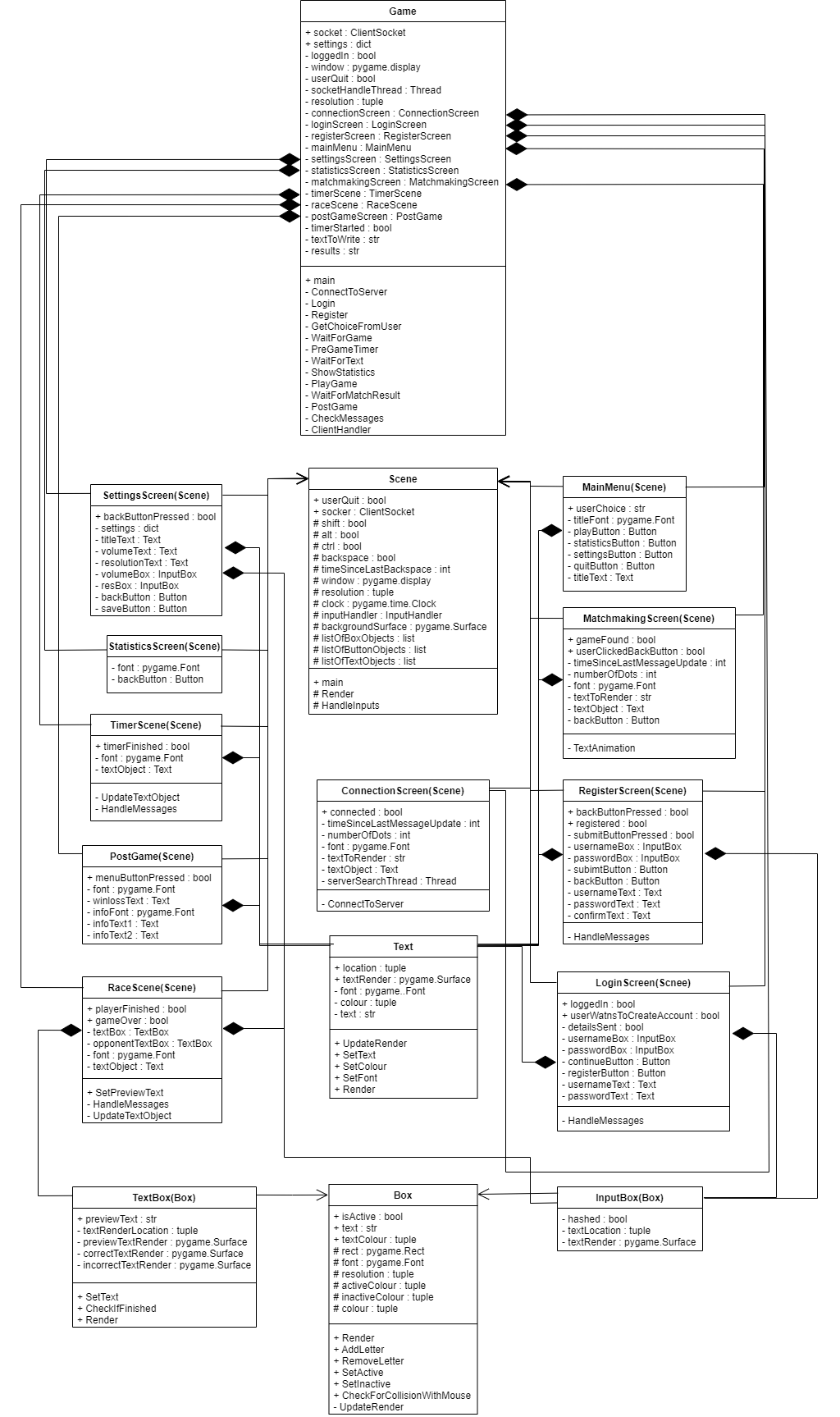
To play the game, the player would open the Play.py file. This file makes a Game object and gives it information about the settings, the resolution and the settings.

This Game object has certain functions which act as “Game loops”. These loops which are just methods in the Game object simply use Scene object which store information about the current “scene”. A scene is simply a stage that the program is in, for example in a shooter game one scene would be the main menu, another would be choosing their loadout, or another would be the main gameplay of going around and shooting other things.

The scenes will store information about the user’s inputs to determine if the loop should end or not, which is detected by Game. I will go into this further in the Game section.

The game is made up of scenes which are made up of a mixture of Boxes, Buttons and Text. Box is a class that is inherited by InputBox and TextBox.

The diagram below isn’t exhaustive, for example socket and window are shared around quite a lot, but as it is already quite cluttered and hard to read I decided not to include those details, the diagram below is to illustrate the way scenes and boxes interact with the Game class.



The ClientSocket is an object started earlier in the document and is finished slightly after here. The object allows communication between the client and the server, with functions like SendMsg(), GetMsgs() and EndConnection(). This will be used to check for messages from the server in order to interpret them into things in-game.

The Text class makes an object that can be used to store text and a font and a location, so that rendering is much easier and things that need to be rendered can be added to a list and rendered that way.

## Input Handling throughout the program

This object needs to check for inputs and translate them into commands that will be returned as an array to be used in the main program.

### Flowchart

Diagram

Description automatically generated

This method of handling input seems unnecessary but it allows me to pause the usage of certain inputs in case they need to be queued.

### Pseudocode

This does not really require pseudocode as it is just a bunch of IF statements.

### Python code for handling input:

This is using an example for the textbox being updated during the main game.

import pygame

*#Object that stores list of inputs made by user*

class InputHandler:

    def \_\_init\_\_(self):

*#Priority queue for storing userinputs*

        self.inputsList = []

*#Checks for inputs and adds them to the priority queue*

    def CheckInputs(self):

*#For event in events that happened*

        for event in pygame.event.get():

*#If the player quit*

            if event.type == pygame.QUIT:

*#High priority = front of queue*

                self.inputsList.append("QUIT")

*#If player clicks, appends the mouse location to the command that player clicked*

            elif event.type == pygame.MOUSEBUTTONDOWN:

                mousePos = pygame.mouse.get\_pos()

                command = f"CLICK:{mousePos[0]},{mousePos[1]}"

                self.inputsList.append(command)

*#If player stops clicking, same as clicking*

            elif event.type == pygame.MOUSEBUTTONUP:

                mousePos = pygame.mouse.get\_pos()

                command = f"UNCLICK:{mousePos[0]},{mousePos[1]}"

                self.inputsList.append(command)

*#Handling keyboard inputs*

            elif event.type == pygame.KEYDOWN:

                if event.key == pygame.K\_RETURN:

                    self.inputsList.append("RETURNDOWN")

                elif event.key == pygame.K\_LSHIFT or event.key == pygame.K\_RSHIFT:

                    self.inputsList.append("SHIFTDOWN")

                elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:

                    self.inputsList.append("CONTROLDOWN")

                elif event.key == pygame.K\_LALT or event.key == pygame.K\_RALT:

                    self.inputsList.append("ALTDOWN")

                elif event.key == pygame.K\_BACKSPACE:

                    self.inputsList.append("BACKSPACEDOWN")

                elif event.key == pygame.K\_TAB:

                    self.inputsList.append("TABDOWN")

                else:

*#KD = KeyDown*

                    self.inputsList.append(f"KD\_{event.unicode}")

            elif event.type == pygame.KEYUP:

                if event.key == pygame.K\_RETURN:

                    self.inputsList.append("RETURNUP")

                elif event.key == pygame.K\_LSHIFT or event.key == pygame.K\_RSHIFT:

                    self.inputsList.append("SHIFTUP")

                elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:

                    self.inputsList.append("CONTROLUP")

                elif event.key == pygame.K\_LALT or event.key == pygame.K\_RALT:

                    self.inputsList.append("ALTUP")

                elif event.key == pygame.K\_BACKSPACE:

                    self.inputsList.append("BACKSPACEUP")

                elif event.key == pygame.K\_TAB:

                    self.inputsList.append("TABUP")

                else:

*#KU = KeyUp*

                    self.inputsList.append(f"KU\_{event.unicode}")

### Testing for InputHandler:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | Logs the correct inputs | I will perform certain inputs | The terminal will print the relevant commands | Normal | Pass |
| 2 | Returns QUIT when the player enters the relevant input | I will press alt + f4 and I will also press the close window button in the top right | The terminal will print that the player has quit | Normal | Pass |
| 3 | Detects if the player has clicked on the box or outside the box | I will click on the textbox | The terminal will print that the player has clicked on the box | Normal | Pass |

The program will need to be

        for command in commands:

            print(f"[COMMAND]{command} ", end="")

#### 1

The program needs to recognise certain inputs from the player in order to either display the letters, close the program or something else. In this case it will need to log everything that I do and will therefore need to display it after the test. To achieve this, I have added a print statement to print all the commands every time this method is called:

The results of this test after having typed is:

Text

Description automatically generated

The object logs commands and therefore this test is passed.

#### 2

The program will need to allow the player to quit at any point, therefore the InputHandler object will need to be able to detect if the player has quit the program.

To test this, I will simply press alt and f4 at the same time and see if the program closes. I will also do this with the close window button.

When pressing alt + f4:



When pressing the cross in the top right:



The object detects when the player has closed the application, therefore this test is passed.

#### 3

The object will need to detect if the box has been clicked on, so to test this I will click on the box and outside the box and see what is printed on the terminal.

Text

Description automatically generated

This shows that the object detects if the player has clicked on the box or not, therefore this test is passed.

## ClientSocket

The clientsocket needed some extra functionality that I coded as I went along, but most the original documentation and testing is still relevant for this, so refer to that for more information.

### Python code

import socket

class ClientSocket:

    def \_\_init\_\_(self):

        self.\_\_HEADER = 8

        self.\_\_PORT = 5000

        self.\_\_SERVER = socket.gethostbyname(socket.gethostname()) *#temporary // sets ip of host to client ip, which is same as host ip*

        self.\_\_FORMAT = 'utf-8'

        self.\_\_ADDRESS = (self.\_\_SERVER, self.\_\_PORT)

        self.\_\_client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) *#Same as before, ipv4 and TCP*

        self.\_\_client.connect(self.\_\_ADDRESS) *#Connects to the right local address, in this case its my own pc*

        self.connected = True

        self.msgsToSend = []

        self.receivedMsgs = []

*#Made to be used in a seperate thread*

*#Checks if any messages need to be sent*

*#Sends them to the server and removes them from the list*

    def SendMsgs(self):

        self.\_\_client.setblocking(False)

        unsentMessages = []

        while self.msgsToSend != []:

            message = self.msgsToSend.pop(0)

            try:

                encMessage = message.encode(self.\_\_FORMAT) *#encodes msg with utf-8*

                msgLen = str(len(encMessage)).encode(self.\_\_FORMAT)

                msgLen += b' ' \* (self.\_\_HEADER - len(msgLen)) *#makes the message length be 8 bytes long so the server recognises it*

*#b' ' means the byte representation of a space*

                self.\_\_client.send(msgLen)

                self.\_\_client.send(encMessage)

                print(f"Message Sent:{message}")

            except socket.error:

                unsentMessages.append(message)

        self.\_\_client.setblocking(True)

        for message in unsentMessages:

            self.msgsToSend.append(message)

    def GetMsgs(self):

        self.\_\_client.setblocking(False)

        msgLen = 0

        try:

            msgLen = self.\_\_client.recv(self.\_\_HEADER).decode(self.\_\_FORMAT)

            msgLen = int(msgLen)

            if msgLen > 0:

                self.\_\_client.setblocking(True)

                msg = self.\_\_client.recv(msgLen)

                try:

                    msg = msg.decode(self.\_\_FORMAT) *#Waits for a message with length msgLen to be received*

                except:

                    pass

                self.receivedMsgs.append(msg)

                print(f"Message Received:{msg}")

        except:

            self.\_\_client.setblocking(True)

*#This function needs to make sure the message is sent before closing the socket*

    def EndConnection(self):

        encMessage = "!DISCONNECT".encode(self.\_\_FORMAT) *#encodes msg with utf-8*

        msgLen = str(len(encMessage)).encode(self.\_\_FORMAT)

        msgLen += b' ' \* (self.\_\_HEADER - len(msgLen)) *#makes the message length be 8 bytes long so the server recognises it*

*#b' ' means the byte representation of a space*

        self.\_\_client.send(msgLen)

        self.\_\_client.send(encMessage)

        self.connected = False

### Explanation for non-blocking sockets

The changes are mostly just that sending and getting messages are now designed for being ran every frame in another thread. This was so that they could be done in order so as to not clash with each other.

Due to the nature of the game, there will not always be a message from the server for the client to receive. To work around this I used a non blocking socket, which meant that if both getting and sending messages were done in separate threads, they would often change the blocking status of the socket and mess up the operation of the other thread. This was easily worked around by simply making a method in the Game object which is run in a separate thread which will just run these in order.

*#Handles getting and sending of messages*

*#Necessary as if there are 2 threads then they will mistime the blocking settings*

    def \_\_ClientHandler(self):

        while self.socket.connected:

            self.socket.GetMsgs()

            self.socket.SendMsgs()

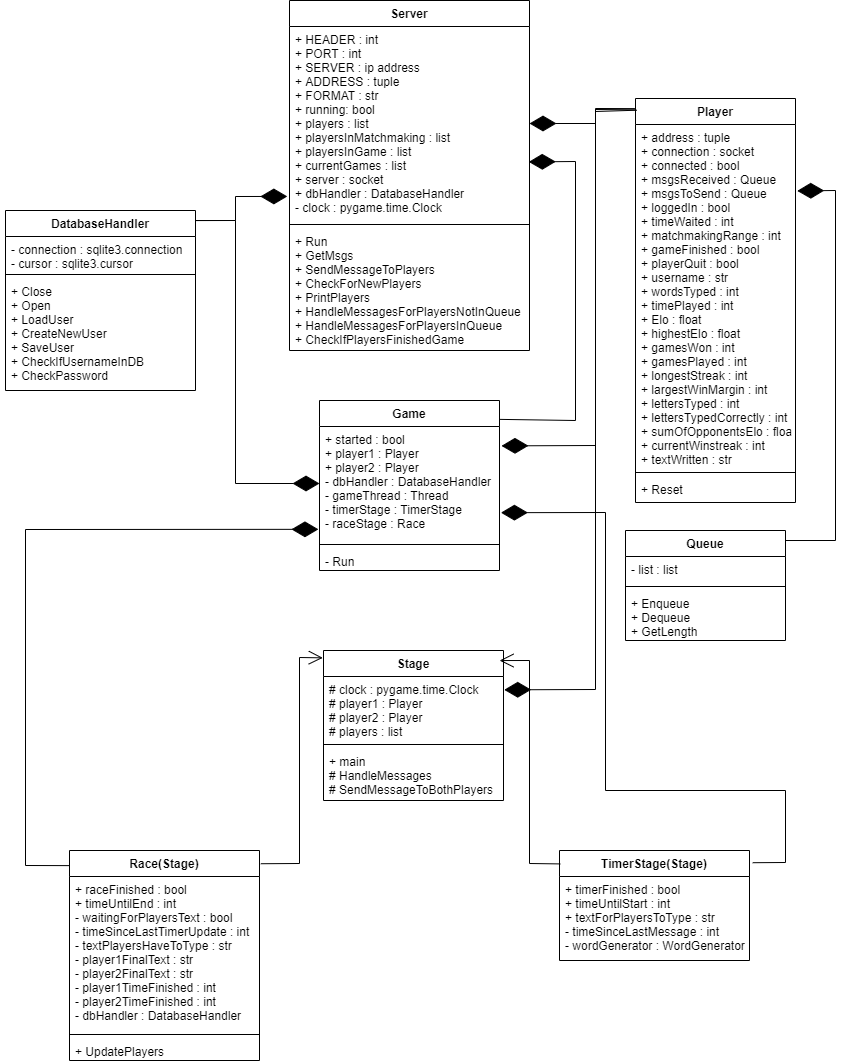
# Server

The server is there to facilitate the game for both players. The players will need to be able to connect to the server, send messages and get responses.

This can be done using sockets, and the proof of concept has been done previously in this document in the “connecting a client and server” section.

The server must be able to keep track of the players, add them to the proper queue depending on their message and then put them in a game if a player with a similar (defined below) ELo rating is found. The server needs to then send the player’s text to the other player and at the end of the game update their stats.

## UML diagram



The server uses 3 lists to determine if a player is in queue, in game or neither. This is useful as it allows different things to happen depending on which they are in.

When enough players are found to start a game, the CreateGame() method is called and the players are moved from the queue list to the in game list. The CreateGame method will make a Game object which creates a thread that the game is carried out in.

### Pseudocode

CLASS Server

SUBROUTINE init(self)

self.HEADER <-- 8

self.PORT <-- 5000

self.SERVER <-- local ip address

self.ADDRESS <-- (self.SERVER, self.ADDRESS)

self.FORMAT <-- 'utf-8'

self.running <-- True

self.players <-- []

self.playersInMatchmaking <-- []

self.playersInGame <-- []

self.currentGames <-- []

self.server <-- socket object

self.dbHandler <-- DatabaseHandler object

OUTPUT "[SERVER STARTED]"

self.\_\_clock <-- pygame clock object

ENDSUBROUTINE

SUBROUTINE Run(self)

WHILE self.running

tick clock

FOR player IN self.playersInMatchmaking

player.timeWaited <-- player.timeWaited + time since last frame

IF player.timeWaited >= 200000 then

player.matchmakingRange <-- player.matchmakingRange \* 1.5 ^ player.timeWaited // 20000

ENDIF

ENDFOR

self.CheckForNewPlayers()

self.PrintPlayers()

IF 2 players in self.playersInMatchmaking in each other's ELo range then

self.CreateGame()

ENDIF

remove disconnected players from all lists

self.GetMsgs(self.players)

self.HandleMessagesForPlayersNotInQueue()

self.SendMessageToPlayers(self.players)

self.GetMsgs(self.playersInMatchmaking)

self.HandleMessagesForPlayersInQueue()

self.SendMessageToPlayers(self.playersInMatchmaking)

self.GetMsgs(self.playersInGame)

self.CheckIfPlayerFinishedGame()

self.SendMessageToPlayers(self.playersInGame)

ENDWHILE

self.dbHandler.Close()

ENDSUBROUTINE

SUBROUTINE GetMsgs(self, players)

FOR player IN players

try to get message from player

ENDFOR

ENDSUBROUTINE

SUBROUTINE SendMessageToPlayers(self, players)

FOR player IN players

try to send message to players

ENDFOR

ENDSUBROUTINE

SUBROUTINE CheckForNewPlayers(self)

try to accept a connection

ENDSUBROUTINE

#Displays list of players to console

SUBROUTINE PrintPlayers(self)

OUTPUT 3 lists of players

ENDSUBROUTINE

SUBROUTINE HandleMessagesForPlayersNotInQueue(self)

playersQuit <-- []

FOR player IN self.players

WHILE player.msgsReceived.GetLength() <> 0

message <-- player.msgsReceived.Dequeue()

IF message = "!DISCONNECT" THEN

playersQuit.APPEND(player)

ELSE IF player.loggedIn AND message = "!QUEUE" THEN

playersQuit.APPEND(player)

self.playersInMatchmaking.APPPEND(player)

player.timeWaited <-- 0

ELSE IF message[:7] = "!LOGIN:" THEN

IF NOT player.loggedIn THEN

details <-- split message with ","

username <-- details[0]

password <-- details[1]

IF self.dbHandler.CheckIfUsernameInDB(username) THEN

IF self.dbHandler.CheckPassword(username, password) THEN

player.loggedIn <-- True

player.username <-- username

self.dbHandler.LoadUser(player)

player.msgsToSend.Enqueue("!PASSWORDCORRECT")

ELSE

player.msgsToSend.Enqueue("!PASSWORDINCORRECT")

ENDIF

ELSE

player.msgsToSend.Enqueue("!USERNAMENOTFOUND")

ENDIF

ELSE

player.msgsToSend.Enqueue("!ALREADYLOGGEDIN")

ENDIF

ELSE IF message[:10] = "REGISTER:" THEN

details <-- split message with ","

player.usename <-- details[0]

TRY

self.dbHandler.CreateNewUser(details[0], details[1])

data <-- self.dbHandler.LoadUser(player)

player.msgsToSend.Enqueue("!REGISTEREDSUCCESFULLY")

player.msgsToSend.Enqueue(data)

EXCEPT

#Username was in database already

player.msgsToSend.Enqueue("!ANERROROCCURRED")

ELSE IF message = "!STATISTICS" THEN

data <-- self.dbHandler.LoadUser(player)

player.msgsToSend.Enqueue(data)

OUTPUT "Data sent"

ENDIF

ENDWHILE

ENDFOR

FOR player in playersQuit

listIndex <-- LinearSearch(player, self.players)

IF listIndex <> None THEN

self.players.POP(listIndex)

ENDIF

ENDFOR

ENDSUBROUTINE

SUBROUTINE HandleMessagesForPlayersInQueue(self)

playersQuit <-- []

FOR player IN self.playersInMatchmaking

WHILE player.msgsReceived.GetLength() <> 0

message <-- player.msgsReceived.Dequeue()

OUTPUT message

IF message = "!DISCONNECT" THEN

playersQuit.APPEND(player)

ELSE IF message = "!DEQUEUE" THEN

playersQuit.APPEND(player)

self.players.APPEND(player)

ENDIF

ENDWHILE

ENDFOR

FOR player IN playersQuit

listIndex <-- LinearSearch(player, self.playersInMatchmaking)

IF listIndex <> None THEN

self.playersInMatchmaking.POP(listIndex)

ENDIF

ENDFOR

ENDSUBROUTINE

SUBROUTINE CheckIfPlayerFinishedGame(self)

check if any players have disconnected

ENDSUBROUTINE

ENDCLASS

### Python code

import socket, pygame, pickle

from Game import Game

from Player import Player

from Database import DatabaseHandler

from SearchingAlgorithms import LinearSearch

class Server:

    def \_\_init\_\_(self):

        self.HEADER = 8

        self.PORT = 5000

        self.SERVER = socket.gethostbyname(socket.gethostname()) *#Gets the local IP address*

        self.ADDRESS = (self.SERVER, self.PORT) *#Makes a tuple for the address*

        self.FORMAT = 'utf-8'

        self.running = True

        self.players = []

        self.playersInMatchmaking = []

        self.playersInGame = []

        self.currentGames = []

        self.server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) *#AF\_INET is for ipv4. SOCK\_STREAM is for TCP, SOCK\_DGRAM is UDP*

        self.server.bind(self.ADDRESS)

        self.dbHandler = DatabaseHandler()

        print("[SERVER STARTED]")

        self.\_\_clock = pygame.time.Clock()

    def Run(self):

        self.server.listen() *#Looks for connections*

        while self.running:

            self.\_\_clock.tick()

*#Update how long a player has been in queue*

            for player in self.playersInMatchmaking:

                player.timeWaited += self.\_\_clock.get\_time()

*#If player has waited 20 seconds in queue, their mmr range will increase*

*#Player's mmr range increases by 50% every 20 seconds*

                if player.timeWaited >= 20000:

                    player.matchmakingRange \*= 1.5 \*\* (player.timeWaited // 20000)

            self.CheckForNewPlayers()

            self.PrintPlayers()

*#Creates game if there is more than 1 player in queue*

            playersGameNotFound = []

            while len(self.playersInMatchmaking) >= 2:

*#Loop through each player*

*#Check if any other players are within their elo range*

                player1 : Player = self.playersInMatchmaking.pop(0)

                gameMade = False

                for i in range(len(self.playersInMatchmaking)):

                    if player1.Elo - player1.matchmakingRange < self.playersInMatchmaking[i].Elo < player1.Elo + player1.matchmakingRange:

                        player2 = self.playersInMatchmaking.pop(i)

                        player1.msgsToSend.Enqueue("!GAMEFOUND")

                        player2.msgsToSend.Enqueue("!GAMEFOUND")

                        self.playersInGame.append(player1)

                        self.playersInGame.append(player2)

                        self.currentGames.append(Game(player1, player2))

                        gameMade = True

                        break

                if not gameMade:

                    playersGameNotFound.append(player1)

            for player in playersGameNotFound:

                self.playersInMatchmaking.append(player)

*#Checks if any players in game have disconnected*

            i = 0

            while i < len(self.playersInGame):

                if not self.playersInGame[i].connected:

                    self.playersInGame.pop(i)

                else:

                    i += 1

*#Handling messages*

            self.GetMsgs(self.players)

            self.HandleMessagesForPlayersNotInQueue()

            self.SendMessageToPlayers(self.players)

            self.GetMsgs(self.playersInMatchmaking)

            self.HandleMessagesForPlayersInQueue()

            self.SendMessageToPlayers(self.playersInMatchmaking)

            self.GetMsgs(self.playersInGame)

            self.CheckIfPlayerFinishedGame()

            self.SendMessageToPlayers(self.playersInGame)

        self.dbHandler.Close()

*#Made to be used in a seperate thread*

*#Checks each player for a message being sent*

*#If a message is received it is appended to the list player.msgsReceived*

    def GetMsgs(self, players):

        for player in players:

            player.connection.setblocking(False)

            try:

                msgLen = int(player.connection.recv(self.HEADER).decode(self.FORMAT)) *#Waits for message with length 8 bytes to be received from the client and then decodes it*

                player.connection.setblocking(True)

                if msgLen > 0:  *#First message will always be empty*

                    msg = player.connection.recv(msgLen).decode(self.FORMAT) *#Waits for a message with length msgLen to be received*

                    player.msgsReceived.Enqueue(msg)

                    print(f"Message Received:{msg}")

                player.connection.setblocking(True)

            except socket.error:

                player.connection.setblocking(True)

*#Made to be used in a seperate thread*

*#Checks each player for a message that needs to be sent from player.msgsToSend list*

*#If a message needs to be sent it will send it and remove it from the list*

    def SendMessageToPlayers(self, listOfPlayers):

*#Checks queue for each player to send messages that need to be sent.*

        for player in listOfPlayers:

            player.connection.setblocking(False)

            while player.msgsToSend.GetLength() != 0:

                message = player.msgsToSend.Dequeue()

                try:

                    conn = player.connection

                    try:

                        encMessage = message.encode(self.FORMAT) *#encodes msg with utf-8*

                        msgLen = len(encMessage)

                        msgLen = str(msgLen).encode(self.FORMAT)

                        msgLen += b' ' \* (self.HEADER - len(msgLen)) *#makes the message length be 8 bytes long so the server recognises it*

*#b' ' means the byte representation of a space*

                        conn.send(msgLen)

                        conn.send(encMessage)

                        print(f"Message sent:{message}")

*#There will be an error if the message is a pickle as you cant encode a pickle*

                    except:

                        msgLen = len(message)

                        msgLen = str(msgLen).encode(self.FORMAT)

                        msgLen += b' ' \* (self.HEADER - len(msgLen)) *#makes the message length be 8 bytes long so the server recognises it*

*#b' ' means the byte representation of a space*

                        conn.send(msgLen)

                        conn.send(message)

                except socket.error:

                    player.msgsToSend.Enqueue(message)

            player.connection.setblocking(True)

*#Function to be used in the run function to look for new players and not block everything else that needs to happen*

*#Ran in parallel by using threading module*

    def CheckForNewPlayers(self):

        self.server.setblocking(False)

        try:

            conn, addr = self.server.accept() *#When connection occurs*

            thisPlayer = Player(addr, conn)

            self.players.append(thisPlayer)

        except:

            pass

        self.server.setblocking(True)

    def PrintPlayers(self):

        print(f"Players:{len(self.players)}, in matchmaking:{len(self.playersInMatchmaking)}, in game:{len(self.playersInGame)}", end="\r")

    def HandleMessagesForPlayersNotInQueue(self):

        playersQuit = []

        for player in self.players:

            while player.msgsReceived.GetLength() != 0:

                message = player.msgsReceived.Dequeue()

                if message == "!DISCONNECT":

                    playersQuit.append(player)

                elif player.loggedIn and message == "!QUEUE":

                    playersQuit.append(player)

                    self.playersInMatchmaking.append(player)

                    player.timeWaited = 0

                elif message[:7] == "!LOGIN:":

                    if not player.loggedIn:

                        details = message[7:].split(",")

                        username = details[0]

                        password = details[1]

*#Checks password and sends message to user to confirm if they are signed in or not*

                        if self.dbHandler.CheckIfUsernameInDB(username):

                            if self.dbHandler.CheckPassword(username, password):

                                player.loggedIn = True

                                player.username = username

                                self.dbHandler.LoadUser(player)

                                player.msgsToSend.Enqueue("!PASSWORDCORRECT")

                            else:

                                player.msgsToSend.Enqueue("!PASSWORDINCORRECT")

                        else:

                            player.msgsToSend.Enqueue("!USERNAMENOTFOUND")

                    else:

                        player.msgsToSend.Enqueue("!ALREADYLOGGEDIN")

                elif message[:10] == "!REGISTER:":

                    details = message[10:].split(",")

                    player.username = details[0]

                    try:

                        self.dbHandler.CreateNewUser(details[0], details[1])

                        data = self.dbHandler.LoadUser(player)

                        player.msgsToSend.Enqueue("!REGISTEREDSUCCESFULLY")

                        player.msgsTosend.Enqueue(pickle.dumps(data))

                    except:

                        player.msgsToSend.Enqueue("!ANERROROCCURRED")

                elif message == "!STATISTICS":

                    data = self.dbHandler.LoadUser(player)

                    player.msgsToSend.Enqueue(pickle.dumps(data))

                    print("Pickle sent")

*#Removes players who quit from the list of players*

        for player in playersQuit:

            listIndex = LinearSearch(player, self.players)

*#Linear search returns None if item is not in list*

            if listIndex is not None:

                self.players.pop(listIndex)

    def HandleMessagesForPlayersInQueue(self):

        playersQuit = []

        for player in self.playersInMatchmaking:

            while player.msgsReceived.GetLength() != 0:

                message = player.msgsReceived.Dequeue()

                print(message)

                if message == "!DISCONNECT":

                    playersQuit.append(player)

                elif message == "!DEQUEUE":

                    playersQuit.append(player)

                    self.players.append(player)

*#Removes players who quit from the list of players*

        for player in playersQuit:

            listIndex = LinearSearch(player, self.playersInMatchmaking)

*#Linear search returns None if item is not in list*

            if listIndex is not None:

                self.playersInMatchmaking.pop(listIndex)

    def CheckIfPlayerFinishedGame(self):

        playersQuit = []

        for player in self.playersInGame:

            if player.gameFinished:

                playersQuit.append(player)

                self.players.append(player)

            elif player.playerQuit:

                playersQuit.append(player)

*#Removes players who quit from the list of players*

        for player in playersQuit:

            listIndex = LinearSearch(player, self.playersInGame)

*#Linear search returns None if item is not in list*

            if listIndex is not None:

                self.playersInGame.pop(listIndex)

server = Server()

server.Run()

### Testing

The testing for the Server is in the connecting a client and server section, and it has not changed since then. The only difference will be that the server should be able to put players into games, which will be tested in the Game object section.

## Player

The player object is quite simple and is used for storing data about each player that connects to the game.

I didn’t write pseudocode for this because it is just some attributes.

### Python code

from Datastructures import Queue

class Player:

    def \_\_init\_\_(self, address, connection):

        self.address = address

        self.connection = connection

        self.connected = True

        self.msgsToSend = Queue()

        self.msgsReceived = Queue()

        self.loggedIn = False

        self.timeWaited = 0

        self.matchmakingRange = 400

        self.gameFinished = False

        self.playerQuit = False

        self.username = ""

        self.wordsTyped = 0

        self.timePlayed = 0

        self.Elo = 0

        self.highestElo = 0

        self.gamesWon = 0

        self.gamesPlayed = 0

        self.longestStreak = 0

        self.largestWinMargin = 0

        self.lettersTyped = 0

        self.lettersTypedCorrectly = 0

        self.sumOfOpponentsElo = 0

        self.currentWinstreak = 0

        self.textWritten = ""

    def Reset(self):

        self.textWritten = ""

#### Testing

This only has 1 method so it only has 1 test. The following code was used to test it:

player = Player(None, None)

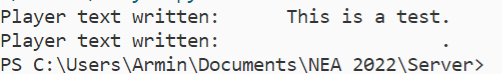
player.textWritten = "This is a test"

print("Player text written:{text:>20}.".format(text=player.textWritten))

player.Reset()

print("Player text written:{text:>20}.".format(text=player.textWritten))

When run this was printed:

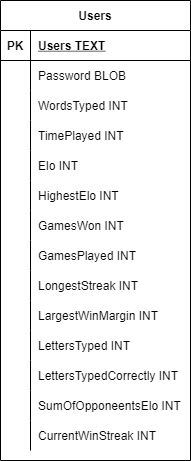


This test is passed.

## Database

The database is super simple and isn’t a focus of this project. This database consists of 1 table and is updated at the end of every game to hold the required information.

### Entity diagram



Password is a BLOB data type as it is stored as a hash and not a string.

### Database Handler

The database is controlled by an object called DatabaseHandler. This is just to encapsulate the code and make it easier to interact with the database around the program.

The database handler needs to be able to create users, load users, update users, check if a user is in the database and to check passwords for users.

This was mainly SQL and assigning things so I didn’t write pseudocode for this so here is the python code:

#### Python code

import sqlite3, random

from PasswordHandling import CheckPW, GenHash

from Player import Player

*#An object that will be able to handle all the operations needed related to the database*

class DatabaseHandler:

    def \_\_init\_\_(self):

*#Connects to database*

        self.\_\_connection = None

        self.\_\_cursor = None

        self.Open()

*#Creates the tables needed if they do not exist*

        command = """

        CREATE TABLE IF NOT EXISTS

        Users(

            Username TEXT PRIMARY KEY,

            Password BLOB,

            WordsTyped INTEGER,

            TimePlayed INTEGER,

            Elo INTEGER,

            HighestElo INTEGER,

            GamesWon INTEGER,

            GamesPlayed INTEGER,

            LongestStreak INTEGER,

            LargestWinMargin FLOAT,

            LettersTyped INTEGER,

            LettersTypedCorrectly INTEGER,

            SumOfOpponentsELo INTEGER,

            CurrentWinstreak INTEGER

            )"""

        self.\_\_cursor.execute(command)

        self.Close()

    def Close(self):

        self.\_\_connection.commit()

        self.\_\_connection.close()

    def Open(self):

        self.\_\_connection = sqlite3.connect("db.db")

        self.\_\_cursor = self.\_\_connection.cursor()

    def LoadUser(self, player : Player):

        self.Open()

        params = (player.username,)

        command = """

        SELECT \* FROM Users

        WHERE Username = (?)

        """

        self.\_\_cursor.execute(command, params)

        data = self.\_\_cursor.fetchall()[0]

        player.wordsTyped = int(data[2])

        player.timePlayed = int(data[3])

        player.Elo = int(data[4])

        player.highestElo = int(data[5])

        player.gamesWon = int(data[6])

        player.gamesPlayed = int(data[7])

        player.longestStreak = int(data[8])

        player.largestWinMargin = float(data[9])

        player.lettersTyped = int(data[10])

        player.lettersTypedCorrectly = int(data[11])

        player.sumOfOpponentsElo = int(data[12])

        player.currentWinstreak = int(data[13])

        self.Close()

        return data

    def SaveUser(self, player : Player):

        self.Open()

        wordsTyped = player.wordsTyped

        timePlayed = player.timePlayed

        Elo = player.Elo

        highestElo = player.highestElo

        gamesWon = player.gamesWon

        gamesPlayed = player.gamesPlayed

        longestStreak = player.longestStreak

        largestWinMargin = player.largestWinMargin

        lettersTyped = player.lettersTyped

        lettersTypedCorrectly = player.lettersTypedCorrectly

        sumOfOpponentsElo = player.sumOfOpponentsElo

        currentWinstreak = player.currentWinstreak

        params = (wordsTyped, timePlayed, Elo, highestElo, gamesWon, gamesPlayed, longestStreak, largestWinMargin, lettersTyped, lettersTypedCorrectly, sumOfOpponentsElo, currentWinstreak, player.username)

        command = """

        UPDATE Users

        SET WordsTyped = (?),

            TimePlayed = (?),

            Elo = (?),

            HighestElo = (?),

            GamesWon = (?),

            GamesPlayed = (?),

            LongestStreak = (?),

            LargestWinMargin = (?),

            LettersTyped = (?),

            LettersTypedCorrectly = (?),

            SumOfOpponentsELo = (?),

            CurrentWinstreak = (?)

        WHERE Username = (?)

        """

        self.\_\_cursor.execute(command, params)

        self.Close()

    def CreateNewUser(self, username, password):

        self.Open()

        password = GenHash(password)

        wordsTyped = 0

        timePlayed = 0

        Elo = 1000

        highestElo = 1000

        gamesWon = 0

        gamesPlayed = 0

        longestStreak = 0

        largestWinMargin = 0

        lettersTyped = 0

        lettersTypedCorrectly = 0

        sumOfOpponentsElo = 0

        currentWinstreak = 0

        params = (str(username), password, wordsTyped, timePlayed, Elo, highestElo, gamesWon, gamesPlayed, longestStreak, largestWinMargin, lettersTyped, lettersTypedCorrectly, sumOfOpponentsElo, currentWinstreak)

        command = """

        INSERT INTO Users

        VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?)

        """

        self.\_\_cursor.execute(command, params)

        self.Close()

*#!Used for testing*

    def CreateRandomUser(self, username, password):

        self.Open()

        password = GenHash(password)

        wordsTyped = random.randint(0,1000000)

        timePlayed = random.randint(0,999999999)

        Elo = random.randint(500, 4000)

        highestElo = random.randint(Elo, Elo + 200)

        gamesPlayed = random.randint(0, 100000)

        gamesWon = random.randint(0, gamesPlayed)

        longestStreak = random.randint(0, gamesWon)

        largestWinMargin = random.randint(0, 100)

        lettersTyped = random.randint(0, 99999999999999)

        lettersTypedCorrectly = random.randint(0, lettersTyped)

        sumOfOpponentsELo = random.randint(0, 9999999999999999)

        currentWinstreak = random.randint(0, longestStreak)

        params = (username, password, wordsTyped, timePlayed, Elo, highestElo, gamesWon, gamesPlayed, longestStreak, largestWinMargin, lettersTyped, lettersTypedCorrectly, sumOfOpponentsELo, currentWinstreak)

        command = """

        INSERT INTO Users

        VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?)

        """

        self.\_\_cursor.execute(command, params)

        self.Close()

    def CheckIfUsernameInDB(self, username):

        self.Open()

        params = (username,)

        command = """SELECT \* FROM Users WHERE Username = ?"""

        self.\_\_cursor.execute(command, params)

        fetchResults = self.\_\_cursor.fetchall()

        self.Close()

        if fetchResults == []:

            return False

        else:

            return True

    def CheckPassword(self, username, password):

        self.Open()

        params = (username,)

        command = """

        SELECT Password FROM Users

        WHERE Username = (?)

        """

        self.\_\_cursor.execute(command, params)

        fetchResult = self.\_\_cursor.fetchall()

        correctPasswordHash = fetchResult[0][0]

        self.Close()

*#Uses bcrypt library to check password against hash*

        return CheckPW(password, correctPasswordHash)

#### Testing

//Todo

## Game

The game object is needed to handle the messages for synchronising the players of the game.

I will explain each section later in this document and I will also repeat the code I have pasted here there.

### Pseudocode

CLASS Game

SUBROUTINE init(self, player1, player2)

self.started <-- False

self.player1 <-- player1

self.player2 <-- player2

self.\_\_dbHandler <-- None

self.\_\_gameThread <-- Thread object

self.\_\_timerStage <-- None

self.\_\_raceStage <-- None

ENDSUBROUTINE

SUBROUTINE \_\_Run(self)

self.\_\_dbHandler <-- DatabaseHandler()

self.\_\_timerStage <-- TimerStage(self.player1, self.player2)

WHILE NOT self.\_\_timerStage.timerFinished

self.\_\_timerStage.main()

ENDWHILE

textForPlayersToType <-- self.\_\_timerStage.textForPlayersToType

self.\_\_raceStage <-- Race(self.player1, self.player2, textForPlayersToType, self.\_\_dbHandler)

WHILE NOT self.\_\_raceStage.raceFinished

self.\_\_raceStage.main()

ENDWHILE

ENDSUBROUTINE

ENDCLASS

### Python code

import pygame

import threading

from WordGeneration import WordGenerator

*#For type hints*

from Player import Player

from Database import DatabaseHandler

class Game:

    def \_\_init\_\_(self, player1 : Player, player2 : Player):

        self.started = False

        self.player1 = player1

        self.player2 = player2

        self.\_\_dbHandler : DatabaseHandler = None

        self.\_\_gameThread = threading.Thread(target=self.\_\_Run)

        self.\_\_gameThread.start()

        self.\_\_timerStage : TimerStage = None

        self.\_\_raceStage : Race  = None

    def \_\_Run(self):

*#Databasehandler creates SQLITE object that can only be used in thread it was created in*

        self.\_\_dbHandler = DatabaseHandler()

        self.\_\_timerStage = TimerStage(self.player1, self.player2)

        while not self.\_\_timerStage.timerFinished:

            self.\_\_timerStage.main()

        textForPlayersToType = self.\_\_timerStage.textForPlayersToType

        self.\_\_raceStage = Race(self.player1, self.player2, textForPlayersToType, self.\_\_dbHandler)

*#Runs racestage when the race is ongoing*

        while not self.\_\_raceStage.raceFinished:

            self.\_\_raceStage.main()

## Stage

The Game object uses Stage objects similar to how the Game object on the client side uses Scenes.

### Pseudocode

CLASS Stage

SUBROUTINE (self, player1, player2)

self.\_clock <-- pygame Clock object

self.\_player1 <-- player1

self.\_player2 <-- player2

self.\_players <-- [self.\_player1, self.\_player2]

ENDSUBROUTINE

SUBROUTINE main(self)

self.\_clock.tick()

self.\_HandleMessages()

ENDSUBROUTINE

SUBROUTINE \_HandleMessages(self)

FOR player IN self.\_players

unusedMessages <-- []

WHILE player.msgsReceived.GetLength() <> 0

message <-- player.msgsReceived.Dequeue()

IF message = "!DISCONNECT" THEN

player.playerQuit <-- True

ELSE

unusedMessages.APPEND(message)

ENDIF

ENDWHILE

FOR message IN unusedMessages

player.msgsReceived.Enqueue(message)

ENDFOR

ENDFOR

ENDSUBROUTINE

SUBROUTINE \_SendMessageToBothPlayers(self, messageToSend)

self.\_player1.msgsToSend.Enqueue(messageToSend)

self.\_player2.msgsToSend.Enqueue(messageToSend)

ENDSUBROUTINE

ENDCLASS

### Python code

class Stage:

    def \_\_init\_\_(self, player1 : Player, player2 : Player) -> None:

        self.\_clock = pygame.time.Clock()

        self.\_player1 = player1

        self.\_player2 = player2

        self.\_players = [self.\_player1, self.\_player2]

    def main(self):

        self.\_clock.tick()

        self.\_HandleMessages()

    def \_HandleMessages(self):

        for player in self.\_players:

            unusedMessages = []

            while player.msgsReceived.GetLength() != 0:

                message = player.msgsReceived.Dequeue()

                if message == "!DISCONNECT":

                    player.playerQuit = True

                else:

                    unusedMessages.append(message)

            for message in unusedMessages:

                player.msgsReceived.Enqueue(message)

    def \_SendMessageToBothPlayers(self, messageToSend):

        self.\_player1.msgsToSend.Enqueue(messageToSend)

        self.\_player2.msgsToSend.Enqueue(messageToSend)

### Timer

This is the stage that the player will be in when the timer is happening before the race starts. It will simply send a message to the user every second of how many seconds are left until the game starts.

It takes a while for the words to generate, and it is intended that the timer stays at 5 for a few seconds while it is generated. Afterwards the players will see the timer decrease at the same time.

#### Pseudocode

CLASS TimerStage(Stage)

SUBROUTINE init(self, player1, player2)

Inherit from Stage

self.timerFinished <-- False

#Attribute for timer in seconds

self.timeUntilStart <-- 5

self.\_\_timeSinceLastMessage <-- 0

self.\_SendMessageToBothPlayers("!STARTTIMER")

self.\_SendMessageToBothPlayers("!TIMELEFT:" + INT\_TO\_STRING(self.timeUntilStart))

#Word generation

self.\_\_wordGenerator <-- WordGenerator object

self.textForPlayersToType <-- self.\_\_wordGenerator.GetWordsForProgram(500)

ENDSUBROUTINE

SUBROUTINE main(self)

Inherit from Stage

self.\_\_timeSinceLastMessage <-- self.\_\_timeSinceLastMessage + time since last frame

IF self.timeUntilStart = 0 THEN

self.timerFinished <-- True

self.\_SendMessageToBothPlayers("!TEXTTOWRITE:" + self.textForPlayersToType)

ELSE IF self.\_\_timeSinceLastMessage >= 1000 THEN

self.\_\_timeSinceLastMessage <-- self.\_\_timeSinceLastMessage - 1000

self.timeUntilStart <-- self.timeUntilStart - 1

self.\_SendMessageToBothPlayers("!TIMELEFT:" + INT\_TO\_STRING(self.timeUntilStart))

ENDIF

ENDSUBROUTINE

ENDCLASS

#### Python code

*#Stage for when players are waiting for game to start*

class TimerStage(Stage):

    def \_\_init\_\_(self, player1 : Player, player2 : Player) -> None:

        super().\_\_init\_\_(player1, player2)

        self.timerFinished = False

*#Attribute for timer in seconds*

        self.timeUntilStart = 5

        self.\_\_timeSinceLastMessage = 0

        self.\_SendMessageToBothPlayers("!STARTTIMER")

        self.\_SendMessageToBothPlayers(f"!TIMELEFT:{self.timeUntilStart}")

*#Word generation*

        self.\_\_wordGenerator = WordGenerator()

        self.textForPlayersToType = self.\_\_wordGenerator.GetWordsForProgram(500)

        self.\_clock.tick()

*#Runs every frame in Game.main() while timerfinished is false*

    def main(self):

        super().main()

        self.\_\_timeSinceLastMessage += self.\_clock.get\_time()

        if self.timeUntilStart == 0:

            self.timerFinished = True

            self.\_SendMessageToBothPlayers(f"!TEXTTOWRITE:{self.textForPlayersToType}")

        elif self.\_\_timeSinceLastMessage >= 1000:

            self.\_\_timeSinceLastMessage -= 1000

            self.timeUntilStart -= 1

            self.\_SendMessageToBothPlayers(f"!TIMELEFT:{self.timeUntilStart}")

### Race

The race stage is the stage where the players are racing against each other. The server needs to send the player’s text to their opponent and vice versa. It also needs to update all the stats that are stored about the user.

#### Pseudocode

CLASS Race(Stage)

SUBROUTINE (self, player1, player2, textPlayersHaveToType, databaseHandler)

Inherit from Stage

self.raceFinished <-- False

self.\_\_waitingForPlayersText <-- False

self.timeUntilEnd <-- 30

self.\_\_timeSinceLastTimerUpdate <-- 0

self.\_\_textPlayersHaveToType <-- textPlayersHaveToType

self.\_\_player1FinalText <-- None

self.\_\_player2FinalText <-- None

self.\_\_player1TimeFinished <-- None

self.\_\_player2TimeFinished <-- None

self.\_\_dbHandler <-- databaseHandler

ENDSUBROUTINE

SUBROUTINE main(self)

Inherit from Stage

self.\_\_timeSinceLastTimerUpdate <-- self.\_\_timeSinceLastTimerUpdate + time since last frame

IF self.\_\_timeSinceLastTimerUpdate >= 1000 AND NOT self.\_\_waitingForPlayersText THEN

self.timeUntilEnd <-- self.timeUntilEnd - 1

self.\_\_timeSinceLastTimerUpdate <-- self.\_\_timeSinceLastTimerUpdate - 1000

self.\_SendMessageToBothPlayers("!TIMELEFT:" + STRING\_TO\_INT(self.timeUntilEnd))

IF self.timeUntilEnd = 0 THEN

self.\_SendMessageToBothPlayers("!GAMECOMPLETE")

self.\_\_waitingForPlayersText = True

ENDIF

ENDIF

FOR player IN self.\_players

IF LEN(player.textWritten) = LEN(self.\_\_textPlayersHaveToType) THEN

OUTPUT "A player has won"

IF self.\_\_player1FinalText IS NOT None AND self.\_\_player2FinalText IS NOT None THEN

self.\_SendMessageToBothPlayers("!GAMECOMPLETED")

self.raceFinished <-- True

self.\_\_UpdatePlayers()

ENDIF

ENDSUBROUTINE

#This method calculates the players' new stats

SUBROUTINE \_\_UpdatePlayers(self)

self.\_player1.wordsTyped <-- self.\_player1.wordsTyped + LEN(self.\_\_player1FinalText.split(" "))

self.\_player2.wordsTyped <-- self.\_player2.wordsTyped + LEN(self.\_\_player2FinalText.split(" "))

self.\_player1.timePlayed <-- self.\_player1.timePlayed + self.\_\_player1TimeFinished

self.\_player2.timePlayed <-- self.\_player2.timePlayed + self.\_\_player2TimeFinished

#Find number of letters each player got correct

player1LettersCorrect <-- 0

FOR i IN RANGE(LEN(self.\_\_player1FinalText))

self.\_player1.lettersTyped <-- self.\_player1.lettersTyped + 1

IF self.\_\_player1FinalText[i] = self.\_\_textPlayersHaveToType[i] THEN

player1LettersCorrect <-- player1LettersCorrect + 1

self.\_player1.lettersTypedCorrectly <-- self.\_player1.lettersTypedCorrectly + 1

ENDIF

ENDFOR

player2LettersCorrect <-- 0

FOR i IN RANGE(LEN(self.\_\_player2FinalText))

self.\_player2.lettersTyped <-- self.\_player2.lettersTyped + 1

IF self.\_\_player2FinalText[i] = self.\_\_textPlayersHaveToType[i] THEN

player2LettersCorrect <-- player2LettersCorrect + 1

self.\_player2.lettersTypedCorrectly <-- self.\_player2.lettersTypedCorrectly + 1

ENDIF

ENDFOR

#Determine winner

IF player1LettersCorrect = player2LettersCorrect THEN

IF self.\_\_player1TimeFinished = self.\_\_player2TimeFinished THEN

#Game is a draw

winner <-- None

loser <-- None

ELSE IF self.\_\_player1TimeFinished < self.\_\_player2TimeFinished THEN

winner <-- self.\_player1

loser <-- self.\_player2

ELSE

winner <-- self.\_player2

loser <-- self.\_player2

ELSE IF player1LettersCorrect > player2LettersCorrect THEN

winner <-- self.\_player1

winMargin <-- player1LettersCorrect - player2LettersCorrect

if winMargin > winner.largestWinMargin:

winner.largestWinMargin <-- winMargin

loser <-- self.\_player2

ELSE

winner <-- self.\_player2

winMargin <-- player2LettersCorrect - player1LettersCorrect

IF winMargin > winner.largestWinMargin THEN

winner.largestWinMargin <-- winMargin

ENDIF

loser <-- self.\_player1

ENDIF

IF winner IS NOT None THEN

winner.msgsToSend.Enqueue("!MATCHOUTCOME:WIN")

loser.msgsToSend.Enqueue("!MATCHOUTCOME:LOSS")

winner.msgsToSend.Enqueue("!MARGIN:" + winMargin)

loser.msgsToSend.Enqueue("!MARGIN:" + winMargin)

winner.gamesWon <-- winner.gamesWon + 1

winner.gamesPlayed <-- winner.gamesPlayed + 1

winner.currentWinstreak <-- winner.currentWinstreak + 1

IF winner.currentWinstreak > winner.longestStreak THEN

winner.longestStreak <-- winner.currentWinstreak

loser.gamesPlayed <-- loser.gamesPlayed + 1

loser.currentWinstreak <-- 0

winner.sumOfOpponentsElo <-- winner.sumOfOpponentsElo + loser.Elo

loser.sumOfOpponentsElo <-- loser.sumOfOpponentsElo + winner.Elo

winnerGamesLost <-- winner.gamesPlayed - winner.gamesWon

#This is the formula for calculating PR (Performance Rating) increase

EloDiff <-- (winner.sumOfOpponentsElo + 400 \* (winner.gamesWon - winnerGamesLost)) / winner.gamesPlayed - winner.Elo

winner.msgsToSend.Enqueue("!ELO:" + EloDiff)

winner.Elo <-- winner.Elo EloDiff

IF winner.Elo > winner.highestElo THEN

winner.highestElo <-- winner.Elo

ENDIF

loserGamesLost <-- loser.gamesPlayed - loser.gamesWon

EloDiff <-- loser.Elo - (loser.sumOfOpponentsElo + 400 \* (loser.gamesWon - loserGamesLost)) / loser.gamesPlayed

loser.msgsToSend.Enqueue("!ELO:" + EloDiff)

loser.Elo <-- loser.Elo - EloDiff

ELSE

self.\_SendMessageToBothPlayers("!MATCHOUTCOME:DRAW")

ENDIF

#Updates database

self.\_\_dbHandler.SaveUser(self.\_player1)

self.\_\_dbHandler.SaveUser(self.\_player2)

self.\_player1.gameFinished <-- True

self.\_player2.gameFinished <-- True

ENDSUBROUTINE

SUBROUTINE \_HandleMessages(self)

Inherit from Stage

unusedMessages <-- []

WHILE self.\_player1.msgsReceived.GetLength() <> 0

message <-- self.\_player1.msgsReceived.Dequeue()

IF message[:6] = "!TEXT:" THEN

text <-- message[6:]

self.\_player1.textWritten <-- text

self.\_player2.msgsToSend.Enqueue("!OTHERPLAYERTEXT:" + self.\_player1.textWritten)

ELSE IF message[:11] = "!FINALTEXT:" THEN

self.\_\_player1FinalText <-- message[11:]

self.\_\_player1TimeFinished <-- 30 - self.timeUntilEnd

ELSE:

unusedMessages.APPEND(message)

ENDIF

ENDWHILE

FOR message IN unusedMessages

self.\_player1.msgsReceived.Enqueue(message)

ENDFOR

#For player2

unusedMessages <-- []

WHILE self.\_player2.msgsReceived.GetLength() <> 0

message = self.\_player2.msgsReceived.Dequeue()

IF message[:6] = "!TEXT:" THEN

text <-- message[6:]

self.\_player2.textWritten <-- text

self.\_player1.msgsToSend.Enqueue("!OTHERPLAYERTEXT:" + self.\_player2.textWritten)

ELSE IF message[:11] = "!FINALTEXT:" THEN

self.\_\_player2FinalText <-- message[11:]

self.\_\_player2TimeFinished <-- 30 - self.timeUntilEnd

ELSE:

unusedMessages.APPEND(message)

ENDIF

ENDWHILE

FOR message IN unusedMessages:

self.\_player2.msgsReceived.Enqueue(message)

ENDFOR

ENDSUBROUTINE

ENDCLASS

#### Python code

class Race(Stage):

    def \_\_init\_\_(self, player1: Player, player2: Player, textPlayersHaveToType, databaseHandler : DatabaseHandler) -> None:

        super().\_\_init\_\_(player1, player2)

        self.raceFinished = False

        self.\_\_waitingForPlayersText = False

        self.timeUntilEnd = 30

        self.\_\_timeSinceLastTimerUpdate = 0

        self.\_\_textPlayersHaveToType = textPlayersHaveToType

        self.\_\_player1FinalText : str = None

        self.\_\_player2FinalText : str = None

        self.\_\_player1TimeFinished : int = None

        self.\_\_player2TimeFinished : int = None

        self.\_\_dbHandler = databaseHandler

    def main(self):

        super().main()

        self.\_\_timeSinceLastTimerUpdate += self.\_clock.get\_time()

        if self.\_\_timeSinceLastTimerUpdate >= 1000 and not self.\_\_waitingForPlayersText:

            self.timeUntilEnd -= 1

            self.\_\_timeSinceLastTimerUpdate -= 1000

            self.\_SendMessageToBothPlayers(f"!TIMELEFT:{self.timeUntilEnd}")

            if self.timeUntilEnd == 0:

                self.\_SendMessageToBothPlayers("!GAMECOMPLETE")

                self.\_\_waitingForPlayersText = True

*#Checking if player has finished*

        for player in self.\_players:

            if len(player.textWritten) == len(self.\_\_textPlayersHaveToType):

                print("A player has won")

        if self.\_\_player1FinalText is not None and self.\_\_player2FinalText is not None:

            self.\_SendMessageToBothPlayers("!GAMECOMPLETED")

            self.raceFinished = True

            self.\_\_UpdatePlayers()

    def \_\_UpdatePlayers(self):

*#For player1*

*#Calculate words typed*

        self.\_player1.wordsTyped += len(self.\_\_player1FinalText.split(" "))

        self.\_player2.wordsTyped += len(self.\_\_player2FinalText.split(" "))

        self.\_player1.timePlayed += self.\_\_player1TimeFinished

        self.\_player2.timePlayed += self.\_\_player2TimeFinished

*#Find number of letters each player got correct*

        player1LettersCorrect = 0

        for i in range(len(self.\_\_player1FinalText)):

            self.\_player1.lettersTyped += 1

            if self.\_\_player1FinalText[i] == self.\_\_textPlayersHaveToType[i]:

                player1LettersCorrect += 1

                self.\_player1.lettersTypedCorrectly += 1

        player2LettersCorrect = 0

        for i in range(len(self.\_\_player2FinalText)):

            self.\_player2.lettersTyped += 1

            if self.\_\_player2FinalText[i] == self.\_\_textPlayersHaveToType[i]:

                player2LettersCorrect += 1

                self.\_player2.lettersTypedCorrectly += 1

*#Determine winner*

*#If both players wrote same number of letters correctly*

        if player1LettersCorrect == player2LettersCorrect:

            if self.\_\_player1TimeFinished == self.\_\_player2TimeFinished:

*#Game is a draw*

                winner = None

                loser = None

            elif self.\_\_player1TimeFinished < self.\_\_player2TimeFinished:

                winner = self.\_player1

                loser = self.\_player2

            else:

                winner = self.\_player2

                loser = self.\_player2

        elif player1LettersCorrect > player2LettersCorrect:

            winner = self.\_player1

            winMargin = player1LettersCorrect - player2LettersCorrect

            if winMargin > winner.largestWinMargin:

                winner.largestWinMargin = winMargin

            loser = self.\_player2

        else:

            winner = self.\_player2

            winMargin = player2LettersCorrect - player1LettersCorrect

            if winMargin > winner.largestWinMargin:

                winner.largestWinMargin = winMargin

            loser = self.\_player1

        if winner is not None:

            winner.msgsToSend.Enqueue("!MATCHOUTCOME:WIN")

            loser.msgsToSend.Enqueue("!MATCHOUTCOME:LOSS")

            winner.msgsToSend.Enqueue(f"!MARGIN:{winMargin}")

            loser.msgsToSend.Enqueue(f"!MARGIN:{winMargin}")

            winner.gamesWon += 1

            winner.gamesPlayed += 1

            winner.currentWinstreak += 1

            if winner.currentWinstreak > winner.longestStreak:

                winner.longestStreak = winner.currentWinstreak

            loser.gamesPlayed += 1

            loser.currentWinstreak = 0

            winner.sumOfOpponentsElo += loser.Elo

            loser.sumOfOpponentsElo += winner.Elo

            winnerGamesLost = winner.gamesPlayed - winner.gamesWon

            EloDiff = (winner.sumOfOpponentsElo + 400 \* (winner.gamesWon - winnerGamesLost)) / winner.gamesPlayed - winner.Elo

            winner.msgsToSend.Enqueue(f"!ELO:{EloDiff}")

            winner.Elo += EloDiff

            if winner.Elo > winner.highestElo:

                winner.highestElo = winner.Elo

            loserGamesLost = loser.gamesPlayed - loser.gamesWon

            EloDiff = loser.Elo - (loser.sumOfOpponentsElo + 400 \* (loser.gamesWon - loserGamesLost)) / loser.gamesPlayed

            loser.msgsToSend.Enqueue(f"!ELO:{EloDiff}")

            loser.Elo -= EloDiff

        else:

            self.\_SendMessageToBothPlayers("!MATCHOUTCOME:DRAW")

*#Updates database*

        self.\_\_dbHandler.SaveUser(self.\_player1)

        self.\_\_dbHandler.SaveUser(self.\_player2)

        self.\_player1.gameFinished = True

        self.\_player2.gameFinished = True

    def \_HandleMessages(self):

        super().\_HandleMessages()

*#For player1*

        unusedMessages = []

        while self.\_player1.msgsReceived.GetLength() != 0:

            message = self.\_player1.msgsReceived.Dequeue()

            if message[:6] == "!TEXT:":

                text = message[6:]

                self.\_player1.textWritten = text

                self.\_player2.msgsToSend.Enqueue(f"!OTHERPLAYERTEXT:{self.\_player1.textWritten}")

            elif message[:11] == "!FINALTEXT:":

                self.\_\_player1FinalText = message[11:]

                self.\_\_player1TimeFinished = 30 - self.timeUntilEnd

            else:

                unusedMessages.append(message)

        for message in unusedMessages:

            self.\_player1.msgsReceived.Enqueue(message)

*#For player2*

        unusedMessages = []

        while self.\_player2.msgsReceived.GetLength() != 0:

            message = self.\_player2.msgsReceived.Dequeue()

            if message[:6] == "!TEXT:":

                text = message[6:]

                self.\_player2.textWritten = text

                self.\_player1.msgsToSend.Enqueue(f"!OTHERPLAYERTEXT:{self.\_player2.textWritten}")

            elif message[:11] == "!FINALTEXT:":

                self.\_\_player2FinalText = message[11:]

                self.\_\_player2TimeFinished = 30 - self.timeUntilEnd

            else:

                unusedMessages.append(message)

        for message in unusedMessages:

            self.\_player2.msgsReceived.Enqueue(message)

# Data structures

## Priority queue

Initially I made the InputHandler object use a priority queue but I later reverted that as it was unnecessary and had a negative impact on performance. With a priority queue you can only dequeue from the start, which meant any unused inputs would have to be sent to the enqueue method and that would search through where to put it in the list. This would all iterate through the list an unnecessary amount of time, while with the new solution I could simply have an index – i – and use that for checking each input and popping the input once it was used.

### Python code

class PriorityQueue:

    def \_\_init\_\_(self):

*#List where the data is stored [priority, item]*

*#Private as it shouldn't be accessed outside of the queue*

        self.\_\_queue = []

*#Adds item to the correct point in the queue*

    def Enqueue(self, priority, itemToAdd):

*#If list is empty appends it to the start of the list*

        if self.\_\_queue == []:

            self.\_\_queue.append([priority, itemToAdd])

            return 0

*#items in self.\_\_queue are lists [priority, actual item]*

        for i in range(len(self.\_\_queue)):

*#Goes through each item in the list*

*#Checks if the priority is lower than the priority of the new item*

            if priority > self.\_\_queue[i][0]:

*#Splits string where item needs to go*

                startOfQueue = self.\_\_queue[:i]

                endOfQueue = self.\_\_queue[i:]

*#Joins the lists together*

                self.\_\_queue = startOfQueue + [[priority, itemToAdd]] + endOfQueue

                break

            elif i == len(self.\_\_queue) - 1:

                self.\_\_queue.append([priority, itemToAdd])

*#Removes item from the start of the queue*

    def Dequeue(self):

        if self.\_\_queue != []:

            return self.\_\_queue.pop(0)

        else:

            return []

    def GetLength(self):

        return len(self.\_\_queue)

## Queue

The final implementation does still use a queue however for the player’s messages in the server section. This is useful because messages can only be taken from the front of the queue and added to the end. This could cause issues with many players playing the game at once on a weak server but that could be later optimised for better performance to change the queue as few times as possible.

### Python code

class Queue:

    def \_\_init\_\_(self) -> None:

        self.\_\_list = []

    def Enqueue(self, itemToAdd):

        self.\_\_list.append(itemToAdd)

    def Dequeue(self):

        return self.\_\_list.pop(0)

    def GetLength(self):

        return len(self.\_\_list)

# Searching algorithm

I required a linear search for the removal of players from the players queues in the server.

## Python code

def LinearSearch(itemSought, list):

    i = 0

    while i < len(list):

        if list[i] == itemSought:

            return i

        i += 1

    return None

# Class Box

This box class needs to be able to hold coordinates, states and other information about a textbox on the client’s side.

This will later be inherited by InputBox and TextBox as they are quite similar to prevent duplicate code.

These boxes will be used in Scenes and will be changed by inputs which are handled in those scenes.

## Pseudocode

CLASS BOX

SUBROUTINE init(self, rectangleObject, font, resolution, colourActive, colourInactive, textColour, text = "")

self.rect <-- rectangleObject

self.font <-- font

self.resolution <-- resolution

self.activeColour <-- colourActive

self.inactiveColour <-- colourInactive

self.isActive <-- False

self.\_colour <-- colourInactive

self.text <-- text

self.textColour <-- textColour

ENDSUBROUTINE

SUBROUTINE UpdateRender(self)

Do nothing since this is just for inheritance

ENDSUBROUTINE

SUBROUTINE Render(self, window)

Draw the rectangle object with self.colour

ENDSUBROUTINE

SUBROUTINE AddLetter(self, letter)

Concatenate self.text and letter

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE RemoveLetter(self, control)

IF self.text <> "" THEN

IF control THEN

Remove trailing spaces in self.text

Remove letters until the last letter is (not "") or " "

ELSE

Remove a letter from self.text

ENDIF

ENDIF

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE SetActive(self)

self.isActive <-- True

self.colour <-- self.activeColour

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE SetInactive(self)

self.isActive <-- False

self.colour <-- self.inactiveColour

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE CheckForCollisionWithMouse(self, mouseLocation)

IF mouseLocation is in self.rect THEN

RETURN True

ENDIF

RETURN False

ENDSUBROUTINE

ENDCLASS

## Python code

import pygame, math

*#A class box that is used for textboxes and inputboxes*

class Box:

    def \_\_init\_\_(self, rect, font, resolution, colourActive, colourInactive, textColour, text = "") -> None:

        self.\_rect = rect

        self.\_font = font

        self.\_resolution = resolution

        self.\_activeColour = colourActive

        self.\_inactiveColour = colourInactive

*#Colour starts inactive*

        self.isActive = False

        self.\_colour = colourInactive

        self.text = text

        self.textColour = textColour

    def UpdateRender(self):

        pass

    def Render(self, window):

        pygame.draw.rect(window, self.\_colour, self.\_rect)

    def AddLetter(self, letter):

        self.text += letter

        self.UpdateRender()

    def RemoveLetter(self, control):

        if self.text != "":

            if control:

*#Removes trailing spaces*

                while self.text[-1] == " ":

                    self.text = self.text[:-1]

*#Removes letters until a space is reached or text has run out*

                while self.text != "" and self.text[-1] != " ":

                    self.text = self.text[:-1]

            else:

                self.text = self.text[:-1]

        self.UpdateRender()

    def SetActive(self):

        self.isActive = True

        self.\_colour = self.\_activeColour

        self.UpdateRender()

    def SetInactive(self):

        self.isActive = False

        self.\_colour = self.\_inactiveColour

        self.UpdateRender()

    def CheckForCollisionWithMouse(self, mouseLocation):

        if self.\_rect.collidepoint(mouseLocation):

            return True

        return False

## Input Box(Box)

This class needs to be able to make a box, render it on screen and show the text. This will be done by inheriting the box class which I made which gives a good foundation for boxes that need text in them.

The Input box is used for getting text input from the user, usually the box will be able to be selected and typed in and the user will be able to press a button to submit their input. This will be handled in the Scene object which will detect when the user has pressed a specific button and will then do whatever is needed for that specific button. The InputBox text attribute will be public and will be checked by the Scene object.

An example would be that in the login screen it will ask the user for a username and password. When the button is pressed the username and password InputBox’s text attribute will sent to the server to be validated against the data in the database.

### Pseudocode

CLASS InputBox(Box)

SUBROUTINE init(self, rectangleObject, font, resolution, colourActive, colourInactive, textColour, text, hashed = False)

Inherit from Box

self.hashed <-- hashed

self.textLocation <-- None

self.textRender <-- None

ENDSUBROUTINE

SUBROUTINE Render(self, window)

Inherit from Box

Draw self.textRender on window at self.textLocation

ENDSUBROUTINE

SUBROUTINE UpdateRender(self)

IF self.hashed THEN

textToRender <-- LEN(self.text) \* "\*"

ELSE

textToRender <-- self.text

ENDIF

textSize <-- size of textToRender when rendered with self.font[0]

characterWidth <-- textSize / LEN(self.textToRender)

spaceAvailable <-- width of self.rect - 10

IF spaceAvailable < textSize THEN

charactersToRemove <-- (textSize - spaceAvailable) / characterWidth

Remove charactersToRemove number of letters from the front of textToRender

ENDIF

self.textRender <-- render textToRender with self.textColour

self.textLocation <-- self.rect's x + 5 and y + 10

ENDSUBROUTINE

ENDCLASS

### Python code

class InputBox(Box):

    def \_\_init\_\_(self, rect, font, resolution, colourActive, colourInactive, textColour, text, hashed = False) -> None:

        super().\_\_init\_\_(rect, font, resolution, colourActive, colourInactive, textColour, text)

        self.\_\_hashed = hashed

        self.\_\_textLocation = None

        self.\_\_textRender = None

        self.UpdateRender()

    def Render(self, window):

        super().Render(window)

        window.blit(self.\_\_textRender, self.\_\_textLocation)

    def UpdateRender(self):

        if self.\_\_hashed:

*#Makes hashed string*

            textToRender = len(self.text) \* "\*"

        else:

            textToRender = self.text

        textSize = self.\_font.size(textToRender)

*#Makes text correct size*

        while textSize[0] > self.\_rect.width - 10:

            textToRender = textToRender[1:]

            textSize = self.\_font.size(textToRender)

        self.\_\_textRender = self.\_font.render(textToRender, True, self.textColour)

        self.\_\_textLocation = (self.\_rect.left + 5 \* self.\_resolution[0], self.\_rect.top + 10 \* self.\_resolution[1])

### Testing code:

For the purposes of testing I made a simple gameloop that will handle the expected inputs for the InputBox and will also update the display.

import pygame

from Boxes import InputBox

dispWidth = 1920

dispHeight = 1080

window = pygame.display.set\_mode((dispWidth, dispHeight))

pygame.display.set\_caption("SpeedTyper")

pygame.font.init()

boxWidth = 1000

boxHeight = 100

*#Boxlocations*

rectangleUsernameBox = pygame.rect.Rect((dispWidth - boxWidth) / 2, 400 \* dispHeight / 1080, boxWidth, boxHeight)

usernameBox = InputBox((255,255,255), (35,35,35), (30,30,30), False, rectangleUsernameBox, dispHeight)

rectanglePasswordBox = pygame.rect.Rect((dispWidth - boxWidth) / 2, 600 \* dispHeight / 1080, boxWidth, boxHeight)

passwordBox = InputBox((255,255,255), (35,35,35), (30,30,30), True, rectanglePasswordBox, dispHeight)

running = True

backspace = False

timeSinceLastBackspace = 0

timeBetweenBackspaces = 50

usernameBoxSelected = False

passWordBoxSelected = False

control = False

clock = pygame.time.Clock()

while running:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            running = False

        elif event.type == pygame.MOUSEBUTTONDOWN:

            clickLocation = pygame.mouse.get\_pos()

            if usernameBox.CheckIfClicked(clickLocation):

                usernameBox.SetActive()

                usernameBoxSelected = True

            else:

                usernameBox.SetDormant()

                usernameBoxSelected = False

            if passwordBox.CheckIfClicked(clickLocation):

                passwordBox.SetActive()

                passWordBoxSelected = True

            else:

                passwordBox.SetDormant()

                passWordBoxSelected = False

        elif event.type == pygame.KEYDOWN:

            if event.key == pygame.K\_BACKSPACE:

                backspace = True

                if usernameBoxSelected:

                    usernameBox.RemoveLetter(control)

                elif passWordBoxSelected:

                    passwordBox.RemoveLetter(control)

                timeSinceLastBackspace = -200

            elif event.key == pygame.K\_RETURN:

                pass

            elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:

                control = True

            else:

                if usernameBoxSelected:

                    usernameBox.AddLetter(event.unicode)

                elif passWordBoxSelected:

                    passwordBox.AddLetter(event.unicode)

        elif event.type == pygame.KEYUP:

            if event.key == pygame.K\_BACKSPACE:

                backspace = False

            elif event.key == pygame.K\_RETURN:

                pass

            elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:

                control = False

    if backspace and timeSinceLastBackspace > timeBetweenBackspaces:

        if usernameBoxSelected:

            usernameBox.RemoveLetter(control)

        elif passWordBoxSelected:

            passwordBox.RemoveLetter(control)

        timeSinceLastBackspace = 0

    clock.tick()

    timeSinceLastBackspace += clock.get\_time()

    window.fill((10,10,10))

    usernameBox.DrawBox(window)

    passwordBox.DrawBox(window)

    pygame.display.update()

This code will simply draw 2 boxes on screen of width

### Testing:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | A box is drawn on screen of the correct dimensions | I will run the program. | The box is drawn on screen of the correct dimensions. | Normal | Pass |
| 2 | The box can be selected by clicking on it and deselected by clicking away from it | I will run the program and click on the box. Then I will click outside the box. | The box should be selected and turn a different colour between when I click on it and outside it. | Normal | Pass |
| 3 | The box will only take input while it is selected | I will run the program, click on the first box and then I will start typing. I will then click outside and continue typing. | The box should accept input while it is selected and should ignore input when it is not selected. | Normal | Pass |
| 4 | The box will take input and display it correctly | I will run the program, click on the first box and then I will start typing. | The box should display whatever I type (except for non-unicode characters such as tab and shift). | Normal | Pass |
| 5 | The box should display only stars when written in the second box | I will run the program, click on the first box and then I will start typing. | The box should display n number of stars where n is the number of letters I have input. | Normal | Pass |
| 6 | The boxes will need to return the text that was typed into them | I will alter the program to print the username and password when the game is closed. | It should return whatever I typed into the boxes | Normal | Pass |

#### 1

Given that the code for the testing is this:

import pygame

from Boxes import InputBox

boxWidth = 1000

boxHeight = 100

*#Boxlocations*

rectangleUsernameBox = pygame.rect.Rect((dispWidth - boxWidth) / 2, 400 \* dispHeight / 1080, boxWidth, boxHeight)

usernameBox = InputBox((255,255,255), (35,35,35), (30,30,30), False, rectangleUsernameBox, dispHeight)

rectanglePasswordBox = pygame.rect.Rect((dispWidth - boxWidth) / 2, 600 \* dispHeight / 1080, boxWidth, boxHeight)

passwordBox = InputBox((255,255,255), (35,35,35), (30,30,30), True, rectanglePasswordBox, dispHeight)

The box needs to be 1000 pixels wide and 100 pixels tall. There are 2 boxes due to the code, however it does not matter as they are both the same size.

To test this, I will run the program and paste a screenshot.

Background pattern

Description automatically generated

This is clearly incorrect as the dimensions of the boxes are roughly 50% larger than they should be. This is due to the windows scale being an issue, which is used on high PPI (pixels per inch) displays such as on laptops.

Text

Description automatically generated

For me, it was on 150%. The way I combated this was importing ctypes and pasting the following line of code at the start:

import \_ctypes

ctypes.windll.user32.SetProcessDPIAware()

This made it so that it disabled the windows stretching and made the dimensions correct.

After adding this, I ran the program and took another screenshot.

Graphical user interface, text

Description automatically generated

These dimensions are now correct and so this test is now passed.

#### 2

The box needs to be able to be selected. This is done by clicking on the box, and so I will do this to test it.

Before clicking on the first box:

Graphical user interface, text

Description automatically generated

After clicking on the first box:

Text

Description automatically generated

While not super obvious, the colour has changed (easy to see by comparing the top box to the bottom box) and it is quite obvious when you click on the box and it changes colour. This test is passed.

#### 3

The box should ignore input when it is not selected and it should take input when it is selected. To test this I will run the program, type “I have not selected the box”, click on the box and type “I have selected the box”. It should only display “I have selected the box” after I have clicked on the box.

Before clicking on the box:

Graphical user interface, text

Description automatically generated

After clicking on the box and typing “I have selected the box”:

Graphical user interface, text

Description automatically generated

The text is cut off but that is intended.

The box only takes input when it is selected and so this test is passed.

#### 4

The box will need to display the user’s input. I will test this by running the program, clicking on the box and then typing “I have started typing”. The program should display what I have written in the textbox.

Graphical user interface, text

Description automatically generated

This test is passed.

#### 5

This object will be used for both the username and password box, and so the 2nd box needs to be starred out.

I will test this by typing 12345 and it should display \*\*\*\*\*

Graphical user interface, text

Description automatically generated

This is the expected result and so this test is passed.

#### 6

I will need to be able to get the text from these boxes back at some point to check if they’re the correct details entered, and so to test this I will need to add the following code:

while running:

    for event in pygame.event.get():

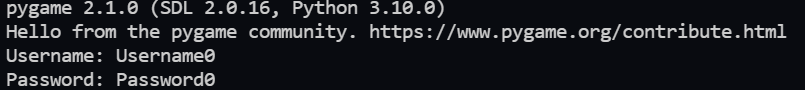
        if event.type == pygame.QUIT:

            running = False

            print(f"Username: {usernameBox.text}")

            print(f"Password: {passwordBox.text}")

I will run the program and enter Username0 into the first box and Password0 into the second box. This is the output:



This test is passed.

## TextBox(Box)

This has gone through many revisions because every way I came up with dealing with it had issues, but I ended up with this which hasn’t got any issues.

### Pseudocode

CLASS TextBox(Box)

SUBROUTINE init(self, rectangleObject, font, resolution, colourActive, colourInactive, textColour, previewText, previewTextColour, incorrectTextColour)

Inherit from Box

self.previewText <-- previewText

self.previewTextColour <-- previewTextColour

self.incorrectTextColour <-- incorrectTextColour

self.textRenderLocation <-- None

self.previewTextRender <-- None

self.correctTextRender <-- None

self.incorrectTextRender <--

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE SetText(self, newText)

self.text <-- newText

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE CheckIfFinished(self)

IF LEN(self.text) = LEN(self.previewText) THEN

RETURN True

ENDIF

RETURN False

ENDSUBROUTINE

SUBROUTINE Render(self, window)

Inherit from Box

Draw self.previewTexxtRender on window

Draw self.correctTextRender on window

Draw self.incorrectTextRender on window

ENDSUBROUTINE

SUBROUTINE UpdateRender(self)

text <-- self.text

previewText <-- self.previewText

#self.resolution is resolution scale and not actual pixels

spaceAvailable <-- FLOAT\_TO\_INT(self.rect.width - 10 \* self.resolution[0])

IF text <> "" THEN

textWidth <-- get size of text when it is rendered with self.font

characterWidth <-- textWidth / LEN(text)

IF textWidth > spaceAvailable / 2 THEN

spaceDifference <-- textWidth - spaceAvailable / 2

charactersToRemove <-- spaceDifference / characterWidth rounded UP to nearest integer

Remove charactersToRemove number of letters from the start of text

Remove charactersToRemove number of letters from the start of previewText

ENDIF

ENDIF

previewTextWidth <-- get size of previewText when it is rendered with self.font

characterWidth <-- previewTextWidth / LEN(previewText)

IF spaceAvailable < previewTextWidth THEN

spaceDifference <-- previewTextWidth - spaceAvailable

ENDIF

charactersToRemove <-- spaceDifference / characterWidth rounded UP to nearest integer

Remove charactersToRemove number of letters from end of previewText

correctText <-- []

incorrectText <-- []

FOR i <-- RANGE(LEN(text))

IF text[i] = previewText[i] THEN

add previewText[i] to correctText

add " " to incorrectText

ELSE

add " " to correctText

add previewText[i] to incorrectText

ENDIF

ENDFOR

Convert correctText to string from list

Convert incorrectText to string from list

self.textRenderLocation <-- textbox's x + 5 and textbox's y + 5

self.previewTextRender <-- render previewText with font and self.previewTextColour

self.correctTextRender <-- render correctText with font and self.textColour

self.incorrectTextRender <-- render incorrectText with font and self.incorrectTextColour

ENDSUBROUTINE

ENDCLASS

### Python code

*#Main box object for race*

class TextBox(Box):

    def \_\_init\_\_(self, rect, font, resolution, colourActive, colourInactive, textColour, previewText, previewTextColour, incorrectTextColour) -> None:

        super().\_\_init\_\_(rect, font, resolution, colourActive, colourInactive, textColour)

        self.previewText = previewText

        self.\_previewTextColour = previewTextColour

        self.\_incorrectTextColour = incorrectTextColour

        self.\_\_textRenderLocation = None

        self.\_\_previewTextRender = None

        self.\_\_correctTextRender = None

        self.\_\_incorrectTextRender = None

        self.UpdateRender()

    def SetText(self, newText):

        self.text = newText

        self.UpdateRender()

    def CheckIfFinished(self):

        if len(self.text) == len(self.previewText):

            return True

        return False

    def Render(self, window):

        super().Render(window)

*#Draws surface objects onto window*

        window.blit(self.\_\_previewTextRender, self.\_\_textRenderLocation)

        window.blit(self.\_\_correctTextRender, self.\_\_textRenderLocation)

        window.blit(self.\_\_incorrectTextRender, self.\_\_textRenderLocation)

    def UpdateRender(self):

*#Copies text so that it can be changed*

        text = self.text

        previewText = self.previewText

        spaceAvailable = int(self.\_rect.width - 10 \* self.\_resolution[0])

*#Prevents division by 0*

        if text != "":

*#Cuts text so that it is at most halfway through the box*

            textWidth = self.\_font.size(text)[0]

            characterWidth = textWidth / len(text)

*#Only does this if text is longer than the middle of the textbox*

            if textWidth > spaceAvailable / 2:

*#Divided by 2 as spaceAvailable is the entire box - 10 pixels*

                spaceDifference = textWidth - spaceAvailable / 2

                charactersToRemove = int(math.ceil(spaceDifference / characterWidth))

*#Removes enough characters from front so that it reaches the middle*

                text = text[charactersToRemove:]

                previewText = previewText[charactersToRemove:]

        previewTextWidth = self.\_font.size(previewText)[0]

        characterWidth = previewTextWidth / len(previewText)

*#Does same but to make sure previewtext remains in the box*

        if spaceAvailable < previewTextWidth:

            spaceDifference = previewTextWidth - spaceAvailable

        charactersToRemove = int(math.ceil(spaceDifference / characterWidth))

*#Removes it from the back of previewtext*

        previewText = previewText[:-charactersToRemove]

*#Makes string of correct and incorrect text*

        correctText = []

        incorrectText = []

        for i in range(len(text)):

            if text[i] == previewText[i]:

                correctText += previewText[i]

                incorrectText += " "

            else:

                correctText += " "

                incorrectText += previewText[i]

        correctText = "".join(correctText)

        incorrectText = "".join(incorrectText)

*#Location is textbox (x + 5, y + 5)*

        self.\_\_textRenderLocation = (int(self.\_rect.left + 5 \* self.\_resolution[0]), int(self.\_rect.top + 5 \* self.\_resolution[1]))

*#Converts text to surface objects*

        self.\_\_previewTextRender = self.\_font.render(previewText, True, self.\_previewTextColour)

        self.\_\_correctTextRender = self.\_font.render(correctText, True, self.textColour)

        self.\_\_incorrectTextRender = self.\_font.render(incorrectText, True, self.\_incorrectTextColour)

### Testing

I had to do the testing for this much later than I made it due to needing the Scene objects to be present, so please refer to the Race scene for the code used to test this, and this is also why there are 2 textboxes, one for the player and one for the opponent. The number in the middle is the timer that counts down from 30.

This is the code added to allow me to test this easier, it effectively prevents the game from ending when the timer reaches 0.

            elif message == "!GAMECOMPLETED":

*#!THIS IS TEMPORARILY DISABLED*

                self.gameOver = False

*# self.gameOver = True*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | Textbox is drawn correctly | I will run the program | A textbox with 2/5 the width of the resolution and 50/1080 x the resolution height. | Normal | Pass |
| 2 | Textbox will change colour when clicked on | I will click on the box | The colour will change to the chosen one, in this case (40,40,40) from (25,25,25) | Normal | Pass |
| 3 | The front letters will start being deleted when the written text reaches halfway through the box | I will type words into the box until they reach halfway and type a few more letters after | The text at the front will be removed to make room for the new text | Normal | Pass |
| 4 | The backspace key will delete letters when pressed | I will press the backspace key after typing a few letters | The text should be deleted | Normal | Pass |
| 5 | The backspace key will do nothing if no text is present on screen | I will press the backspace key when the program starts | Nothing should happen | Erroneous | Pass |
| 6 | The backspace key will continue to delete letters when held down | I will hold the backspace key after having typed a few words | More than 1 letter should be removed | Normal | Pass |
| 7 | The backspace key will remove entire words when control is held down | I will hold control and press backspace after having typed some words | The word should be deleted | Normal | Pass |
| 8 | The backspace key will remove entire words when control is held down and will continue to do so when backspace is held down | I will hold control and backspace after having typed some words | Words should be deleted very quickly | Normal | Pass |
| 9 | The letters removed from the front will return once the backspace key is pressed | I will type words until the box is halfway and then delete letters | The text at the front should return | Normal | Pass |

#### 1

Screenshot of textbox:

Graphical user interface, text

Description automatically generated

The textbox is the correct size so this test is passed.

#### 2

Screenshot before clicking on the textbox:

Graphical user interface

Description automatically generated

Screenshot after:

Graphical user interface, application

Description automatically generated

The textbox became lighter and so this test is passed.

#### 3

Before typing into the box:

Graphical user interface, text

Description automatically generated

After typing into the box:

Graphical user interface, text

Description automatically generated

The text is red as it does not match but the test is about how the front letters are deleted as the player reaches the middle of the box. This test is passed.

#### 4

Screen before deleting text:

Graphical user interface, text, application

Description automatically generated

Screen after:

Graphical user interface, application

Description automatically generated

The letter was deleted so this test is passed.

#### 5

Before deleting

Graphical user interface, application

Description automatically generated

After:

Graphical user interface, application

Description automatically generated

The game did not crash and so this test is passed.

#### 6

Screen before holding down backspace for 1 second:

Graphical user interface, application

Description automatically generated

Screen after:

Graphical user interface, application

Description automatically generated

This test is passed.

#### 7

Screen before pressing ctrl + backspace:

Graphical user interface, application

Description automatically generated

Screen after:

Graphical user interface, application

Description automatically generated

This test is passed.

#### 8

Screen before:

Graphical user interface, application

Description automatically generated

Screen after:

Graphical user interface, application

Description automatically generated

This test is passed.

#### 9

Before:

Graphical user interface, application

Description automatically generated

After:

Graphical user interface, application

Description automatically generated

This test is passed.

# Class Button

The button object will need to be able to be pressed, change colour when being hovered over and will need to display text with certain size inside it. The size will by default scale to be the biggest while fitting in the box.

The way this will interact with Scene is that it has the public attribute clicked, which will be able to be changed by Scene when the user clicks on the button’s “hitbox”.

The button’s “hitbox” is the pygame.Rect object which has a method called collidepoint(coordinates) which will return True for when the coordinates are in the rectangle and False when they are not.

When I was writing the Pseudocode below I included some setters which did not get used as I didn’t have a use for changing these values after the button was made, but I wrote the code for them in case I needed them later while I remembered that it was more than just setting the attribute to the new value. I have kept them in here but SetText() and SetFont() are not needed for the program to run.

## Pseudocode

CLASS Button

SUBROUTINE Initialise(self, rect, colourActive, colourInactive, textColour, text <-- "")

self.text <-- text

self.\_\_rect <-- rect

self.\_\_activeColour <-- colourActive

self.\_\_inactiveColour <-- colourInactive

self.\_\_colour <-- colourInactive

self.clicked <-- False

self.textColour <-- textColour

find largest text size that fits in the box and create a pygame.Font object set to self.\_\_font

self.textObject <-- Text(self.\_\_font, self.textColour, self.text, location text needs to be)

ENDSUBROUTINE

SUBROUTINE Render(self, window)

draw self.rect rectangle

use self.textObject's Render() method

ENDSUBROUTINE

SUBROUTINE SetText(self, newText)

self.text <-- newText

set self.textObject's text to newText with SetText()

ENDSUBROUTINE

SUBROUTINE SetFont(self, newFont)

self.\_\_font <-- newFont

set textObject's font to newFont

textLocation <-- Centre of self.rect

set self.textObject's location to textLocation

ENDSUBROUTINE

SUBROUTINE SetActive(self)

self.\_\_colour <-- self.\_\_colourActive

ENDSUBROUTINE

SUBROUTINE SetInactive(self)

self.colour <-- self.\_\_colourInactive

ENDSUBROUTINE

SUBROUTINE CheckForCollision(self, mouseLocation)

check if mouseLocation is inside self.rect

ENDSUBROUTINE

ENDCLASS

## Python code

import pygame

from Text import Text

class Button():

    def \_\_init\_\_(self, rect, colourActive, colourInactive, textColour, text=""):

        self.text = text

        self.\_\_rect = rect

        self.\_\_activeColour = colourActive

        self.\_\_inactiveColour = colourInactive

        self.\_\_colour = colourInactive

        self.clicked = False

        self.textColour = textColour

*#Finds correct fontsize*

        fontSize = 1

        font = pygame.font.SysFont("Calibri", int(fontSize))

        fontRenderSize = font.size(self.text)

*#Checks if the text will fit in the texbox*

        while fontRenderSize[0] < self.\_\_rect.size[0] and fontRenderSize[1] < self.\_\_rect.size[1]:

            fontSize += 1

            font = pygame.font.SysFont("Calibri", int(fontSize))

            fontRenderSize = font.size(self.text)

        self.\_\_font = pygame.font.SysFont("Calibri", int(fontSize - 1))

*#Makes Text object*

        textSize = self.\_\_font.size(self.text)

        textLocation = (int(self.\_\_rect.left + (self.\_\_rect.width - textSize[0]) / 2), int(self.\_\_rect.top + (self.\_\_rect.height - textSize[1]) / 2))

        self.textObject = Text(self.\_\_font, self.textColour, self.text, location=textLocation)

    def Render(self, window):

        pygame.draw.rect(window, self.\_\_colour, self.\_\_rect)

        self.textObject.Render(window)

    def SetText(self, newText):

        self.text = newText

        self.textObject.SetText(newText)

    def SetFont(self, newFont):

        self.\_\_font = newFont

        self.textObject.SetFont(newFont)

        textLocation = (int(self.\_\_rect.x + (self.\_\_rect.width - self.textObject.textRender.get\_size()[0]) / 2), int(self.\_\_rect.y + (self.\_\_rect.height - self.textObject.textRender.get\_size()[1]) / 2))

        self.textObject.SetLocation(textLocation)

    def SetActive(self):

        self.\_\_colour = self.\_\_activeColour

    def SetInactive(self):

        self.\_\_colour = self.\_\_inactiveColour

    def CheckForCollision(self, pos):

        if self.\_\_rect.collidepoint(pos):

            return True

        return False

## Code for testing Button:

import ctypes

ctypes.windll.user32.SetProcessDPIAware()

window = pygame.display.set\_mode((0,0), pygame.FULLSCREEN)

pygame.display.set\_caption("SpeedTyper")

pygame.font.init()

ThisButton = Button("TestButton", (100,100), (200,50), (30,30,30), (35,35,35), (255,255,255), 1080)

running = True

stillInBox = False

while running:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            running = False

        elif event.type == pygame.MOUSEBUTTONDOWN:

            if ThisButton.CheckIfHovering(pygame.mouse.get\_pos()):

                stillInBox = True

                print("Clicked on box")

        elif event.type == pygame.MOUSEBUTTONUP and stillInBox:

            ThisButton.clicked = True

    if ThisButton.CheckIfHovering(pygame.mouse.get\_pos()):

        ThisButton.SetActive()

    else:

        ThisButton.SetDormant()

        stillInBox = False

    ThisButton.Render(window)

    pygame.display.update()

All this code does is check if the mouse is hovering on the button (to change the colour), check for clicks and to check if the player presses alt + f4 to close the game.

## Testing for Button:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | The button is rendered on the screen | I will run the program. | The button is rendered on the screen. | Normal | Pass |
| 2 | The button changes colour when hovered over | I will hover over the button. | The button should turn a lighter colour. | Normal | Pass |
| 3 | The button will register being clicked when I click on it | I will click on the button. | The program log should print that the button has been clicked. | Normal | Pass |
| 4 | The button will be created in the correct location | I will change the code to set the location of the button to the centre of the screen. | The button should be located at the centre of the screen. | Normal | Pass |
| 5 | The button should be the correct size | I will set the button to be 500x500 pixels. | The button should be a square with dimensions 500x500. | Normal | Pass |
| 6 | The button text should be the maximum size it can be while still fitting in the box | I will set the button to be 500x500 pixels. | The button text should fit the square. | Normal | Pass |

### 1

The button needs to be rendered on the screen when appropriate. When the program is run, a window is opened with a button labelled “test button”. A screenshot is below.

Shape

Description automatically generated with low confidence

This test is passed.

### 2

The button needs to change colour when the mouse is hovering over the button. This is to give the user some feedback that the game is not frozen/the button can be clicked. This can be tested very simply by comparing the colour before and after hovering over it.

The colour has been set to turn from grey to white for the purposes of testing, but in the main program the colour change will be much more subtle.

Before:

Shape

Description automatically generated with low confidence

After:

Shape

Description automatically generated with low confidence

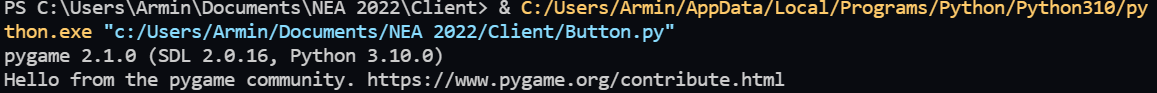
The colour has changed when I hover over the button so this test is passed.

### 3

The button needs to be able to be pressed and do something in order to fulfil the function of being a button. This can be tested by recording the “trigger” of the button. Trigger being the detection of the button being clicked.

I have made it so that whenever I click on the button the console prints a message to say that it was clicked. This will allow me to test it by running the program, clicking the button 3 times and seeing the output.

Before clicking:



After clicking 3 times:

Text

Description automatically generated

This test is passed.

### 4

The button needs to be displayed in the correct location on the screen. This will only need to be done per scene so that means that I can simply change where the button is placed before running the program.

To test this, I will set the location of the button to be in the centre of the screen. This will not centre the button; it will set the top left corner of the button to be the centre of the screen.

To achieve this I used the ctypes module in python to get the screen resolution, so the start of the program has been updated to the following:

import ctypes

user32 = ctypes.windll.user32

*#Prevents the screen from scaling with windows resolution scale*

user32.SetProcessDPIAware()

*#Gets the screen resolution*

dispWidth = user32.GetSystemMetrics(0)

dispHeight = user32.GetSystemMetrics(1)

This sets dispWidth and dispHeight to the correct values, which can be used in the following to set the location of the button.

ThisButton = Button("TestButton", (dispWidth / 2, dispHeight / 2), (200,50), (30,30,30), (100,100,100), (255,255,255), 1080)

After the changes, the window displayed looks like this:

Graphical user interface

Description automatically generated with medium confidence

To test the location of this I simply pasted this screenshot into paint.net and checked the position of the top left corner of the button.

A screenshot of a computer

Description automatically generated

In the bottom left it says selection top left: 960, 540 which is the centre of the screen. This test is passed.

### 5

The button should be the correct size in order to fit the screen properly. To test this, I will set the button to be 500x500 and then check the dimensions using paint.net.

ThisButton = Button("TestButton", (dispWidth / 2, dispHeight / 2), (500,500), (30,30,30), (100,100,100), (255,255,255), 1080)

Running the code with the above changes results in the following:

Graphical user interface

Description automatically generated with medium confidence

In paint.net:

A screenshot of a computer

Description automatically generated

In the bottom left it says bounding rectangle size: 500 x 500, so this test is passed.

### 6

The button text needs to be as big as possible while still fitting the box. To test this I will set the button to be 100x100, paste a screenshot and then set it to 500x500 to check if the text size will scale with it.

Object creation for 100x100:

Object creation for 500x500:

ThisButton = Button("TestButton", (dispWidth / 2, dispHeight / 2), (500,500), (30,30,30), (100,100,100), (255,255,255), 1080)

Button with dimensions 100x100:

Graphical user interface, application

Description automatically generated

Button with dimensions 500x500:

Graphical user interface

Description automatically generated with medium confidence

The text size scales with button size and so this test is passed.

# Text

This is another object that is heavily used in Scene. This is because it allows for rendering text on screen very easily and makes the code much tidier and readable.

The benefit of having this is that it allows for the colour, location and pygame.Surface object to be stored in one object and so can be put in a list which gets iterated through to render everything.

## UpdateRender

This method is called whenever an attribute is changed that would change the appearance of the button. Rendering any surface in pygame is less efficient than just drawing it on a screen, this helps improve performance as the surface is only created when it is changed.

## Render

This method is called to use the data in the object to draw the Text on screen.

## SetColour

This method is used to change the colour of the text, it is used when the colour needs to be changed when the object has already been created.

## SetFont

This method is used when the font (either size or style) needs to be changed.

## SetText

This method is used when the text needs to be changed.

## Pseudocode

CLASS Text

SUBROUTINE init(self, font, colour, text, location)

self.\_\_font <-- font

self.\_\_colour <-- colour

self.\_\_text <-- text

self.\_\_location <-- location

self.\_\_textRender <-- render self.\_\_text

ENDSUBROUTINE

SUBROUTINE UpdateRender(self)

self.\_\_textRender <-- render self.\_\_text

ENDSUBROUTINE

SUBROUTINE SetText(self, newText)

self.\_\_text <-- newText

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE SetColour(self, newColour)

self.\_\_colour <-- newColour

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE SetFont(self, newFont)

self.\_\_font <-- newFont

self.UpdateRender()

ENDSUBROUTINE

SUBROUTINE Render(self, window)

draw self.textRender on window

ENDSUBROUTINE

ENDCLASS

## Python code

*#Text object that is stored in an array of objects in scene objects*

class Text:

*#Font is the pygame font object that it will be rendered with*

*#Colour is the RGB value in the form of a tuple*

*#Text is the text that needs to be displayed*

*#Location is the location of the top left of the render*

    def \_\_init\_\_(self, font, colour = (255,255,255), text = "", location = (0,0)):

        self.\_\_font = font

        self.\_\_colour = colour

        self.\_\_text = text

        self.location = location

*#Pygame surface object that can be drawn on other surface objects*

        self.textRender = self.\_\_font.render(self.\_\_text, True, self.\_\_colour)

*#Having this here reduces number of times it needs to be rendered, better performance*

    def UpdateRender(self):

        self.textRender = self.\_\_font.render(self.\_\_text, True, self.\_\_colour)

*#Changes the text that needs to be rendered*

    def SetText(self, newText):

        self.\_\_text = newText

        self.UpdateRender()

*#Changes the colour*

    def SetColour(self, newColour):

        self.\_\_colour = newColour

        self.UpdateRender()

*#Changes the font*

    def SetFont(self, newFont):

        self.\_\_font = newFont

        self.UpdateRender()

*#Renders the text onto the given window*

    def Render(self, window):

        window.blit(self.textRender, self.location)

## Testing

//Todo

# Scene

After having made the previous classes, making Scenes become much easier. The scene uses InputBoxes, TextBoxes, Buttons and Text. This is what makes up the UI for the game. Different scenes will use different combinations of these, which can be seen in the UML diagram earlier in this document but they will all have the same common attributes.

## Common attributes

The scene object is made to be inherited, in that all scene objects will get treated in the same way. In the Game object, each Scene object will have its main() method called and then the program will check if the userQuit attribute is True. If it is then the program terminates, otherwise it will call the main() method again.

userQuit is simply a Boolean attribute which will be set to False upon the creation of a new object. This will then be set to false if the input handling of the scene finds the user to have quit in any way, such as the “Quit” button in the main menu or by pressing alt + f4 at any other time.

The socket attribute is public as it needs to be set to the correct one after the creation of the object. When the object is created, it can either take socket as a parameter, but cannot always. The case when it cannot do this is in the ConnectionScreen Scene. This is because the ConnectionScreen scene is what is responsible for creating the socket they all use, which is then detected by the Game object and then given to all the other Scene objects. The socket attribute is actually a ClientSocket object, which was made earlier in this document.

The rest of the attributes here are all protected, as they need to be usable by the child objects.

Shift, alt, ctrl and backspace are all Booleans which are changed inside the HandleInputs method. These are changed based on what the player presses and are used to determine whether certain inputs should do certain things. For example, ctrl and backspace are used to determine if the deletion of characters should be word at a time or single character at a time, as when the player is holding down ctrl and backspace, full words should be deleted.

timeSinceLastBackspace is an integer which is used to measure how much time(ms) since the last character was deleted. This is updated every frame in the main() method of every Scene object. When it is above a certain amount and backspace is being held down, it will delete another character, this time is usually 50 seconds. When the button is initially pressed, there needs to be a delay before automatic deletion starts, therefore whenever the backspace key is pressed this attribute is set to -200, so that there is a 250ms delay until automatic deletion starts.

The lists of objects are also attributes used in the Render method to call each of the objects which are stored in them’s Render method. This is how rendering is done of the Box, Button and Text objects. Any objects that the scene needs to render will be stored in these.

Window is the pygame.display object which is used to display things onto.

Resolution is a tuple of the resolution scale, this is used to keep everything proportional to the screen resolution. The way this is done is by multiplying the measurements on a 1920x1080 screen by resolution[0] for width and resolution [1] for height. When the actual resolution is needed, these values are multiplied by 1920 and 1080 for actual pixels.

Clock is a pygame.time.Clock object which is used for the adjustment of timeSinceLastBacspace using the Clock.tick() and Clock.get\_time() methods. Tick() will update the clock to store the time since the last Clock.tick() method call. Get\_time() returns this value (in ms).

inputHandler is an InputHandler object which simply returns the keypresses the user makes, so that they can be selectively used and to prevent them from discarding every unused input.

backGroundSurface is a surface object used to fill the background. Instead of rendering the 1920x1080 pixels every frame, this stores them until they need to be changed. This helps performance which was implemented when I was working on the RaceScene Scene, as I had performance issues before changing something that was causing those issues.

## Pseudocode

CLASS Scene

SUBROUTINE init(self, window, resolution, socket <-- None)

self.userQuit <-- False

self.socket <-- socket

self.\_shift <-- False

self.\_alt <-- False

self.\_ctrl <-- False

self.\_backspace <-- False

self.\_timeSinceLastBackspace <-- 0

self.\_listOfBoxObjects <-- []

self.\_listOfButtonObjects <-- []

self.\_listOfTextObjects <-- []

self.\_window <-- window

self.\_resolution <-- resolution

self.\_clock <-- pygame Clock object

self.\_inputHandler <-- InputHandler object

self.\_backgroundSurface <-- Black surface object which is the full screen in dimensions

draw self.\_backgroundSurface onto self.\_window

ENDSUBROUTINE

SUBROUTINE main(self)

self.\_timeSinceLastBackspace <-- self.\_timeSinceLastBackspace + time since last frame

self.\_HandleInputs()

IF self.\_backspace AND self.\_timeSinceLastBackspace ><-- 50 THEN

FOR box IN self.\_listOfBoxObjects

IF box.isActive THEN

box.RemoveLetter(self.\_ctrl)

ENDIF

ENDFOR

self.\_timeSinceLastBackspace <-- 0

ENDIF

FOR button in self.\_listOfButtonObjects

IF button.CheckForCollision(mouse location) THEN

button.SetActive()

ELSE

button.SetInactive()

ENDIF

ENDFOR

self.\_Render()

ENDSUBROUTINE

SUBROUTINE \_Render(self)

draw self.\_backgroundSurface on self.\_window

FOR box in self.\_listOfBoxObjects

box.Render(self.\_window)

ENDFOR

FOR button in self.\_listOfButtonObjects

button.Render(self.\_window)

ENDFOR

FOR text in self.\_listOfTextObjects

text.Render(self.\_window)

ENDFOR

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

self.\_inputHandler.CheckInputs()

i <-- 0

WHILE i < LEN(self.InputHandler.inputsList)

IF self.\_inputHandler.inputsList[i] = "QUIT" THEN

self.userQuit <-- True

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "SHIFTDOWN" THEN

self.\_shift <-- True

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "SHIFTUP" THEN

self.\_shift <-- False

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "ALTDOWN" THEN

self.\_alt <-- True

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "ALTUP" THEN

self.\_alt <-- False

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "CONTROLDOWN" THEN

self.\_ctrl <-- True

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "CONTROLUP" THEN

self.\_ctrl <-- False

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "BACKSPACEDOWN" THEN

self.\_backspace <-- True

self.\_timeSinceLastBackspace <-- -200

FOR box IN self.\_listOfBoxObjects THEN

IF box.isActive THEN

box.RemoveLetter(self.\_ctrl)

ENDFOR

self.\_inputHandler.inputsList.POP(i)

ELSE IF self.\_inputHandler.inputsList[i] = "BACKSPACEUP" THEN

self.\_backspace <-- False

self.\_inputHandler.inputsList.POP(i)

ELSE THEN

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

ENDCLASS

## Python code

class Scene:

    def \_\_init\_\_(self, window, resolution, socket : ClientSocket = None) -> None:

        self.userQuit = False

        self.socket = socket

*#Whether or not these are held down*

        self.\_shift = False

        self.\_alt = False

        self.\_ctrl = False

        self.\_backspace = False

        self.\_timeSinceLastBackspace = 0

*#Used to redner things on screen*

        self.\_listOfBoxObjects = []

        self.\_listOfButtonObjects = []

        self.\_listOfTextObjects = []

        self.\_window = window

        self.\_resolution = resolution

        self.\_clock = pygame.time.Clock()

        self.\_inputHandler = InputHandler()

*#Helps performance to have a surface object instead of filling the entire screen every frame*

        self.\_backgroundSurface = pygame.Surface((int(self.\_resolution[0] \* 1920), int(self.\_resolution[1] \* 1080)))

        self.\_backgroundSurface.fill((0,0,0))

    def main(self):

        self.\_clock.tick()

        self.\_timeSinceLastBackspace += self.\_clock.get\_time()

        self.\_HandleInputs()

*#Automatic removal of text every 50 milliseconds*

        if self.\_backspace and self.\_timeSinceLastBackspace >= 50:

            for box in self.\_listOfBoxObjects:

                if box.isActive:

                    box.RemoveLetter(self.\_ctrl)

            self.\_timeSinceLastBackspace = 0

        for buttonObject in self.\_listOfButtonObjects:

            if buttonObject.CheckForCollision(pygame.mouse.get\_pos()):

                buttonObject.SetActive()

            else:

                buttonObject.SetInactive()

        self.\_Render()

    def \_Render(self):

        self.\_window.blit(self.\_backgroundSurface, (0,0))

        for box in self.\_listOfBoxObjects:

            box.Render(self.\_window)

        for button in self.\_listOfButtonObjects:

            button.Render(self.\_window)

        for textObject in self.\_listOfTextObjects:

            textObject.Render(self.\_window)

    def \_HandleInputs(self):

        self.\_inputHandler.CheckInputs()

        i = 0

        while i < len(self.\_inputHandler.inputsList):

            if self.\_inputHandler.inputsList[i] == "QUIT":

                self.userQuit = True

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "SHIFTDOWN":

                self.\_shift = True

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "SHIFTUP":

                self.\_shift = False

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "ALTDOWN":

                self.\_alt = True

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "ALTUP":

                self.\_alt = False

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "CONTROLDOWN":

                self.\_ctrl = True

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "CONTROLUP":

                self.\_ctrl = False

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "BACKSPACEDOWN":

                self.\_backspace = True

                self.\_timeSinceLastBackspace = -200

                for box in self.\_listOfBoxObjects:

                    if box.isActive:

                        box.RemoveLetter(self.\_ctrl)

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i] == "BACKSPACEUP":

                self.\_backspace = False

                self.\_inputHandler.inputsList.pop(i)

            else:

                i += 1

This and all the Scene subclasses are in the Scene.py file, which has this at the top:

import pygame, threading, socket, json

from ClientSocket import ClientSocket

from InputHandler import InputHandler

from Boxes import InputBox, TextBox

from Button import Button

from Text import Text

These are the modules/classes from different files used in the Scene.py file.

# Game object

The Game object is where the game takes place, in that this is where the Scene is chosen. I decided to have every scene have its own method so that they can be edited separately and for the actual loop that calls these methods is more readable.

All these methods that the game object has follow a similar structure, they have a while loop that checks if the condition for leaving that scene has been met and if not they will call the main method of that scene. It will then check if the user quit and if not then it will update the display.

Because it isn’t helpful to have all of these in one place, I will separate all these modules and spread them out in the document so they are close to the respective Scene.

The main loop however, I will include here.

## Pseudocode

CLASS Game

SUBROUTINE init(self, window, settings)

self.socket <-- None

self.settings <-- settings

self.\_\_loggedIn <-- False

self.\_\_window <-- window

self.\_\_userQuit <-- False

self.\_\_socketHandleThread <-- None

self.\_\_resolution <-- resolution divided by 1920 and 1080 to get scale

self.\_\_connectionScreen <-- None

self.\_\_loginScreen <-- None

self.\_\_registerScreen <-- None

self.\_\_mainMenu <-- None

self.\_\_settingsScreen <-- None

self.\_\_statisticsScreen <-- None

self.\_\_matchmakingScreen <-- None

self.\_\_timerScene <-- None

self.\_\_raceScene <-- None

self.\_\_postGameScreen <-- None

self.\_\_timerStarted <-- False

self.\_\_textToWrite <-- None

self.\_\_results <-- None

ENDSUBROUTINE

SUBROUTINE main(self)

self.\_\_ConnectToServer()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

WHILE NOT self.\_\_loggedIn

self.\_\_Login()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.loginscreen.userWantsToCreateAccount THEN

self.\_\_Register()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.\_\_registerScreen.registered THEN self.loggedIn <-- True

ENDIF

ENDIF

ENDWHILE

WHILE True

self.\_\_mainMenu <-- MainMenu object

self.\_\_matchmakingScreen <-- MatchmakingScreen object

self.\_\_timerScene <-- TimerScene object

self.\_\_timerStarted <-- False

self.\_\_textToWrite <-- None

self.\_\_results <-- None

self.\_\_margin <-- None

self.\_\_ELodiff <-- None

self.\_\_stats <-- None

self.\_\_GetChoiceFromUser()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

userChoice <-- self.\_\_mainMenu.userChoice

IF userChoice = "Play" THEN

self.\_\_WaitForGame()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.\_\_matchmakingScreen.userClickedBackButton THEN

self.socket.msgsToSend.APPEND("!DEQUEUE")

ELSE IF self.\_\_matchmakingScreen.gameFound THEN

self.\_\_PreGameTimer()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

self.\_\_WaitForText()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

self.\_\_raceScene <-- RaceScene object

self.\_\_PlayGame()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

self.\_\_WaitForMatchResult()

self.\_\_postGameScreen = PostGame object

self.\_\_PostGame()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

ENDIF

ELSE IF userChoice = "Statistics" THEN

self.\_\_WaitForData()

self.\_\_ShowStatistics()

ELSE IF userChoice = "Settings" THEN

self.\_\_Settings()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

ENDIF

ENDWHILE

ENDSUBROUTINE

ENDCLASS

## Python code

import pygame, threading, pickle

from Scene import \*

class Game:

    def \_\_init\_\_(self, window, settings):

        self.socket = None

        self.settings = settings

        self.\_\_loggedIn = False

        self.\_\_window = window

        self.\_\_userQuit = False

*#Temporary variable which will be set once the socket is connected*

        self.\_\_socketHandleThread = None

        self.\_\_resolution = pygame.display.Info()

        self.\_\_resolution = (self.\_\_resolution.current\_w / 1920, self.\_\_resolution.current\_h / 1080)

        self.\_\_connectionScreen : ConnectionScreen = None

        self.\_\_loginScreen : LoginScreen = None

        self.\_\_registerScreen : RegisterScreen = None

        self.\_\_mainMenu : MainMenu = None

        self.\_\_settingsScreen : SettingsScreen = None

        self.\_\_statisticsScreen : StatisticsScreen = None

        self.\_\_matchmakingScreen : MatchmakingScreen = None

        self.\_\_timerScene : TimerScene = None

        self.\_\_raceScene : RaceScene = None

        self.\_\_postGameScreen : PostGame = None

        self.\_\_timerStarted = False

        self.\_\_textToWrite = None

        self.\_\_results = None

    def main(self):

*#Connects to server*

        self.\_\_ConnectToServer()

*#I couldn't come up with a more readable solution for having to end the game after any of these scenes than to simply check after every one*

*#I previously had a solution with an activescene attribute but that was unreadable and provided no real benefit*

        if self.\_\_userQuit:

            return 0

        while not self.\_\_loggedIn:

*#Goes to loginscreen*

            self.\_\_Login()

            if self.\_\_userQuit:

                return 0

            elif self.\_\_loginScreen.userWantsToCreateAccount:

                self.\_\_Register()

                if self.\_\_userQuit:

                    return 0

                if self.\_\_registerScreen.registered:

                    self.\_\_loggedIn = True

*#Main loop where player goes to main menu and back#*

        while True:

            self.\_\_mainMenu = MainMenu(self.\_\_window, self.\_\_resolution, self.socket)

            self.\_\_matchmakingScreen = MatchmakingScreen(self.\_\_window, self.\_\_resolution, self.socket)

            self.\_\_timerScene = TimerScene(self.\_\_window, self.\_\_resolution, self.socket)

            self.\_\_timerStarted = False

            self.\_\_textToWrite = None

            self.\_\_results = None

            self.\_\_margin = None

            self.\_\_ELodiff = None

            self.\_\_stats = None

*#Goes to main menu to get choice from user*

            self.\_\_GetChoiceFromUser()

            if self.\_\_userQuit:

                return 0

            userChoice = self.\_\_mainMenu.userChoice

            if userChoice == "Play":

*#Goes to matchmaking screen*

                self.\_\_WaitForGame()

*#If user quit during matchmaking*

*#Disconnecting from server is done in Play.py*

*#If the user disconnects from server while in queue, server will automatically dequeue them*

                if self.\_\_userQuit:

                    return 0

*#If user left queue*

                elif self.\_\_matchmakingScreen.userClickedBackButton:

                    self.socket.msgsToSend.append("!DEQUEUE")

*#Loop starts again and user goes to main menu*

*#If game was found*

                elif self.\_\_matchmakingScreen.gameFound:

                    self.\_\_PreGameTimer()

                    if self.\_\_userQuit:

                        return 0

*#Waits for previewtext from server*

                    self.\_\_WaitForText()

                    if self.\_\_userQuit:

                        return 0

*#Plays game while timer is more than 0*

                    self.\_\_raceScene = RaceScene(self.\_\_window, self.\_\_resolution, self.\_\_textToWrite, self.socket)

                    self.\_\_PlayGame()

                    if self.\_\_userQuit:

                        return 0

                    self.\_\_WaitForMatchResult()

*#Goes to post game screen*

                    self.\_\_postGameScreen = PostGame(self.\_\_window, self.\_\_resolution, self.\_\_results, self.\_\_margin, self.\_\_ELodiff, self.socket)

                    self.\_\_PostGame()

                    if self.\_\_userQuit:

                        return 0

            elif userChoice == "Statistics":

                self.\_\_WaitForData()

                self.\_\_ShowStatistics()

            elif userChoice == "Settings":

                self.\_\_Settings()

                if self.\_\_userQuit:

                    return 0

CheckMessages()

def \_\_CheckMessages(self):

    i = 0

    while i < len(self.socket.receivedMsgs):

*#If new phase started*

        try:

            self.\_\_stats = pickle.loads(self.socket.receivedMsgs[i])

            print("Pickle acquired")

            self.socket.receivedMsgs.pop(i)

        except:

            if self.socket.receivedMsgs[i][:11] == "!STARTTIMER":

                self.\_\_timerStarted = True

                self.socket.receivedMsgs.pop(i)

            elif self.socket.receivedMsgs[i][:13] == "!TEXTTOWRITE:":

                self.\_\_textToWrite = self.socket.receivedMsgs[i][13:]

                self.socket.receivedMsgs.pop(i)

            elif self.socket.receivedMsgs[i][:14] == "!MATCHOUTCOME:":

                self.\_\_results = self.socket.receivedMsgs[i][14:]

                if self.socket.receivedMsgs[i][14:] == "DRAW":

                    self.\_\_margin = 0

                    self.\_\_ELodiff = 0

                self.socket.receivedMsgs.pop(i)

            elif self.socket.receivedMsgs[i][:5] == "!ELO:":

                self.\_\_ELodiff = self.socket.receivedMsgs[i][5:]

                self.socket.receivedMsgs.pop(i)

            elif self.socket.receivedMsgs[i][:8] == "!MARGIN:":

                self.\_\_margin = self.socket.receivedMsgs[i][8:]

                self.socket.receivedMsgs.pop(i)

            else:

                i += 1

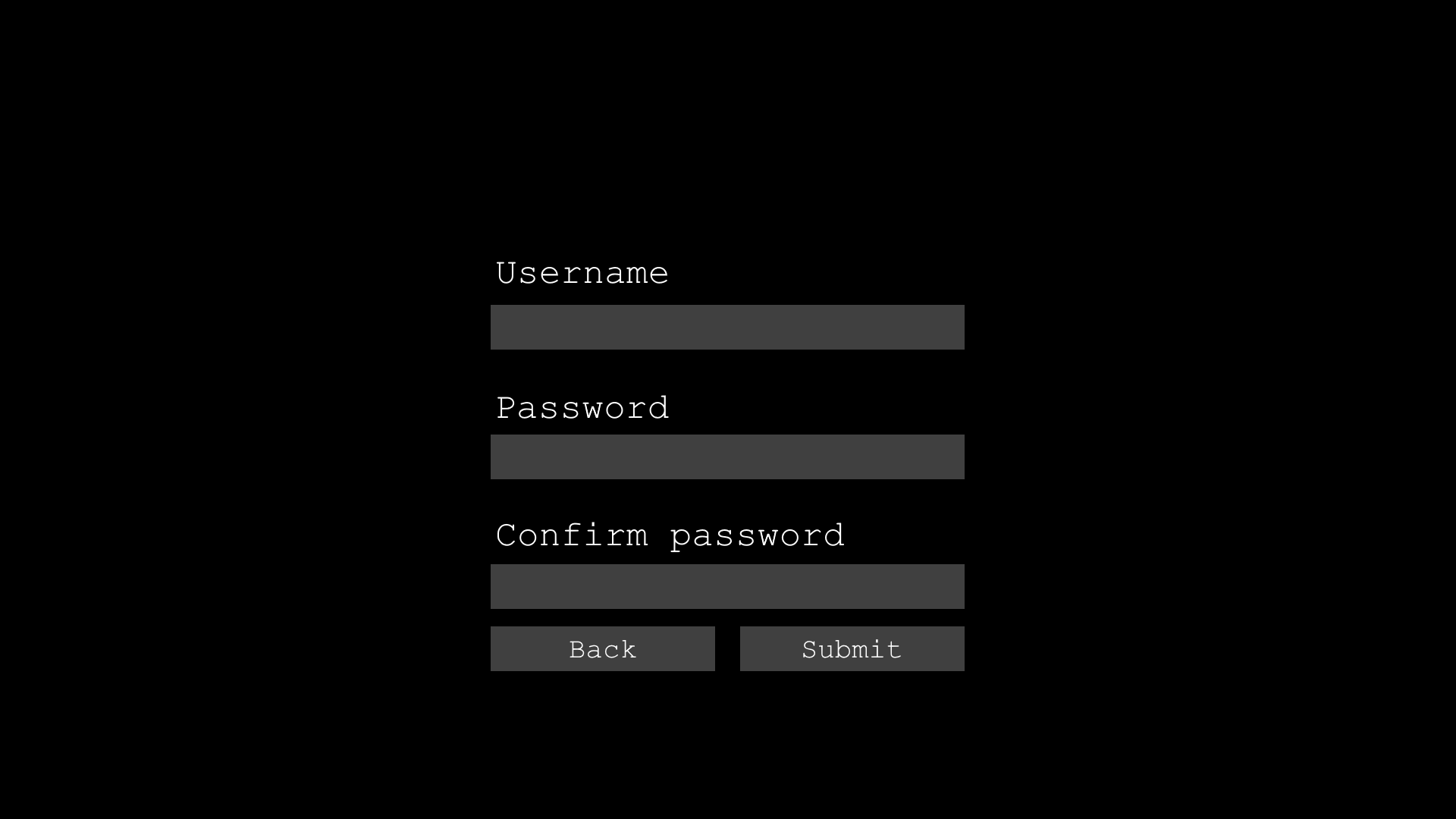
## Testing

This will be tested at the end, as it is better to test everything else first to make sure they are not the problem.

# Account Creation Screen

The player needs to be able to create an account. This can be done by using a new Scene.

Whenever I make a new scene it helps to come up with a design in paint.net and use the measurements of that design to get an idea of where to put all the objects. This is the design I came up with for this:



The username box should display the text written normally and the password box should display the text with “\*” instead of the letters. This is to prevent someone seeing the player’s password if they can see the screen. This is done by having the argument for hashed set to True when creating the InputBox object.

When the back button is pressed, the user is sent back to the login screen, where this screen is accessed from.

When the submit button is pressed, details are sent to the server and the server will attempt to add them to the database with default values, most being 0 as they are statistics. The database is super basic and is only 1 table, this table only stores data about the user such as timeplayed. While some of these statistics could be calculated by having more tables for each game played, there is no feature that would need information about specific games and so I decided to simply update these statistics whenever the race is finished.

If the username entered is already in the database, the server will send a message back saying that creation of account has been failed, this will empty the boxes and the user will have to enter new details.

## Pseudocode

CLASS ResgisterScreen(Scene)

SUBROUTINE init(self, window, resolution, socket <-- None)

self.backbuttonPressed <-- False

self.registered <-- False

self.\_\_submitButtonPressed <-- False

boxFont <-- pygame font object size 36

boxColourActive <-- (40,40,40)

boxColourInactive <-- (25,25,25)

boxRectSize <-- (625, 60)

self.\_\_usernameBox <-- InputBox object

self.\_\_passwordBox <-- InputBox object

self.\_\_confirmBox <-- InputBox object

self.\_listOfBoxObjects <-- [self.\_\_usernameBox, self.\_\_passwordBox, self.\_\_confirmBox]

buttonSize <-- (300,60)

self.\_\_backButton <-- Button object

self.\_\_submitButton <-- Button object

self.\_listOfButtonObjects = [self.\_\_backButton, self.\_\_submitButton]

#Username text

textFont <-- pygame font object with size 36

self.\_\_usernameText <-- Text object

self.\_\_passwordText <-- Text object

self.\_\_confirmText <-- Text object

self.\_listOfTextObjects <-- [self.\_\_usernameText, self.\_\_passwordText, self.\_\_confirmText]

ENDSUBROUTINE

SUBROUTINE main()

FOR button IN self.\_listOfButtonObjects

IF button.clicked THEN

IF button = self.\_\_backButton THEN

self.backbuttonPressed <-- True

ELSE IF button = self.\_\_submitButton THEN

IF self.\_\_passwordBox.text = self.\_\_confirmBox.text THEN

self.socket.msgsToSend.APPEND("!REGISTER:" + self.\_\_usernameBox.text + "," + self.\_\_passwordBox.text)

self.self.\_\_submitButtonPressed <-- True

ELSE

self.\_\_submitButtonPressed <-- False

button.clicked <-- False

FOR box IN self.\_listOfBoxObjects

box.text <-- ""

ENDFOR

ENDIF

ENDIF

ENDIF

ENDFOR

WHILE self.\_\_submitButtonPressed AND NOT self.registered

self.\_\_HandleMessages()

ENDWHILE

ENDSUBROUTINE

SUBROUTINE \_\_HandleMessages(self)

i <-- 0

WHILE i < LEN(self.socket.receivedMsgs)

IF self.socket.receivedMsgs[i] = "!REGISTEREDSUCCESFULLY" THEN

self.registered <-- True

self.socket.receivedMsgs.POP(i)

ELSE IF self.socket.receivedMsgs[i] = "!ANERROROCCURRED" THEN

reset scene so that it waits for user to try again

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

inherit from Scene

i <-- 0

WHILE i < LEN(self.\_inputHandler.inputsList)

IF input is a click THEN

IF click location is in a box THEN

set box to active

ELSE

set box to inactive

ENDIF

IF click location is in a button THEN

set button to clicked

ENDIF

remove input from list of inputs

ELSE IF input is a keypress THEN

IF key is a normal letter THEN

IF a box is active THEN

add letter to active box

ENDIF

ENDIF

remove input from list of inputs

ELSE IF input is the tab key THEN

change activetext to the next one down or the first if the active box is the bottom one

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

ENDCLASS

## Python code

class RegisterScreen(Scene):

    def \_\_init\_\_(self, window, resolution, socket: ClientSocket = None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        self.backbuttonPressed = False

        self.registered = False

        self.\_\_submitButtonPressed = False

        boxFont = pygame.font.SysFont("Courier New", int(28 \* self.\_resolution[1]))

        boxColourActive = (40,40,40)

        boxColourInactive = (25,25,25)

        boxRectSize = (int(625 \* self.\_resolution[0]), int(60 \* self.\_resolution[1]))

*#Usernamebox*

        usernameRectLocation = (int((self.\_resolution[0] \* 1920 - boxRectSize[0]) / 2), int(400 \* self.\_resolution[1]))

        usernameRect = pygame.Rect(usernameRectLocation, boxRectSize)

        self.\_\_usernameBox = InputBox(usernameRect, boxFont, self.\_resolution, boxColourActive, boxColourInactive, (255,255,255), "")

*#Passwordbox*

        passwordRectLocation = (int((self.\_resolution[0] \* 1920 - boxRectSize[0]) / 2), int(570 \* self.\_resolution[1]))

        passwordRect = pygame.Rect(passwordRectLocation, boxRectSize)

        self.\_\_passwordBox = InputBox(passwordRect, boxFont, self.\_resolution, boxColourActive, boxColourInactive, (255,255,255), "", hashed=True)

*#Password confirm box*

        confirmRectLocation = (int((self.\_resolution[0] \* 1920 - boxRectSize[0]) / 2), int(740 \* self.\_resolution[1]))

        confirmRect = pygame.Rect(confirmRectLocation, boxRectSize)

        self.\_\_confirmBox = InputBox(confirmRect, boxFont, self.\_resolution, boxColourActive, boxColourInactive, (255,255,255), "", hashed=True)

        self.\_listOfBoxObjects = [self.\_\_usernameBox, self.\_\_passwordBox, self.\_\_confirmBox]

        buttonSize = (int(300 \* self.\_resolution[0]), int(60 \* self.\_resolution[1]))

*#Back button*

        backButtonLocation = (int(650 \* self.\_resolution[0]), int(830 \* self.\_resolution[1]))

        backButtonRect = pygame.Rect(backButtonLocation, buttonSize)

        self.\_\_backButton = Button(backButtonRect, boxColourActive, boxColourInactive, (255,255,255), text="Back")

*#Submit button*

        submitButtonLocation = (int(980 \* self.\_resolution[0]), int(830 \* self.\_resolution[1]))

        submitButtonRect = pygame.Rect(submitButtonLocation, buttonSize)

        self.\_\_submitButton = Button(submitButtonRect, boxColourActive, boxColourInactive, (255,255,255), text="Submit")

        self.\_listOfButtonObjects = [self.\_\_backButton, self.\_\_submitButton]

*#Username text*

        textFont = pygame.font.SysFont("Courier New", int(36 \* self.\_resolution[1]))

        self.\_\_usernameText = Text(textFont, text="Username", location=(int(655 \* self.\_resolution[0]), int(350 \* self.\_resolution[1])))

        self.\_\_passwordText = Text(textFont, text="Password", location=(int(655 \* self.\_resolution[0]), int(520 \* self.\_resolution[1])))

        self.\_\_confirmText = Text(textFont, text="Confirm password", location=(int(655 \* self.\_resolution[0]), int(690 \* self.\_resolution[1])))

        self.\_listOfTextObjects = [self.\_\_usernameText, self.\_\_passwordText, self.\_\_confirmText]

    def main(self):

        super().main()

*#Checks if buttons were pressed*

        for button in self.\_listOfButtonObjects:

            if button.clicked:

                if button == self.\_\_backButton:

                    self.backbuttonPressed = True

                elif button == self.\_\_submitButton:

                    if self.\_\_passwordBox.text == self.\_\_confirmBox.text and self.\_\_passwordBox.text != "":

                        self.socket.msgsToSend.append(f"!REGISTER:{self.\_\_usernameBox.text},{self.\_\_passwordBox.text}")

                        self.\_\_submitButtonPressed = True

                    else:

                        self.\_\_submitButton.clicked = False

                        button.clicked = False

                        for box in self.\_listOfBoxObjects:

                            box.text = ""

        while self.\_\_submitButtonPressed and not self.registered:

            self.\_\_HandleMessages()

    def \_\_HandleMessages(self):

        i = 0

        while i < len(self.socket.receivedMsgs):

            if self.socket.receivedMsgs[i] == "!REGISTEREDSUCCESFULLY":

                self.registered = True

                self.socket.receivedMsgs.pop(i)

            elif self.socket.receivedMsgs[i] == "!ANERROROCCURRED":

                self.\_\_submitButtonPressed = False

                self.\_\_submitButton.clicked = False

                self.socket.receivedMsgs.pop(i)

            else:

                i += 1

    def \_HandleInputs(self):

        super().\_HandleInputs()

*#Need to check for clicks, tab or keys being pressed*

        i = 0

        while i < len(self.\_inputHandler.inputsList):

            if self.\_inputHandler.inputsList[i][:6] == "CLICK:":

                clickLocation = self.\_inputHandler.inputsList[i][6:].split(",")

                clickLocation = (int(clickLocation[0]), int(clickLocation[1]))

                for box in self.\_listOfBoxObjects:

*#If clicked on box*

                    if box.CheckForCollisionWithMouse(clickLocation):

                        box.SetActive()

                    else:

                        box.SetInactive()

                for button in self.\_listOfButtonObjects:

                    if button.CheckForCollision(clickLocation):

                        button.clicked = True

                self.\_inputHandler.inputsList.pop(i)

*#Adds letters to textbox*

            elif self.\_inputHandler.inputsList[i][:3] == "KD\_":

                key = self.\_inputHandler.inputsList[i][3:]

*#Checks if the key is a letter and not a special character*

                if key.isalpha():

                    for box in self.\_listOfBoxObjects:

                        if box.isActive:

                            box.AddLetter(key)

                self.\_inputHandler.inputsList.pop(i)

*#Switches active textbox*

            elif self.\_inputHandler.inputsList[i] == "TABDOWN":

                if self.\_\_usernameBox.isActive:

                    self.\_\_usernameBox.SetInactive()

                    self.\_\_passwordBox.SetActive()

                elif self.\_\_passwordBox.isActive:

                    self.\_\_passwordBox.SetInactive()

                    self.\_\_confirmBox.SetActive()

                elif self.\_\_confirmBox.isActive:

                    self.\_\_confirmBox.SetInactive

                    self.\_\_usernameBox.SetActive()

                self.\_inputHandler.inputsList.pop(i)

            else:

                i += 1

## Server

### Password hashing

If the username entered is not already in the database, the server will generate a hash for the password and put that in the database. From my research, using the bcrypt module allows this quite easily, and also allows for checking if a given plaintext encoded bytes object is the same as a given hash.

The server will then respond with creation successful, and the user will be logged in. The logging in of the user is done server side as an attribute for the Player object is set to True. Whenever this happens, the server gets the data for this user from the database and stores it in the object. This could cause memory issues in the future, but it could also be changed quite easily by simply making it so the data is retrieved when needed.

If the creation is successful, the player will go to the main menu.

#### Pseudocode

IMPORT bcrypt

SUBROUTINE CheckPw(password, hashedPassword)

IF bcrypt.checkpw(password, hashedPassword) THEN RETURN True

ELSE RETURN False

ENDIF

ENDSUBROUTINE

SUBROUTINE GenHash(password)

RETURN bcrypt.hashpw(password, bcrypt.gensalt())

ENDSUBROUTINE

This is pretty much the whole thing, except when making this into code I found out the password needs to be encoded. This meant that the I needed to use the string.encode() method on password.

#### Python code

import bcrypt

def CheckPW(password, hashedPassword):

    if bcrypt.checkpw(password.encode("utf-8"), hashedPassword):

        return True

    else:

        return False

def GenHash(password):

    return bcrypt.hashpw(password.encode("utf-8"), bcrypt.gensalt())

#### Testing

//todo

### Handling account creation request

#### Pseudocode

IF message [:10] = "!REGISTER:" THEN

details <-- split message

player.username <-- username from details

TRY

self.dbHandler.CreateNewUser(username, password)

data <-- self.dbHandler.LoadUser(player)

player.msgsToSend.Enqueue("!REGISTEREDSUCCESFULLY")

player.msgsToSend.Enqueue(data)

EXCEPT

player.msgsToSend.Enqueue("!ANERROROCCURRED")

ENDIF

There is a key difference between this and the code and that is the use of the pickle module. The pickle module allows for sending of objects across sockets, which is what is needed here as data is a list of data which is returned by the database.

#### Python code

elif message[:10] == "!REGISTER:":

    details = message[10:].split(",")

    player.username = details[0]

    try:

        self.dbHandler.CreateNewUser(details[0], details[1])

        data = self.dbHandler.LoadUser(player)

        player.msgsToSend.Enqueue("!REGISTEREDSUCCESFULLY")

        player.msgsTosend.Enqueue(pickle.dumps(data))

    except:

        player.msgsToSend.Enqueue("!ANERROROCCURRED")

The reason this is an elif is that this is in the HandleMessagesForPlayerNotInQueue method of the server.

#### Testing

//todo

## Game

The game object need to run the main method of the registration screen object when the player has pressed the register button in the login screen.

### Pseudocode

For deciding what loop to go to:

WHILE NOT self.\_\_loggedIn

self.\_\_Login()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.loginscreen.userWantsToCreateAccount THEN

self.\_\_Register()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.\_\_registerScreen.registered THEN self.loggedIn <-- True

ENDIF

ENDIF

ENDWHILE

### Python code

while not self.\_\_loggedIn:

*#Goes to loginscreen*

    self.\_\_Login()

    if self.\_\_userQuit:

        return 0

    elif self.\_\_loginScreen.userWantsToCreateAccount:

        self.\_\_Register()

        if self.\_\_userQuit:

            return 0

        if self.\_\_registerScreen.registered:

            self.\_\_loggedIn = True

Register():

def \_\_Register(self):

*#When user wants to create account*

    self.\_\_registerScreen = RegisterScreen(self.\_\_window, self.\_\_resolution, self.socket)

    while not self.\_\_registerScreen.backbuttonPressed and not self.\_\_registerScreen.registered:

        self.\_\_registerScreen.main()

        if self.\_\_registerScreen.userQuit:

            self.\_\_userQuit = True

            return 0

        pygame.display.update()

# Login Screen

The player will need to be able to login to the system. This is done by having 2 InputBox objects which are like the TextBox objects, but do not have the feature of having background text.

I made the following design for this menu:

Graphical user interface

Description automatically generated

This design later got updated to have a register button, which will be how the account creation screen is accessed.

The username and password boxes will need to be able to be selected and change colour, and the continue button needs to send the data to the server for verification.

## Pseudocode

CLASS

SUBROUTINE init(self, window, resolution, socket <-- None)

inherit from Scene

self.loggedIn <-- False

self.userWantsToCreateAccount <-- False

self.\_\_detailsSent <-- False

inputBoxFont <-- pygame Font object

#Need 2 inputboxes, 2 text and 1 button

inputBoxSize <-- (625, 60)

usernameBoxLocation <-- centred and 400 pixels down

passwordBoxLocation <-- centred and 540 pixels down

usernameRect <-- pygame Rect object

passwordRect <-- pygame Rect object

self.\_\_usernameBox <-- InputBox object

self.\_\_passwordBox <-- InputBox object

#Button size for continue and register buttons

buttonSize <-- (400,60)

continueButtonLocation <-- centred and 680 pixels down

continueButtonRect <-- pygame Rect object

self.\_\_continueButton <-- Button object

#Need register button same size as continue button

registerButtonLocation <-- (1400,950)

resgisterButtonRect <-- pygame Rect object

self.\_\_registerButton <-- Button

#Text needs to be 5 pixels to the right of the corresponding box and needs to be 25 pixels above (so 25 pixels and the height of the text itself)

usernameTextSize <-- size of "Username" when it is rendered

usernameTextLocation <-- slightly above usernameBox

self.\_\_usernameText <-- Text object

passwordTextSize <-- size of "Password" when it is rendered

passwordTextLocation <-- slightly above passwordBox

self.\_\_passwordText <-- Text object

#These lists are used to render things on the screen, they are iterated through

self.\_listOfBoxObjects <-- [self.\_\_usernameBox, self.\_\_passwordBox]

self.\_listOfButtonObjects <-- [self.\_\_continueButton, self.\_\_registerButton]

self.\_listOfTextObjects <-- [self.\_\_usernameText, self.\_\_passwordText]

ENDSUBROUTINE

SUBROUTINE main(self)

inherit from Scene

IF self.\_\_continueButton.clicked THEN

IF NOT self.\_\_detailsSent THEN

username <-- self.usernameBox.text

password <-- self.\_\_passwordBox.text

send login message to server

self.\_\_detailsSent <-- True

ELSE

self.\_\_HandleMessages()

ENDIF

ENDIF

ENDSUBROUTINE

SUBROUTINE \_\_HandleMessages(self)

unusedMessages <-- []

WHILE self.socket.receivedMsgs != []

message <-- self.socket.receivedMsgs.POP()

IF message = "!PASSWORDCORRECT" OR message = "ALREADYLOGGEDIN" THEN

OUTPUT "Logged in"

self.loggedIn <-- True

ELSE IF message = "!PASSWORDINCORRECT" or message = "!USERNAMENOTFOUND" THEN

self.\_\_continueButton.clicked <-- True

self.\_\_detailsSent <-- False

self.\_\_usernameBox.text <-- ""

self.\_\_passwordBox.text <-- ""

self.\_\_continueButton.SetText("Continue")

ENDIF

ENDWHILE

FOR box in self.\_listOfBoxObjects

box.UpdateRender()

ENDFOR

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

inherit from Scene

IF NOT self.\_\_continueButton.clicked THEN

i <-- 0

WHILE i < LEN(self.\_inputHandler.inputsList)

IF input is click THEN

IF button is continue THEN

self.\_\_continueButton.clicked <-- True

self.\_\_continueButton.SetText("Checking...")

self.\_inputHandler.inputsList.POP(i)

ELSE IF button is register THEN

self.\_\_registerButton.clicked <-- True

self.userWantsToCreateAccount <-- True

self.\_inputHandler.inputsList.POP(i)

ELSE IF click is on box THEN

box.SetActive()

self.\_inputHandler.inputsList.POP(i)

ELSE

box.SetInactive()

self.\_inputHandler.inputsList.POP(i)

ENDIF

ELSE IF input is keypress THEN

IF box is active THEN

box.AddLetter(keypress)

ENDIF

self.\_inputHandler.inputsList.POP(i)

ELSE IF input is tab key THEN

set next box to active and set every other box to inactive

self.\_inputHandler.inputsList.POP(i)

ELSE IF input is return key THEN

self.\_\_continueButton.clicked <-- True

self.\_\_continueButton.SetText("Checking...")

self.\_inputHandler.inputsList.POP(i)

ELSE

i += 1

ENDIF

ENDWHILE

ENDIF

ENDSUBROUTINE

ENDCLASS

## Python code

*#Displays username and password input boxes*

class LoginScreen(Scene):

    def \_\_init\_\_(self, window, resolution, socket=None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        self.loggedIn = False

        self.userWantsToCreateAccount = False

        self.\_\_detailsSent = False

        inputBoxFont = pygame.font.SysFont("Courier New", int(36 \* self.\_resolution[1]))

*#Need 2 inputboxes, 2 text and 1 button*

        inputBoxSize = (int(625 \* self.\_resolution[0]), int(60 \* self.\_resolution[1]))

*#Button needs to be centred and 400 pixels down*

        usernameBoxLocation = (int((self.\_resolution[0] \* 1920 - inputBoxSize[0]) / 2), int(400 \* self.\_resolution[1]))

        passwordBoxLocation = (int((self.\_resolution[0] \* 1920 - inputBoxSize[0]) / 2), int(540 \* self.\_resolution[1]))

        usernameRect = pygame.Rect(usernameBoxLocation[0], usernameBoxLocation[1], inputBoxSize[0], inputBoxSize[1])

        passwordRect = pygame.Rect(passwordBoxLocation[0], passwordBoxLocation[1], inputBoxSize[0], inputBoxSize[1])

        self.\_\_usernameBox = InputBox(usernameRect, inputBoxFont, self.\_resolution, (40,40,40), (25,25,25), (255,255,255), "")

        self.\_\_passwordBox = InputBox(passwordRect, inputBoxFont, self.\_resolution, (40,40,40), (25,25,25), (255,255,255), "", hashed=True)

*#Button size for continue and register buttons*

        buttonSize = (400 \* self.\_resolution[0], 60 \* self.\_resolution[1])

*#Button needs to be centred and 680 pixels down*

        continueButtonLocation = (int((self.\_resolution[0] \* 1920 - buttonSize[0]) / 2), int(680 \* self.\_resolution[1]))

        continueButtonRect = pygame.Rect(continueButtonLocation[0], continueButtonLocation[1], buttonSize[0], buttonSize[1])

        self.\_\_continueButton = Button(continueButtonRect, (40,40,40), (25,25,25), (255,255,255), text="Continue")

*#Need register button same size as continue button*

        registerButtonLocation = (int(1400 \* self.\_resolution[0]), int(950 \* self.\_resolution[1]))

        resgisterButtonRect = pygame.Rect(registerButtonLocation, buttonSize)

        self.\_\_registerButton = Button(resgisterButtonRect, (40,40,40), (25,25,25), (255,255,255), text="Register")

*#Text needs to be 5 pixels to the right of the corresponding box and needs to be 25 pixels above (so 25 pixels and the height of the text itself)*

        usernameTextSize = inputBoxFont.size("Username")

        usernameTextLocation = (int(usernameBoxLocation[0] + 5 \* self.\_resolution[0]), int(usernameBoxLocation[1] - 25 \* self.\_resolution[1] - usernameTextSize[1]))

        self.\_\_usernameText = Text(inputBoxFont, text="Username", location=usernameTextLocation)

        passwordTextSize = inputBoxFont.size("Password")

        passwordTextLocation = (int(passwordBoxLocation[0] + 5 \* self.\_resolution[0]), int(passwordBoxLocation[1] - 25 \* self.\_resolution[1] - passwordTextSize[1]))

        self.\_\_passwordText = Text(inputBoxFont, text="Password", location=passwordTextLocation)

*#These lists are used to render things on the screen, they are iterated through*

        self.\_listOfBoxObjects = [self.\_\_usernameBox, self.\_\_passwordBox]

        self.\_listOfButtonObjects = [self.\_\_continueButton, self.\_\_registerButton]

        self.\_listOfTextObjects = [self.\_\_usernameText, self.\_\_passwordText]

    def main(self):

        super().main()

        if self.\_\_continueButton.clicked:

            if not self.\_\_detailsSent:

                username = self.\_\_usernameBox.text

                password = self.\_\_passwordBox.text

                self.socket.msgsToSend.append(f"!LOGIN:{username},{password}")

                self.\_\_detailsSent = True

            else:

                self.\_\_HandleMessages()

    def \_\_HandleMessages(self):

        unusedMessages = []

        while len(self.socket.receivedMsgs) != 0:

            message = self.socket.receivedMsgs.pop()

            if message == "!PASSWORDCORRECT" or message == "!ALREADYLOGGEDIN":

                print("Logged in")

                self.loggedIn = True

            elif message == "!PASSWORDINCORRECT" or message == "!USERNAMENOTFOUND":

                self.\_\_continueButton.clicked = False

                self.\_\_detailsSent = False

                self.\_\_usernameBox.text = ""

                self.\_\_passwordBox.text = ""

                self.\_\_continueButton.SetText("Continue")

        for box in self.\_listOfBoxObjects:

            box.UpdateRender()

    def \_HandleInputs(self):

        super().\_HandleInputs()

        if not self.\_\_continueButton.clicked:

            i = 0

            while i < len(self.\_inputHandler.inputsList):

                if self.\_inputHandler.inputsList[i][:6] == "CLICK:":

                    clickLocation = self.\_inputHandler.inputsList[i][6:].split(",")

                    clickLocation = (int(clickLocation[0]), int(clickLocation[1]))

                    if self.\_\_continueButton.CheckForCollision(clickLocation):

                        self.\_\_continueButton.clicked = True

                        self.\_\_continueButton.SetText("Checking...")

                    elif self.\_\_registerButton.CheckForCollision(clickLocation):

                        self.\_\_registerButton.clicked = True

                        self.userWantsToCreateAccount = True

                    else:

                        for box in self.\_listOfBoxObjects:

                            if box.CheckForCollisionWithMouse(clickLocation):

                                box.SetActive()

                            else:

                                box.SetInactive()

                    self.\_inputHandler.inputsList.pop(i)

                elif self.\_inputHandler.inputsList[i][:3] == "KD\_":

                    key = self.\_inputHandler.inputsList[i][3:]

                    for box in self.\_listOfBoxObjects:

                        if box.isActive:

                            box.AddLetter(key)

                    self.\_inputHandler.inputsList.pop(i)

*#Sets the other box to be active than the one that is active*

                elif self.\_inputHandler.inputsList[i] == "TABDOWN":

                    if self.\_\_usernameBox.isActive:

                        self.\_\_usernameBox.SetInactive()

                        self.\_\_passwordBox.SetActive()

                    elif self.\_\_passwordBox.isActive:

                        self.\_\_usernameBox.SetActive()

                        self.\_\_passwordBox.SetInactive()

                    self.\_inputHandler.inputsList.pop(i)

                elif self.\_inputHandler.inputsList[i] == "RETURNDOWN":

                    self.\_\_continueButton.clicked = True

                    self.\_\_continueButton.SetText("Checking...")

                    self.\_inputHandler.inputsList.pop(i)

                elif self.\_inputHandler.inputsList[i] == "RETURNUP":

*#Returnup shouldnt be added back to queue*

                    self.\_inputHandler.inputsList.pop(i)

                else:

                    i += 1

## Server

The server will check if the username exists in the database and then check if the password is correct. If it is correct then it will load the user using the LoadUser() method in DatabaseHandler.

The code and pseudocode below will be ELSE IF and elif because they are from the section above where the pseudocode and code for the server was written with the UML diagram for the server.

### Pseudocode

ELSE IF message[:7] = "!LOGIN:" THEN

IF NOT player.loggedIn THEN

details <-- split message with ","

username <-- details[0]

password <-- details[1]

IF self.dbHandler.CheckIfUsernameInDB(username) THEN

IF self.dbHandler.CheckPassword(username, password) THEN

player.loggedIn <-- True

player.username <-- username

self.dbHandler.LoadUser(player)

player.msgsToSend.Enqueue("!PASSWORDCORRECT")

ELSE

player.msgsToSend.Enqueue("!PASSWORDINCORRECT")

ENDIF

ELSE

player.msgsToSend.Enqueue("!USERNAMENOTFOUND")

ENDIF

ELSE

player.msgsToSend.Enqueue("!ALREADYLOGGEDIN")

ENDIF

### Python code

elif message[:7] == "!LOGIN:":

    if not player.loggedIn:

        details = message[7:].split(",")

        username = details[0]

        password = details[1]

*#Checks password and sends message to user to confirm if they are signed in or not*

        if self.dbHandler.CheckIfUsernameInDB(username):

            if self.dbHandler.CheckPassword(username, password):

                player.loggedIn = True

                player.username = username

                self.dbHandler.LoadUser(player)

                player.msgsToSend.Enqueue("!PASSWORDCORRECT")

            else:

                player.msgsToSend.Enqueue("!PASSWORDINCORRECT")

        else:

            player.msgsToSend.Enqueue("!USERNAMENOTFOUND")

    else:

        player.msgsToSend.Enqueue("!ALREADYLOGGEDIN")

## Game

The game object needs to run the login screen main method while the player is not logged in:

### Pseudocode

Checking for what screen to go to:

WHILE NOT self.\_\_loggedIn

self.\_\_Login()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.loginscreen.userWantsToCreateAccount THEN

self.\_\_Register()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.\_\_registerScreen.registered THEN self.loggedIn <-- True

ENDIF

ENDIF

ENDWHILE

### Python code

while not self.\_\_loggedIn:

*#Goes to loginscreen*

    self.\_\_Login()

    if self.\_\_userQuit:

        return 0

    elif self.\_\_loginScreen.userWantsToCreateAccount:

        self.\_\_Register()

        if self.\_\_userQuit:

            return 0

        if self.\_\_registerScreen.registered:

            self.\_\_loggedIn = True

Login():

def \_\_Login(self):

*#Before user has logged in*

    self.\_\_loginScreen = LoginScreen(self.\_\_window, self.\_\_resolution, self.socket)

    while not self.\_\_loggedIn and not self.\_\_loginScreen.userWantsToCreateAccount:

        if self.\_\_loginScreen.loggedIn:

            self.\_\_loggedIn = True

        else:

            self.\_\_loginScreen.main()

            if self.\_\_loginScreen.userQuit:

                self.\_\_userQuit = True

                return 0

            pygame.display.update()

# Main Menu

The game needs a main menu where the user makes a choice on what screen to go to next. As usual, this will be a Scene subclass called MainMenu. I have made a design for it in paint.net below:

Graphical user interface, application

Description automatically generated

There needs to be 4 buttons and this scene needs to have a way of returning the choice. This choice gets used in Game to determine what loop to go to from there.

## Pseudocode

CLASS MainMenu(Scene)

SUBROUTINE init(self, window, resolution, socket)

inherit from Scene

#This is how this object gives a value to Game object for userChoice

self.userChoice <-- None

colourActive <-- (40,40,40)

colourInactive <-- (25,25,25)

textColour <-- (255,255,255)

self.\_\_titleFont <-- pygame Font object

boxSize <-- 2/5 the screen width and 150 pixels tall

boxX <-- correct x value so that it is centred

playButtonRect <-- pygame Rect object

statisticsButtonRect <-- pygame Rect object

settingsButtonRect <-- pygame Rect object

quitButtonRect <-- pygame Rect object

self.\_\_playButton <-- Button object

self.\_\_statisticsButton <-- Button object

self.\_\_settingsButton <-- Button object

self.\_\_quitButton <-- Button object

self.\_listOfButtonObjects <-- [self.\_\_playButton, self.\_\_statisticsButton, self.\_\_settingsButton, self.\_\_quitButton]

titleSize <-- size of "SpeedTyper" when rendered with self.\_\_titleFont

titleLocation <-- centred and on y = 160

self.\_\_titleText <-- Text object

ENDSUBROUTINE

SUBROUTINE main(self)

inherit from Scene

FOR button IN self.\_listOfButtonObjects

IF button.clicked THEN

IF button.text = "Quit" THEN

self.userQuit <-- True

ELSE

self.userChoice <-- button.text

ENDIF

ENDIF

ENDFOR

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

inherit from Scene

i <-- 0

WHILE i < LEN(self.\_inputHandler.inputsList)

IF self.\_inputHandler.inputsList[i][:6] = "CLICK:" THEN

FOR button IN self.\_listOfButtonObjects

IF button is clicked THEN

button.clicked <-- True

ENDIF

ENDFOR

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

ENDCLASS

## Python code

*#Displays main menu for user to make choice what to see*

class MainMenu(Scene):

    def \_\_init\_\_(self, window, resolution, socket=None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        self.userChoice = None

        colourActive = (40,40,40)

        colourInactive = (25,25,25)

        textColour = (255,255,255)

        self.\_\_titleFont = pygame.font.SysFont("Calibri", int(140 \* self.\_resolution[1]))

        boxSize = (int(2/5 \* self.\_resolution[0] \* 1920), int(150 \* self.\_resolution[1]))

        boxX = int((self.\_resolution[0] \* 1920 - boxSize[0]) / 2)

*#Needs 4 buttons and a title text*

        playButtonRect = pygame.Rect(boxX, int(320 \* self.\_resolution[1]), boxSize[0], boxSize[1])

        statisticsButtonRect = pygame.Rect(boxX, int(500 \* self.\_resolution[1]), boxSize[0], boxSize[1])

        settingsButtonRect = pygame.Rect(boxX, int(680 \* self.\_resolution[1]), boxSize[0], boxSize[1])

        quitButtonRect = pygame.Rect(boxX, int(860 \* self.\_resolution[1]), boxSize[0], boxSize[1])

        self.\_\_playButton = Button(playButtonRect, colourActive, colourInactive, textColour, text="Play")

        self.\_\_statisticsButton = Button(statisticsButtonRect, colourActive, colourInactive, textColour, text="Statistics")

        self.\_\_settingsButton = Button(settingsButtonRect, colourActive, colourInactive, textColour, text = "Settings")

        self.\_\_quitButton = Button(quitButtonRect, colourActive, colourInactive, textColour, text="Quit")

        self.\_listOfButtonObjects = [self.\_\_playButton, self.\_\_statisticsButton, self.\_\_settingsButton, self.\_\_quitButton]

*#Need title text*

        titleSize = self.\_\_titleFont.size("SpeedTyper")

        titleLocation = (int((self.\_resolution[0] \* 1920 - titleSize[0]) / 2), int(160 \* self.\_resolution[1]))

        self.\_\_titleText = Text(self.\_\_titleFont, text="SpeedTyper", location=titleLocation, colour=(255,165,0))

        self.\_listOfTextObjects = [self.\_\_titleText]

    def main(self):

        super().main()

        for button in self.\_listOfButtonObjects:

            if button.clicked:

                if button.text == "Quit":

                    self.userQuit = True

                else:

                    self.userChoice = button.text

    def \_HandleInputs(self):

        super().\_HandleInputs()

        i = 0

        while i < len(self.\_inputHandler.inputsList):

            if self.\_inputHandler.inputsList[i][:6] == "CLICK:":

                clickLocation = self.\_inputHandler.inputsList[i][6:].split(",")

                clickLocation = (int(clickLocation[0]), int(clickLocation[1]))

                for button in self.\_listOfButtonObjects:

                    if button.CheckForCollision(clickLocation):

                        button.clicked = True

                self.\_inputHandler.inputsList.pop(i)

            else:

                i += 1

## Server

The server will need to accept a message from the user when the user is trying to queue into the game when they press play. The server also needs to disconnect the player when they press the quit button, and the server also needs to return the user’s statistics when they press on the statistics button.

### Pseudocode

For this section I will include the pseudocode here and the python code for it after.

When the player tries to queue for a game, the client will send the message “!QUEUE” to the server (this happens in the Game object in the WaitForGame method). This will let the server know that it needs to move them from the players list to the playersInMatchmaking list.

A snippet of the pseudocode from the HandleMessagesForPlayersNotInQueue is below:

ELSE IF player.loggedIn AND message = "!QUEUE" THEN

playersQuit.APPEND(player)

self.playersInMatchmaking.APPPEND(player)

player.timeWaited <-- 0

When the player presses the statistics button, the client will send “!STATISTICS” to the server. The server should see this and respond with the user’s statistics, which are then dealt with in the statistics screen. The part of the server that is responsible for this is below:

ELSE IF message = "!STATISTICS" THEN

data <-- self.dbHandler.LoadUser(player)

player.msgsToSend.Enqueue(data)

OUTPUT "Data sent"

LoadUsers is in Database handler and will return a list of the user’s data that is retrieved from the database, there is no pseudocode for the database handler, though I will include the python code for it in the python code section below.

### Python code

Handling message from player wanting to queue:

elif player.loggedIn and message == "!QUEUE":

    playersQuit.append(player)

    self.playersInMatchmaking.append(player)

    player.timeWaited = 0

Handling of statistics:

elif message == "!STATISTICS":

    data = self.dbHandler.LoadUser(player)

    player.msgsToSend.Enqueue(pickle.dumps(data))

    print("Pickle sent")

I have previously discussed the pickle module in the account creation section.

Loading users:

def LoadUser(self, player : Player):

    self.Open()

    params = (player.username,)

    command = """

    SELECT \* FROM Users

    WHERE Username = (?)

    """

    self.\_\_cursor.execute(command, params)

    data = self.\_\_cursor.fetchall()[0]

    player.wordsTyped = int(data[2])

    player.timePlayed = int(data[3])

    player.Elo = int(data[4])

    player.highestElo = int(data[5])

    player.gamesWon = int(data[6])

    player.gamesPlayed = int(data[7])

    player.longestStreak = int(data[8])

    player.largestWinMargin = float(data[9])

    player.lettersTyped = int(data[10])

    player.lettersTypedCorrectly = int(data[11])

    player.sumOfOpponentsElo = int(data[12])

    player.currentWinstreak = int(data[13])

    self.Close()

    return data

## Game

The game object needs to get an input from the user through the login screen and then send them to a different method based on that choice:

### Pseudocode

This is where the main game loop happens, where the player choose an option in the menu, goes to that option and then returns to the menu.

WHILE True

self.\_\_mainMenu <-- MainMenu object

self.\_\_matchmakingScreen <-- MatchmakingScreen object

self.\_\_timerScene <-- TimerScene object

self.\_\_timerStarted <-- False

self.\_\_textToWrite <-- None

self.\_\_results <-- None

self.\_\_margin <-- None

self.\_\_ELodiff <-- None

self.\_\_stats <-- None

self.\_\_GetChoiceFromUser()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

userChoice <-- self.\_\_mainMenu.userChoice

### Python code

while True:

    self.\_\_mainMenu = MainMenu(self.\_\_window, self.\_\_resolution, self.socket)

    self.\_\_matchmakingScreen = MatchmakingScreen(self.\_\_window, self.\_\_resolution, self.socket)

    self.\_\_timerScene = TimerScene(self.\_\_window, self.\_\_resolution, self.socket)

    self.\_\_timerStarted = False

    self.\_\_textToWrite = None

    self.\_\_results = None

    self.\_\_margin = None

    self.\_\_ELodiff = None

    self.\_\_stats = None

*#Goes to main menu to get choice from user*

    self.\_\_GetChoiceFromUser()

    if self.\_\_userQuit:

        return 0

    userChoice = self.\_\_mainMenu.userChoice

GetChoiceFromUser():

def \_\_GetChoiceFromUser(self):

*#Waits for choice from user*

    while self.\_\_mainMenu.userChoice is None:

        self.\_\_mainMenu.main()

        if self.\_\_mainMenu.userQuit:

            self.\_\_userQuit = True

            return 0

        pygame.display.update()

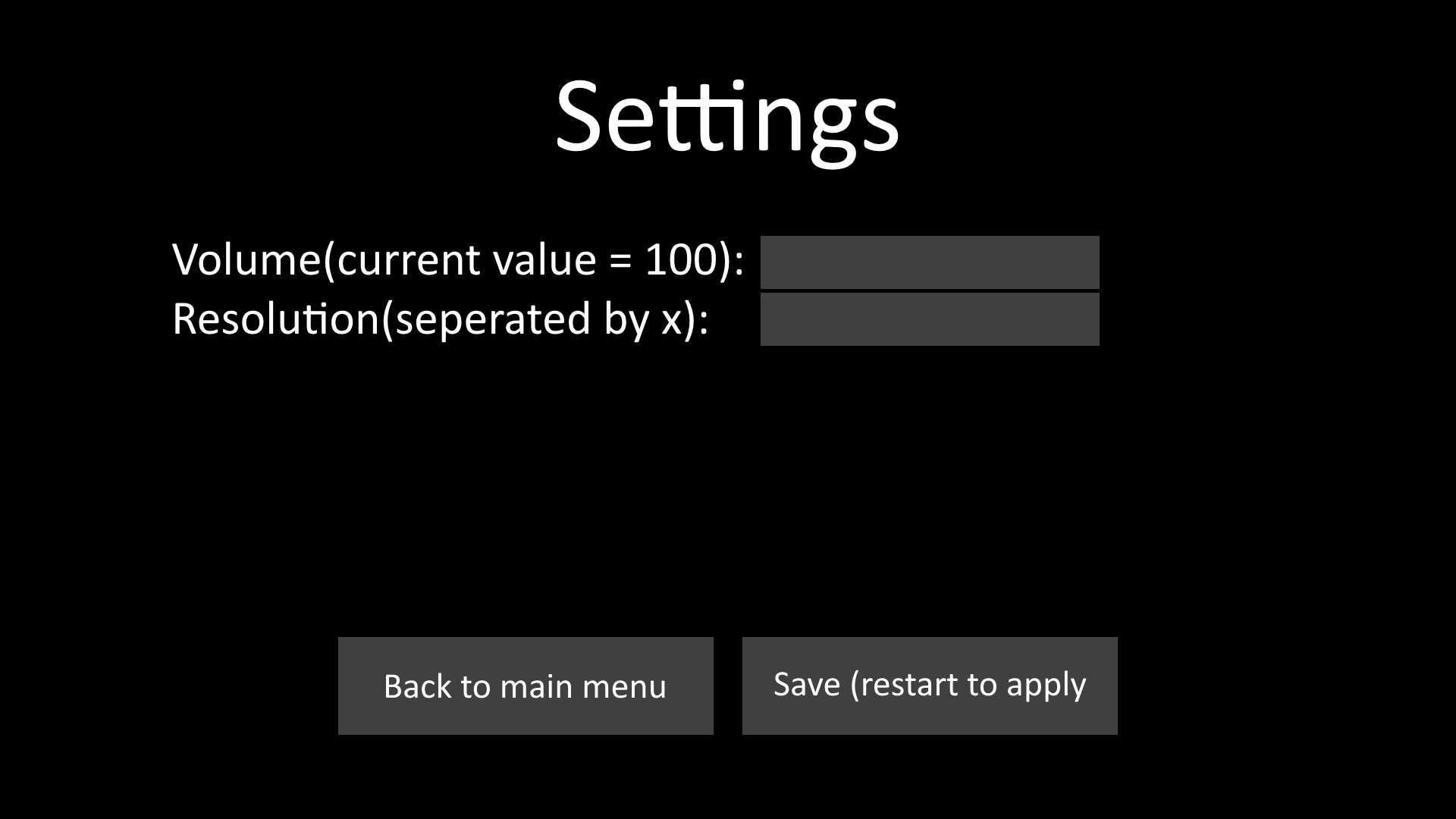
# Settings screen

The settings screen is yet another scene that is used to display information and take information from the user. This was one of the easiest scenes to make as it does not need any communication with the server.

The settings screen needs to simply show the options stated in the objective and allow the user to change them. The options were:



I made a quick design to make this using paint.net:



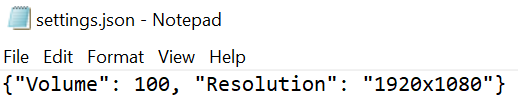
This will be quite easy to implement as all it requires is 2 button objects, 2 input box objects and 3 text objects.

After making the settings screen I just need to store the data and use it when the game launches. This can be done with json files which are easy to use due to the built in json module.

The json module has 2 methods I will be using which are json.loads and json.dumps, which will load from json and save to json respectively.

## Json file

The json file looks like the following:



## Pseudocode

CLASS SettingsScreen(Scene)

SUBROUTINE init(self, window, resolution, settings, socket)

inherit from Scene

self.\_\_settings <-- settings

self.backbuttonPressed <-- False

textColour <-- (255,255,255)

activeColour <-- (40,40,40)

inactiveColour <-- (25,25,25)

#Settings title text

titleFont <-- pygame Font object

titleSize <-- size of "Settings" when rendered

titleLocation <-- centred on x and y <-- 140

self.\_\_titleText <-- Text object

infoTextFont <-- pygame Font object

#Volume text

volumeLocation <-- (255, 315)

self.\_\_volumeText <-- Text object

#Resolution text

resLocation <-- (235, 390)

self.\_\_resolutionText <-- Text object

self.\_listOfTextObjects <-- [self.\_\_titleText, self.\_\_volumeText, self.\_\_resolutionText]

#Boxes

boxSize <-- (450, 60)

boxTextFont <-- pygame Font object

volumeBoxLocation <-- (1000,305)

volumeRect <-- pygame Rect object

self.\_\_volumeBox <-- InputBox object

resBoxLocation <-- (1000,385)

resRect <-- pygame Rect object

self.\_\_resBox <-- InputBox object

self.\_listOfBoxObjects <-- [self.\_\_volumeBox, self.\_\_resBox]

#Buttons

buttonSize <-- (500,130)

backButtonLocation <-- (446,840)

backButtonRect <-- pygame Rect object

self.\_\_backButton <-- Button object

saveButtonLocation <-- (980,840)

saveButtonRect <-- pygame Rect object

self.\_\_saveButton <-- Button object

self.\_listOfButtonObjects <-- [self.\_\_backButton, self.\_\_saveButton]

ENDSUBROUTINE

SUBROUTINE main(self)

Inherit from Scene

IF self.\_\_saveButton.clicked THEN

TRY

newVolume <-- STRING\_TO\_INT(self.\_\_volumeText.text)

IF newVolume > 100 THEN

newVolume <-- 100

ELSE IF newVolume < 0 THEN

newVolume <-- 0

ENDIF

EXCEPT

newVolume <-- self.\_\_settings["Volume"]

TRY

newRes <-- split self.\_\_resBox.text with "x"

STRING\_TO\_INT(newRes[0])

STRING\_TO\_INT(newRes[1])

newRes <-- newRes[0] + "x" + newRes[1]

EXCEPT

newRes <-- self.\_\_settings["Resolution"]

self.\_\_settings["Volume"] <-- newVolume

self.\_\_settings["Resolution"] <-- newRes

turn self.\_\_settings to json

save self.\_\_settings to file

ENDIF

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

inherit from Scene

i <-- 0

WHILE i < LEN(self.\_inputHandler.inputsList)

IF input is a click THEN

IF click location is in a box THEN

set box to active

ELSE

set box to inactive

ENDIF

IF click location is in a button THEN

set button to clicked

ENDIF

remove input from list of inputs

ELSE IF input is a keypress THEN

IF key is a normal letter THEN

IF a box is active THEN

add letter to active box

ENDIF

ENDIF

remove input from list of inputs

ELSE IF input is the tab key THEN

change activetext to the next one down or the first if the active box is the bottom one

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

ENDCLASS

## Python code

class SettingsScreen(Scene):

    def \_\_init\_\_(self, window, resolution, settings, socket: ClientSocket = None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        self.\_\_settings = settings

        self.backButtonPressed = False

        textColour = (255,255,255)

        activeColour = (40,40,40)

        inactiveColour = (25,25,25)

*#Settings title text*

        titleFont = pygame.font.SysFont("Calibri", int(108 \* self.\_resolution[1]))

        titleSize = titleFont.size("Settings")

        titleLocation = (int((self.\_resolution[0] \* 1920 - titleSize[0]) / 2), int(105 \* self.\_resolution[1]))

        self.\_\_titleText = Text(titleFont, text="Settings", location=titleLocation)

        infoTextFont = pygame.font.SysFont("Calibri", int(48 \* self.\_resolution[1]))

*#Volume text*

        volumeLocation = (int(225 \* self.\_resolution[0]), int(315 \* self.\_resolution[1]))

        self.\_\_volumeText = Text(infoTextFont, text=f'Volume(current value = {self.\_\_settings["Volume"]}):', location=volumeLocation)

*#Resolution text*

        resLocation = (int(235 \* self.\_resolution[0]), int(390 \* self.\_resolution[1]))

        self.\_\_resolutionText = Text(infoTextFont, text="Resolution(seperated by x):", location=resLocation)

        self.\_listOfTextObjects = [self.\_\_titleText, self.\_\_volumeText, self.\_\_resolutionText]

*#Boxes*

        boxSize = (int(450 \* self.\_resolution[0]), int(60 \* self.\_resolution[1]))

        boxTextFont = pygame.font.SysFont("Courier New", int(48 \* self.\_resolution[1]))

        volumeBoxLocation = (int(1000 \* self.\_resolution[0]), int(305 \* self.\_resolution[1]))

        volumeRect = pygame.Rect(volumeBoxLocation, boxSize)

        self.\_\_volumeBox = InputBox(volumeRect, boxTextFont, self.\_resolution, activeColour, inactiveColour, textColour, "")

        resBoxLocation = (int(1000 \* self.\_resolution[0]), int(385 \* self.\_resolution[1]))

        resRect = pygame.Rect(resBoxLocation, boxSize)

        self.\_\_resBox = InputBox(resRect, boxTextFont, self.\_resolution, activeColour, inactiveColour, textColour, "")

        self.\_listOfBoxObjects = [self.\_\_volumeBox, self.\_\_resBox]

*#Buttons*

        buttonSize = (int(500 \* self.\_resolution[0]), int(130 \* self.\_resolution[1]))

        backButtonLocation = (int(446 \* self.\_resolution[0]), int(840 \* self.\_resolution[1]))

        backButtonRect = pygame.Rect(backButtonLocation, buttonSize)

        self.\_\_backButton = Button(backButtonRect, activeColour, inactiveColour, textColour, text="Back to main menu")

        saveButtonLocation = (int(980 \* self.\_resolution[0]), int(840 \* self.\_resolution[1]))

        saveButtonRect = pygame.Rect(saveButtonLocation, buttonSize)

        self.\_\_saveButton = Button(saveButtonRect, activeColour, inactiveColour, textColour, text=("Save(restart to apply)"))

        self.\_listOfButtonObjects = [self.\_\_backButton, self.\_\_saveButton]

    def main(self):

        super().main()

        if self.\_\_saveButton.clicked:

            try:

                newVolume = int(self.\_\_volumeBox.text)

                if newVolume > 100:

                    newVolume = 100

                elif newVolume < 0:

                    newVolume = 0

            except:

                newVolume = self.\_\_settings["Volume"]

            try:

                newRes = self.\_\_resBox.text.split("x")

                int(newRes[0])

                int(newRes[1])

                newRes = f"{newRes[0]}x{newRes[1]}"

            except:

                newRes = self.\_\_settings["Resolution"]

            self.\_\_settings["Volume"] = newVolume

            self.\_\_settings["Resolution"] = newRes

            file = open("settings.json", "w")

            jsonsettings = json.dumps(self.\_\_settings)

            file.write(jsonsettings)

            file.close()

            self.\_\_saveButton.clicked = False

    def \_HandleInputs(self):

        super().\_HandleInputs()

        i = 0

        while i < len(self.\_inputHandler.inputsList):

            if self.\_inputHandler.inputsList[i][:6] == "CLICK:":

                clickLocation = self.\_inputHandler.inputsList[i][6:].split(",")

                clickLocation = (int(clickLocation[0]), int(clickLocation[1]))

                if self.\_\_saveButton.CheckForCollision(clickLocation):

                    self.\_\_saveButton.clicked = True

                elif self.\_\_backButton.CheckForCollision(clickLocation):

                    self.\_\_backButton.clicked = True

                    self.backButtonPressed = True

                else:

                    for box in self.\_listOfBoxObjects:

                        if box.CheckForCollisionWithMouse(clickLocation):

                            box.SetActive()

                        else:

                            box.SetInactive()

                self.\_inputHandler.inputsList.pop(i)

            elif self.\_inputHandler.inputsList[i][:3] == "KD\_":

                key = self.\_inputHandler.inputsList[i][3:]

                for box in self.\_listOfBoxObjects:

                    if box.isActive:

                        box.AddLetter(key)

                self.\_inputHandler.inputsList.pop(i)

*#Sets the other box to be active than the one that is active*

            elif self.\_inputHandler.inputsList[i] == "TABDOWN":

                if self.\_\_volumeBox.isActive:

                    self.\_\_volumeBox.SetInactive()

                    self.\_\_resBox.SetActive()

                elif self.\_\_resBox.isActive:

                    self.\_\_volumeBox.SetActive()

                    self.\_\_resBox.SetInactive()

                self.\_inputHandler.inputsList.pop(i)

            else:

                i += 1

As previously mentioned, this does not require anything server side.

## Game

The game object needs to recognise that the user has chosen the settings menu option and will then send the player to the settings screen.

### Pseudocode

ELSE IF userChoice = "Settings" THEN

self.\_\_Settings()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

ENDIF

This is after the main menu once the user has made their choice.

### Python code

elif userChoice == "Settings":

    self.\_\_Settings()

    if self.\_\_userQuit:

        return 0

Settings():

def \_\_Settings(self):

    self.\_\_settingsScreen = SettingsScreen(self.\_\_window, self.\_\_resolution, self.settings, self.socket)

    while not self.\_\_settingsScreen.backButtonPressed:

        self.\_\_settingsScreen.main()

        if self.\_\_settingsScreen.userQuit:

            self.\_\_userQuit = True

            return 0

        pygame.display.update()

# Statistics screen

The statistics screen needs to display the following information as per the objectives:

Text

Description automatically generated with low confidence

The data that I decided to store in the database was specifically to allow these to be displayed. As was shown in the main menu section, the client sends a message to the server asking for the data to be sent back. The data is received in bytes and decoded using the pickle module. Pickle allows me to send python objects along sockets and so fits this task perfectly.

I won’t go into detail about how the server and client communicate on this as it isn’t really necessary due to it being written in the main menu section, but I will include the changes I made to the ClientSocket object to fit with this functionality.

I had to add a try except in the case that a pickle was sent, because in the GetMsgs method of ClientSocket, it will attempt to decode the message. This is done with the string.decode() method which does not work on pickles. This try except simply make it so it appends the pickle to the list as bytes, which also makes checking for the pickle easy later in the Game object’s CheckMessages method.

The updated method is as follows:

def GetMsgs(self):

    self.\_\_client.setblocking(False)

    msgLen = 0

    try:

        msgLen = self.\_\_client.recv(self.\_\_HEADER).decode(self.\_\_FORMAT)

        msgLen = int(msgLen)

        if msgLen > 0:

            self.\_\_client.setblocking(True)

            msg = self.\_\_client.recv(msgLen)

            try:

                msg = msg.decode(self.\_\_FORMAT) *#Waits for a message with length msgLen to be received*

            except:

                pass

            self.receivedMsgs.append(msg)

            print(f"Message Received:{msg}")

    except:

        self.\_\_client.setblocking(True)

The code for CheckMessages is here to show the checking for pickles:

def \_\_CheckMessages(self):

    i = 0

    while i < len(self.socket.receivedMsgs):

*#If new phase started*

        try:

            self.\_\_stats = pickle.loads(self.socket.receivedMsgs[i])

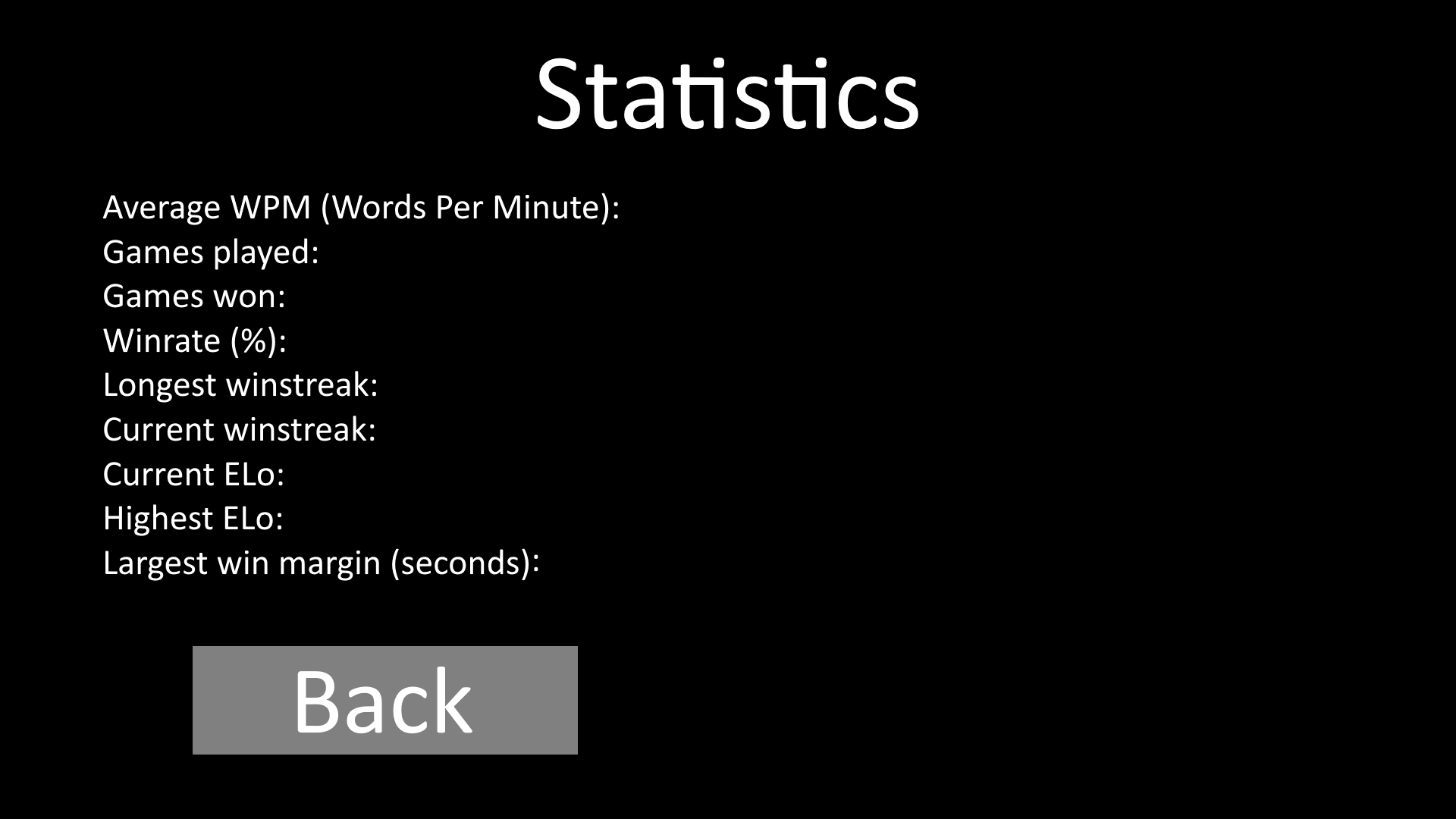
            print("Pickle acquired")

            self.socket.receivedMsgs.pop(i)

        except:

This is followed by checking the rest of the messages but the code above shows the pickle being located.

I made a design of the statistics screen:



The numbers would be on the right side of the screen, this is achieved by using string.format() which allows alignment of text.

Apart from the initial acquisition of the data from the server, there is no further communication between the client and server for this scene.

## Pseudocode

CLASS StatisticsScreen(Scene)

SUBROUTINE init(self, window, resolution, data, socket)

# These are here for reference as to what is in data, it has been copied from the generation of the database

# Username TEXT PRIMARY KEY, 0

# Password BLOB, 1

# WordsTyped INTEGER, 2

# TimePlayed INTEGER, 3

# Elo INTEGER, 4

# HighestElo INTEGER, 5

# GamesWon INTEGER, 6

# GamesPlayed INTEGER, 7

# LongestStreak INTEGER, 8

# LargestWinMargin FLOAT, 9

# LettersTyped INTEGER, 10

# LettersTypedCorrectly INTEGER, 11

# SumOfOpponentsELo INTEGER, 12

# CurrentWinstreak INTEGER 13

#Find WPM

wordsTyped <-- data[2]

timePlayed <-- data[3]

#ROUND will round the first argument to the 2nd argument decimal places

avgWPM <-- ROUND(gamesWon, gamesPlayed \* 100, 2)

#Just data

longestStreak <-- data[8]

currentStreak <-- data[13]

currentELo <-- ROUND(data[4], 2)

highestELo <-- ROUND(data[5], 2)

largestWinMargin <-- STRING\_TO\_INT(data[9])

self.\_\_font <-- pygame Font object

maxTextHeight <-- height of "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

thingsToRender <-- [avgWPM, gamesPlayed, gamesWon, winrate, longestStreak, currentStreak, currentELo, highestELo, largestWinMargin]

textBoxTopLeft <-- (140,260)

maxWidth <-- maximum width that the text render can be

#This works because the font is going to be monospaced

charWidth <-- width of letter "a"

maxTextHeight <-- ROUND(maxWidth / charWidth, 0)

text <-- "Average WPM (Words Per Minute):"

lenText <-- LEN(text)

avgWPM <-- text + formatted data

text <-- "Games played:"

lenText <-- LEN(text)

gamesPlayed <-- text + formatted data

text <-- "Games won:"

lenText <-- LEN(text)

gamesWon <-- text + formatted data

text <-- "Winrate (%):"

lenText <-- LEN(text)

winrate <-- text + formatted data

text <-- "Longest winstreak:"

lenText <-- LEN(text)

longestStreak <-- text + formatted data

text <-- "Current winstreak:"

lenText <-- LEN(text)

currentStreak <-- text + formatted data

text <-- "Current ELo:"

lenText <-- LEN(text)

currentELo <-- text + formatted data

text <-- "Highest ELo:"

lenText <-- LEN(text)

highestELo <-- text + formatted data

text <-- "Largest win margin:"

lenText <-- LEN(text)

largestWinMargin <-- text + formatted data

textToRender <-- [avgWPM, gamesPlayed, gamesWon, winrate, longestStreak, currentStreak, currentELo, highestELo, largestWinMargin]

textX <-- textBoxTopLeft[0]

textY <-- textBoxTopLeft[1]

fill background will colour black

#Render title text

titleFont <-- pygame Font object

titleRender <-- pygame Surface object

titleSize <-- size of titleRender

titleLocation <-- centred x and y = 75

render titleRender on screen

FOR i <-- RANGE(LEN(textToRender))

textRender <-- render textToRender[i] with self.\_\_font

renderLocation <-- (textX, textY + (maxTextHeight + 5) \* i)

draw textRender on screen

ENDFOR

self.backbuttonPressed <-- False

backButtonSize <-- (500,130)

backButtonLocation <-- (250,850)

backRect <-- pygame Rect object

self.\_\_backButton <-- Button object

ENDSUBROUTINE

SUBROUTINE main(self)

self.\_HandleInputs()

IF self.\_\_backButton.CheckForCollision(mouse location) THEN

self.\_\_backButton.SetActive()

ELSE

self.\_\_backButton.SetInactive()

ENDIF

self.\_Render()

ENDSUBROUTINE

SUBROUTINE \_Render(self)

self.\_\_backButton.Render(self.\_window)

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

Inherit from Scene

i <-- 0

WHILE i < LEN(self.\_inputHandler.inputsList)

IF self.\_inputHandler.inputsList[i][:6] = "CLICK:" THEN

clickLocation <-- mouse location

IF self.\_\_backButton.CheckForCollision(clickLocation) THEN

self.backbuttonPressed <-- True

self.\_\_backButton.clicked <-- True

ENDIF

self.\_inputHandler.inputsList.POP(i)

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

ENDCLASS

## Python code

class StatisticsScreen(Scene):

    def \_\_init\_\_(self, window, resolution, data, socket: ClientSocket = None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

*# These are here for reference as to what is in data, it has been copied from the generation of the database*

*# Username TEXT PRIMARY KEY, 0*

*# Password BLOB, 1*

*# WordsTyped INTEGER, 2*

*# TimePlayed INTEGER, 3*

*# Elo INTEGER, 4*

*# HighestElo INTEGER, 5*

*# GamesWon INTEGER, 6*

*# GamesPlayed INTEGER, 7*

*# LongestStreak INTEGER, 8*

*# LargestWinMargin FLOAT, 9*

*# LettersTyped INTEGER, 10*

*# LettersTypedCorrectly INTEGER, 11*

*# SumOfOpponentsELo INTEGER, 12*

*# CurrentWinstreak INTEGER 13*

*#Find WPM*

        wordsTyped = data[2]

        timePlayed = data[3]

        avgWPM = round(wordsTyped / timePlayed, 2)

*#Winrate*

        gamesWon = data[6]

        gamesPlayed = data[7]

        winrate = round(gamesWon / gamesPlayed \* 100, 2)

*#Just data*

        longestStreak = data[8]

        currentStreak = data[13]

        currentELo = round(data[4], 2)

        highestELo = round(data[5], 2)

        largestWinMargin = int(data[9])

        self.\_\_font = pygame.font.SysFont("Courier New", int(36 \* self.\_resolution[1]), bold=True)

        maxTextHeight = self.\_\_font.size("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")[1]

        thingsToRender = [avgWPM, gamesPlayed, gamesWon, winrate, longestStreak, currentStreak, currentELo, highestELo, largestWinMargin]

*#Dimensions for area text is rendered in*

        textBoxTopLeft = (int(140 \* self.\_resolution[0]), int(260 \* self.\_resolution[0]))

        maxWidth = self.\_resolution[0] \* 1920 - 2 \* textBoxTopLeft[0]

        charWidth = self.\_\_font.size("a")[0]

        maxTextWidth = int(maxWidth / charWidth)

        text = "Average WPM (Words Per Minute):"

        lenText = len(text)

        avgWPM = text + "{info:>{width}}".format(info = avgWPM, width = maxTextWidth - lenText)

        text = "Games played:"

        lenText = len(text)

        gamesPlayed = text + "{info:>{width}}".format(info = gamesPlayed, width = maxTextWidth - lenText)

        text = "Games won:"

        lenText = len(text)

        gamesWon = text + "{info:>{width}}".format(info = gamesWon, width = maxTextWidth - lenText)

        text = "Winrate (%):"

        lenText = len(text)

        winrate = text + "{info:>{width}}".format(info = winrate, width = maxTextWidth - lenText)

        text = "Longest winstreak:"

        lenText = len(text)

        longestStreak = text + "{info:>{width}}".format(info = longestStreak, width = maxTextWidth - lenText)

        text = "Current winstreak:"

        lenText = len(text)

        currentStreak = text + "{info:>{width}}".format(info = currentStreak, width = maxTextWidth - lenText)

        text = "Current ELo:"

        lenText = len(text)

        currentELo = text + "{info:>{width}}".format(info = currentELo, width = maxTextWidth - lenText)

        text = "Highest ELo:"

        lenText = len(text)

        highestELo = text + "{info:>{width}}".format(info = highestELo, width = maxTextWidth - lenText)

        text = "Largest win margin:"

        lenText = len(text)

        largestWinMargin = text + "{info:>{width}}".format(info = largestWinMargin, width = maxTextWidth - lenText)

        textToRender = [avgWPM, gamesPlayed, gamesWon, winrate, longestStreak, currentStreak, currentELo, highestELo, largestWinMargin]

        textX = textBoxTopLeft[0]

        textY = textBoxTopLeft[1]

*#Make background black*

        self.\_window.fill((0,0,0))

*#Render title text*

        titleFont = pygame.font.SysFont("Calibri", int(108 \* self.\_resolution[1]))

        titleRender = titleFont.render("Statistics", True, (255,255,255))

        titleSize = titleRender.get\_size()

        titleLocation = (int((self.\_resolution[0] \* 1920 - titleSize[0]) / 2), int(75 \* self.\_resolution[1]))

        self.\_window.blit(titleRender, titleLocation)

*#Render information on screen*

        for i in range(len(textToRender)):

            textRender = self.\_\_font.render(textToRender[i], True, (255,255,255))

            renderLocation = (textX, textY + (maxTextHeight + 5) \* i)

            self.\_window.blit(textRender, renderLocation)

*#For back button*

        self.backButtonPressed = False

        backButtonSize = (int(500 \* self.\_resolution[0]), int(130 \* self.\_resolution[1]))

        backButtonLocation = (int(250 \* self.\_resolution[0]), int(850 \* self.\_resolution[1]))

        backRect = pygame.Rect(backButtonLocation, backButtonSize)

        self.\_\_backButton = Button(backRect, (40,40,40), (25,25,25), (255,255,255), text="Back")

    def main(self):

        self.\_HandleInputs()

        if self.\_\_backButton.CheckForCollision(pygame.mouse.get\_pos()):

            self.\_\_backButton.SetActive()

        else:

            self.\_\_backButton.SetInactive()

        self.\_Render()

    def \_Render(self):

*#Override to prevent background fill*

        self.\_\_backButton.Render(self.\_window)

    def \_HandleInputs(self):

        super().\_HandleInputs()

        i = 0

        while i < len(self.\_inputHandler.inputsList):

            if self.\_inputHandler.inputsList[i][:6] == "CLICK:":

                clickLocation = self.\_inputHandler.inputsList[i][6:].split(",")

                clickLocation = (int(clickLocation[0]), int(clickLocation[1]))

                if self.\_\_backButton.CheckForCollision(clickLocation):

                    self.backButtonPressed = True

                    self.\_\_backButton.clicked = True

                self.\_inputHandler.inputsList.pop(i)

            else:

                i += 1

## Game

The game object needs to recognise the user has chosen to go to the statistics screen and call the main method of the statistics screen scene object.

### Pseudocode

ELSE IF userChoice = "Statistics" THEN

self.\_\_WaitForData()

self.\_\_ShowStatistics()

### Python code

elif userChoice == "Statistics":

    self.\_\_WaitForData()

    self.\_\_ShowStatistics()

WaitForData()

def \_\_WaitForData(self):

    self.socket.msgsToSend.append("!STATISTICS")

    while self.\_\_stats is None:

        self.\_\_CheckMessages()

ShowStatistics()

def \_\_ShowStatistics(self):

    self.\_\_statisticsScreen = StatisticsScreen(self.\_\_window, self.\_\_resolution, self.\_\_stats, self.socket)

    while not self.\_\_statisticsScreen.backButtonPressed:

        self.\_\_statisticsScreen.main()

        if self.\_\_statisticsScreen.userQuit:

            self.\_\_userQuit = True

            return 0

        pygame.display.update()

CheckMessages() was covered in the game section

# Matchmaking screen

The matchmaking screen is a simple screen the user sees while they are looking for a game. It comes after the main menu once the user presses the play button.

The matchmaking screen needs to show that the player is in queue and have a button so they can leave the queue.

I didn’t make a design for this one as it is just a text in the middle of the screen and a box slightly below it.

## Pseudocode

CLASS MatchmakingScreen(Scene)

SUBROUTINE init(self, window, resolution, socket)

self.gameFound <-- False

self.userClickedBackButton <-- False

self.\_\_timeSinceLastMessageUpdate <-- 0

self.\_\_numberOfDots <-- 0

self.\_\_font <-- pygame Font object

textSize <-- size for text "Looking for game..."

textLocation <-- centred x and y

self.\_\_textToRender <-- "Looking for game"

self.\_\_textObject <-- Text object

self.\_listOfTextObjects <-- [self.\_\_textObject]

backButtonSize <-- (400,60)

backButtonLocation <-- centred x and y = 680

backButtonRect <-- pygame Rect object

self.backButton <-- Button object

self.\_listOfButtonObjects <-- [self.backButton]

ENDSUBROUTINE

SUBROUTINE main(self)

Inherit from Scene

self.\_\_TextAnimation()

self.\_\_HandleMessages()

ENDSUBROUTINE

SUBROUTINE \_\_HandleMessages(self)

i <-- 0

WHILE i < LEN(self.socket.receivedMsgs)

IF self.socket.receivedMsgs[i] = "!GAMEFOUND" THEN

self.gameFound <-- True

self.socket.receivedMsgs.POP(i)

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

Inherit from Scene

i <-- 0

WHILE i < LEN(self.\_inputHandler.inputsList)

IF self.\_inputHandler.inputsList[i][:6] = "CLICK:" THEN

clickLocation <-- mouse location

IF self.backButton.CheckForCollision(clickLocation) THEN

self.backbuttonPressed <-- True

self.backButton.clicked <-- True

ENDIF

self.\_inputHandler.inputsList.POP(i)

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

SUBROUTINE \_\_TextAnimation(self)

add time since last frame to self.\_\_timeSinceLastMessageUpdate

IF self.\_\_timeSinceLastMessageUpdate >= 700 THEN

self.\_\_textToRender <-- "Looking for game" + "." \* self.\_\_numberOfDots

self.textObject.SetText(self.\_\_textToRender)

self.\_\_numberOfDots <-- self.\_\_numberOfDots + 1

IF self.\_\_numberOfDots = 4 THEN

self.\_\_numberOfDots <-- 0

ENDIF

self.\_\_timeSinceLastMessageUpdate <-- 0

ENDIF

ENDSUBROUTINE

ENDCLASS

## Python code

*#Displays message that they are in queue*

class MatchmakingScreen(Scene):

    def \_\_init\_\_(self, window, resolution, socket=None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        self.gameFound = False

        self.userClickedBackButton = False

*#Used for changing the number of dots in the text*

        self.\_\_timeSinceLastMessageUpdate = 0

        self.\_\_numberOfDots = 0

        self.\_\_font = pygame.font.SysFont("Calibri", int(72 \* self.\_resolution[1]))

*#Text for the screen*

        textSize = self.\_\_font.size("Looking for game...")

        textLocation = (int((self.\_resolution[0] \* 1920 - textSize[0]) / 2), int((self.\_resolution[1] \* 1080 - textSize[1]) / 2))

        self.\_\_textToRender = "Looking for game"

        self.\_\_textObject = Text(self.\_\_font, text=self.\_\_textToRender, location=textLocation)

        self.\_listOfTextObjects = [self.\_\_textObject]

*#Button to dequeue*

        backButtonSize = (400 \* self.\_resolution[0], 60 \* self.\_resolution[1])

*#Button needs to be centred and 680 pixels down*

        backButtonLocation = ((self.\_resolution[0] \* 1920 - backButtonSize[0]) / 2, 680 \* self.\_resolution[1])

        backButtonRect = pygame.Rect(backButtonLocation[0], backButtonLocation[1], backButtonSize[0], backButtonSize[1])

        self.backButton = Button(backButtonRect, (40,40,40), (25,25,25), (255,255,255), text="Back")

        self.\_listOfButtonObjects = [self.backButton]

    def main(self):

        super().main()

        self.\_\_TextAnimation()

        self.\_\_HandleMessages()

    def \_\_HandleMessages(self):

        i = 0

        while i < len(self.socket.receivedMsgs):

            if self.socket.receivedMsgs[i] == "!GAMEFOUND":

                self.gameFound = True

                self.socket.receivedMsgs.pop(i)

            else:

                i += 1

    def \_HandleInputs(self):

        super().\_HandleInputs()

        i = 0

        while i < len(self.\_inputHandler.inputsList):

            if self.\_inputHandler.inputsList[i][:6] == "CLICK:":

                clickLocation = self.\_inputHandler.inputsList[i][6:].split(",")

                clickLocation = (int(clickLocation[0]), int(clickLocation[1]))

                for button in self.\_listOfButtonObjects:

                    if button.CheckForCollision(clickLocation):

                        button.clicked = True

                        if button.text == "Back":

                            self.userClickedBackButton = True

                self.\_inputHandler.inputsList.pop(i)

            else:

                i += 1

    def \_\_TextAnimation(self):

*#Used for animation of looking for game text*

        self.\_\_timeSinceLastMessageUpdate += self.\_clock.get\_time()

        if self.\_\_timeSinceLastMessageUpdate >= 700:

            self.\_\_textToRender = "Looking for game" + "." \* self.\_\_numberOfDots

            self.\_\_textObject.SetText(self.\_\_textToRender)

            self.\_\_numberOfDots += 1

            if self.\_\_numberOfDots == 4:

                self.\_\_numberOfDots = 0

            self.\_\_timeSinceLastMessageUpdate = 0

The back button being pressed triggers the following in the Game object:

if userChoice == "Play":

*#Goes to matchmaking screen*

    self.\_\_WaitForGame()

*#If user quit during matchmaking*

*#Disconnecting from server is done in Play.py*

*#If the user disconnects from server while in queue, server will automatically dequeue them*

    if self.\_\_userQuit:

        return 0

*#If user left queue*

    elif self.\_\_matchmakingScreen.userClickedBackButton:

        self.socket.msgsToSend.append("!DEQUEUE")

*#Loop starts again and user goes to main menu*

## Server

The server needs to be able to dequeue a player from matchmaking. This just entails moving the player from one queue to the other.

### Pseudocode

SUBROUTINE HandleMessagesForPlayersInQueue(self)

playersQuit <-- []

FOR player IN self.playersInMatchmaking

WHILE player.msgsReceived.GetLength() <> 0

message <-- player.msgsReceived.Dequeue()

OUTPUT message

IF message = "!DISCONNECT" THEN

playersQuit.APPEND(player)

ELSE IF message = "!DEQUEUE" THEN

playersQuit.APPEND(player)

self.players.APPEND(player)

ENDIF

ENDWHILE

ENDFOR

The code above checks for the “!DEQUEUE” message and moves the player to the players queue and removes them from the playersInMatchmaking queue.

### Python code

def HandleMessagesForPlayersInQueue(self):

    playersQuit = []

    for player in self.playersInMatchmaking:

        while player.msgsReceived.GetLength() != 0:

            message = player.msgsReceived.Dequeue()

            print(message)

            if message == "!DISCONNECT":

                playersQuit.append(player)

            elif message == "!DEQUEUE":

                playersQuit.append(player)

                self.players.append(player)

## Game

The game object needs to check if the player has queued for a game and a game has not been found. It also needs to make sure the player hasn’t pressed the back button.

If the player has found a game it will send the player to another scene but when the player presses the back button it sends them to the main menu.

### Pseudocode

IF userChoice = "Play" THEN

self.\_\_WaitForGame()

IF self.\_\_userQuit THEN RETURN 0

ELSE IF self.\_\_matchmakingScreen.userClickedBackButton THEN

self.socket.msgsToSend.APPEND("!DEQUEUE")

### Python code

if userChoice == "Play":

*#Goes to matchmaking screen*

    self.\_\_WaitForGame()

*#If user quit during matchmaking*

*#Disconnecting from server is done in Play.py*

*#If the user disconnects from server while in queue, server will automatically dequeue them*

    if self.\_\_userQuit:

        return 0

*#If user left queue*

    elif self.\_\_matchmakingScreen.userClickedBackButton:

        self.socket.msgsToSend.append("!DEQUEUE")

WaitForGame()

def \_\_WaitForGame(self):

    self.socket.msgsToSend.append("!QUEUE")

*#Wait until game is found or user cancels game being found*

*#While no game is found and user hasn't quit the queue*

    while not self.\_\_matchmakingScreen.gameFound and not self.\_\_matchmakingScreen.userClickedBackButton:

        self.\_\_matchmakingScreen.main()

        if self.\_\_matchmakingScreen.userQuit:

            self.\_\_userQuit = True

            return 0

        pygame.display.update()

# Timer screen

The timer screen simply needs to show a timer and this timer is done server side.

## Pseudocode

CLASS TimerScene(Scene)

SUBROUTINE init(self, windowm, resolution, socket)

Inherit from Scene

self.timerFinished <-- False

self.\_\_font <-- pygame Font object

textSize <-- size of "Waiting for timer"

textLocation <-- centre of screen

self.\_\_textObject <-- Text object

self.\_listOfTextObjects <-- [self.\_\_textObject]

ENDSUBROUTINE

SUBROUTINE main(self)

Inherit from Scene

self.\_\_HandleMessages()

ENDSUBROUTINE

SUBROUTINE \_\_HandleMessages(self)

i <-- 0

WHILE i < LEN(self.socket.receivedMsgs)

IF self.socket.receivedMsgs[i][:10] = "!TIMELEFT:" THEN

timeLeft <-- self.socket.receivedMsgs[i][10:]

self.\_\_UpdateTextObject(timeLeft)

IF STRING\_TO\_INT(timeLeft) = 0 THEN

self.timerFinished <-- True

ENDIF

self.socket.receivedMsgs.POP(i)

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDSUBROUTINE

SUBROUTINE \_\_UpdateTextObject(self, newText)

textSize <-- size of newText

textLocation <-- centre of screen

self.\_\_textObject.SetText(newText)

self.\_\_textObject.location <-- textLocation

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

Inherit from Scene

self.\_inputHandler.inputsList <-- []

ENDSUBROUTINE

ENDCLASS

## Python code

*#Scene for timer*

class TimerScene(Scene):

    def \_\_init\_\_(self, window, resolution, socket=None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        self.timerFinished = False

        self.\_\_font = pygame.font.SysFont("Calibri", int(72 \* self.\_resolution[1]))

*#Text for the screen*

        textSize = self.\_\_font.size("Waiting for timer")

        textLocation = (int((self.\_resolution[0] \* 1920 - textSize[0]) / 2), int((self.\_resolution[1] \* 1080 - textSize[1]) / 2))

        self.\_\_textObject = Text(self.\_\_font, text="Waiting for timer", location=textLocation)

        self.\_listOfTextObjects = [self.\_\_textObject]

    def main(self):

        super().main()

        self.\_\_HandleMessages()

    def \_\_HandleMessages(self):

        i = 0

        while i < len(self.socket.receivedMsgs):

            if self.socket.receivedMsgs[i][:10] == "!TIMELEFT:":

                timeLeft = self.socket.receivedMsgs[i][10:]

*#Updates timer on screen*

                self.\_\_UpdateTextObject(timeLeft)

                if int(timeLeft) == 0:

                    self.timerFinished = True

*#Removes message from list*

                self.socket.receivedMsgs.pop(i)

            else:

                i += 1

*#Updates text and location to be centred*

    def \_\_UpdateTextObject(self, newText):

        textSize = self.\_\_font.size(newText)

        textLocation = (int((self.\_resolution[0] \* 1920 - textSize[0]) / 2), int((self.\_resolution[1] \* 1080 - textSize[1]) / 2))

        self.\_\_textObject.SetText(newText)

        self.\_\_textObject.location = textLocation

    def \_HandleInputs(self):

        super().\_HandleInputs()

*#Empties inputs list so they dont carry over to the next scene*

        self.\_inputHandler.inputsList = []

## Server

The server needs to have the timer and send messages to the client for how long is left. While it does this it needs to generate the words needed for the race.

### Pseudocode

CLASS TimerStage(Stage)

SUBROUTINE init(self, player1, player2)

Inherit from Stage

self.timerFinished <-- False

#Attribute for timer in seconds

self.timeUntilStart <-- 5

self.\_\_timeSinceLastMessage <-- 0

self.\_SendMessageToBothPlayers("!STARTTIMER")

self.\_SendMessageToBothPlayers("!TIMELEFT:" + INT\_TO\_STRING(self.timeUntilStart))

#Word generation

self.\_\_wordGenerator <-- WordGenerator object

self.textForPlayersToType <-- self.\_\_wordGenerator.GetWordsForProgram(500)

ENDSUBROUTINE

SUBROUTINE main(self)

Inherit from Stage

self.\_\_timeSinceLastMessage <-- self.\_\_timeSinceLastMessage + time since last frame

IF self.timeUntilStart = 0 THEN

self.timerFinished <-- True

self.\_SendMessageToBothPlayers("!TEXTTOWRITE:" + self.textForPlayersToType)

ELSE IF self.\_\_timeSinceLastMessage >= 1000 THEN

self.\_\_timeSinceLastMessage <-- self.\_\_timeSinceLastMessage - 1000

self.timeUntilStart <-- self.timeUntilStart - 1

self.\_SendMessageToBothPlayers("!TIMELEFT:" + INT\_TO\_STRING(self.timeUntilStart))

ENDIF

ENDSUBROUTINE

ENDCLASS

### Python code

*#Stage for when players are waiting for game to start*

class TimerStage(Stage):

    def \_\_init\_\_(self, player1 : Player, player2 : Player) -> None:

        super().\_\_init\_\_(player1, player2)

        self.timerFinished = False

*#Attribute for timer in seconds*

        self.timeUntilStart = 5

        self.\_\_timeSinceLastMessage = 0

        self.\_SendMessageToBothPlayers("!STARTTIMER")

        self.\_SendMessageToBothPlayers(f"!TIMELEFT:{self.timeUntilStart}")

*#Word generation*

        self.\_\_wordGenerator = WordGenerator()

        self.textForPlayersToType = self.\_\_wordGenerator.GetWordsForProgram(500)

        self.\_clock.tick()

*#Runs every frame in Game.main() while timerfinished is false*

    def main(self):

        super().main()

        self.\_\_timeSinceLastMessage += self.\_clock.get\_time()

        if self.timeUntilStart == 0:

            self.timerFinished = True

#This is meant to be indented, formatting messed up when pasting in word        self.\_SendMessageToBothPlayers(f"!TEXTTOWRITE:{self.textForPlayersToType}")

        elif self.\_\_timeSinceLastMessage >= 1000:

            self.\_\_timeSinceLastMessage -= 1000

            self.timeUntilStart -= 1

            self.\_SendMessageToBothPlayers(f"!TIMELEFT:{self.timeUntilStart}")

## Game

The game object needs to check if a game has been found and then if the timer has finished and send the player to the race screen when it is finished.

### Pseudocode

ELSE IF self.\_\_matchmakingScreen.gameFound THEN

self.\_\_PreGameTimer()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

### Python code

elif self.\_\_matchmakingScreen.gameFound:

    self.\_\_PreGameTimer()

    if self.\_\_userQuit:

        return 0

PreGameTimer()

def \_\_PreGameTimer(self):

*#Waits for server to start the timer*

    while not self.\_\_timerStarted:

        self.\_\_CheckMessages()

    while not self.\_\_timerScene.timerFinished:

        self.\_\_timerScene.main()

        if self.\_\_timerScene.userQuit:

            self.\_\_userQuit = True

            return 0

        pygame.display.update()

# Race screen

The race screen makes use of the RaceScene scene object. This object needs to draw 2 textboxes which the player can type into 1 of. It also needs to draw a timer on the screen.

## Pseudocode

CLASS RaceScene(Scene)

SUBROUTINE init(self, window, resolution, previewText, socket=None)

Inherit from Scene

self.playerFinished <-- False

self.gameOver <-- False

#Colours for textbox

colourActive <-- (40,40,40)

colourInactive <-- (25,25,25)

previewTextColour <-- (160,160,160)

incorrectTextColour <-- (255,0,0)

#Main textbox

textBoxFont <-- pygame Font object

textBoxSize <-- 2/5th the width of the screen and 50 pixels tall

textBoxLocation <-- centred on x and 250 pixels down

textBoxRect <-- pygame Rect object

self.\_\_textBox <-- TextBox object

#Other player textbox

textBoxLocation <-- centred on x and 250 pixels up

textBoxRect <-- pygame Rect object

self.\_\_opponentTextBox <-- TextBox object

self.\_listOfBoxObjects <-- [self.\_\_textBox, self.\_\_opponentTextBox]

self.timerFinished <-- False

self.\_\_font <-- pygame Font object

#Text for timer

textSize <-- size of "30" when rendered

textLocation <-- centre of screen

self.\_\_textObject <-- Text object

self.\_listOfTextObjects <-- [self.\_\_textObject]

ENDSUBROUTINE

#Override for performance improvements

SUBROUTINE main(self)

self.\_timeSinceLastBackspace <-- self.\_timeSinceLastBackspace + time since last frame

self.\_HandleInputs()

self.\_Render()

self.\_\_HandleMessages()

ENDSUBROUTINE

SUBROUTINE \_Render(self)

draw self.\_backgroundSurface on self.\_window

draw self.\_\_textBox

draw self.\_\_opponentTextBox

draw self.\_\_textObject

ENDSUBROUTINE

SUBROUTINE SetPreviewText(self, previewText)

self.\_\_textBox.previewText <-- previewText

ENDSUBROUTINE

SUBROUTINE \_\_HandleMessages(self)

unusedMessages <-- []

#Code for receiving messages from server

WHILE self.socket.receivedMsgs <> []

message <-- self.socket.receivedMsgs.POP(0)

IF message[:10] = "!TIMELEFT:" THEN

timeLeft <-- message[10:]

#Updates timer on screen

self.\_\_UpdateTextObject(timeLeft)

IF timeLeft = 0 THEN

self.timerFinished <-- True

ENDIF

ELSE IF message[:17] = "!OTHERPLAYERTEXT:" THEN

text <-- message[17:]

self.\_\_opponentTextBox.SetText(text)

ELSE IF message = "!GAMECOMPLETE" THEN

self.socket.msgsToSend.APPEND(f"!FINALTEXT:{self.\_\_textBox.text}")

self.timerFinished <-- True

ELSE IF message = "!GAMECOMPLETED" THEN

self.gameOver <-- True

ELSE

unusedMessages.APPEND(message)

ENDIF

FOR message IN unusedMessages

self.socket.receivedMsgs.APPEND(message)

ENDFOR

ENDSUBROUTINE

SUBROUTINE \_\_UpdateTextObject(self, newText)

textSize <-- size of newText when rendered

textLocation <-- centre of screen

self.\_\_textObject.location <-- textLocation

self.\_\_textObject.SetText(newText)

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

FOR event IN events that have happened

IF player has quit THEN

self.userQuit <-- True

ELSE IF player has clicked somewhere THEN

mousePos <-- position of mouse

IF self.\_\_textBox.CheckForCollisionWithMouse(mousePos) THEN

self.\_\_textBox.SetActive()

ELSE

self.\_\_textBox.SetInactive()

ENDIF

ELSE IF player pressed a key THEN

IF player pressed backspace THEN

self.\_backspace <-- True

self.\_timeSinceLastBackspace <-- -200

IF self.\_\_textBox.isActive THEN

self.\_\_textBox.RemoveLetter(self.\_ctrl)

ENDIF

ELSE IF player pressed control THEN

self.\_ctrl <-- True

ELSE

IF self.\_\_textBox.isActive AND input was a letter THEN

self.\_\_textBox.AddLetter(letter player typed)

IF self.\_\_textBox.CheckIfFinished() THEN

self.socket.msgsToSend.APPEND("!FINALTEXT:" + self.\_\_textBox.text)

ELSE

self.socket.msgsToSend.APPEND("!TEXT:" + self.\_\_textBox.text)

ENDIF

ENDIF

ENDIF

ELSE IF player let go of a key THEN

IF key was backspace THEN

self.\_backspace <-- False

ELSE IF key was control THEN

self.\_ctrl <-- False

ENDIF

ENDIF

ENDFOR

ENDSUBROUTINE

ENDCLASS

## Python code

class RaceScene(Scene):

    def \_\_init\_\_(self, window, resolution, previewText, socket=None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        self.playerFinished = False

        self.gameOver = False

*#Colours for textbox*

        colourActive = (40,40,40)

        colourInactive = (25,25,25)

        previewTextColour = (160,160,160)

        incorrectTextColour = (255,0,0)

*#Main textbox*

        textBoxFont = pygame.font.SysFont("Courier New", int(42 \* self.\_resolution[1]))

        textBoxSize = (self.\_resolution[0] \* 1920 \* 2/5, 50 \* self.\_resolution[1])

        textBoxLocation = ((self.\_resolution[0] \* 1920 - textBoxSize[0]) / 2, 250 \* self.\_resolution[1])

        textBoxRect = pygame.Rect(textBoxLocation[0], textBoxLocation[1], textBoxSize[0], textBoxSize[1])

        self.\_\_textBox = TextBox(textBoxRect, textBoxFont, self.\_resolution, colourActive, colourInactive, (38, 191, 79), previewText, previewTextColour, incorrectTextColour)

*#Other player textbox*

*#Box is 250 pixels above the bottom of the screen*

        textBoxLocation = ((self.\_resolution[0] \* 1920 - textBoxSize[0]) / 2, self.\_resolution[1] \* 1080 - (250 + textBoxSize[1]) \* self.\_resolution[1])

        textBoxRect = pygame.Rect(textBoxLocation[0], textBoxLocation[1], textBoxSize[0], textBoxSize[1])

        self.\_\_opponentTextBox = TextBox(textBoxRect, textBoxFont, self.\_resolution, colourActive, colourInactive, (38, 191, 79), previewText, previewTextColour, incorrectTextColour)

        self.\_listOfBoxObjects = [self.\_\_textBox, self.\_\_opponentTextBox]

*#Needs to be attribute as it will be used to change text later on*

        self.timerFinished = False

        self.\_\_font = pygame.font.SysFont("Calibri", int(72 \* self.\_resolution[1]))

*#Text for timer*

        textSize = self.\_\_font.size("30")

        textLocation = (int((self.\_resolution[0] \* 1920 - textSize[0]) / 2), int((self.\_resolution[1] \* 1080 - textSize[1]) / 2))

        self.\_\_textObject = Text(self.\_\_font, text="30", location=textLocation)

        self.\_listOfTextObjects = [self.\_\_textObject]

*#Override for performance improvements*

    def main(self):

        self.\_clock.tick()

        self.\_timeSinceLastBackspace += self.\_clock.get\_time()

        self.\_HandleInputs()

*#Automatic removal of text every 50 milliseconds*

        if self.\_backspace and self.\_timeSinceLastBackspace >= 50 and self.\_\_textBox.isActive:

            self.\_\_textBox.RemoveLetter(self.\_ctrl)

            self.\_timeSinceLastBackspace = 0

        self.\_Render()

        self.\_\_HandleMessages()

    def \_Render(self):

        self.\_window.blit(self.\_backgroundSurface, (0,0))

        self.\_\_textBox.Render(self.\_window)

        self.\_\_opponentTextBox.Render(self.\_window)

        self.\_\_textObject.Render(self.\_window)

    def SetPreviewText(self, previewText):

        self.\_\_textBox.previewText = previewText

    def \_\_HandleMessages(self):

        unusedMessages = []

*#Code for receiving messages from server*

        while self.socket.receivedMsgs != []:

            message = self.socket.receivedMsgs.pop(0)

            if message[:10] == "!TIMELEFT:":

                timeLeft = message[10:]

*#Updates timer on screen*

                self.\_\_UpdateTextObject(timeLeft)

                if timeLeft == 0:

                    self.timerFinished = True

*#Other player's text*

            elif message[:17] == "!OTHERPLAYERTEXT:":

                text = message[17:]

                self.\_\_opponentTextBox.SetText(text)

*#Time has run out, client has to send text*

            elif message == "!GAMECOMPLETE":

                self.socket.msgsToSend.append(f"!FINALTEXT:{self.\_\_textBox.text}")

                self.timerFinished = True

*#Server received both player's final text and game has finished*

*#Emphasis on the D in completed, not the same as complete as it is for server asking for player's final text*

            elif message == "!GAMECOMPLETED":

                self.gameOver = True

            else:

                unusedMessages.append(message)

        for message in unusedMessages:

            self.socket.receivedMsgs.append(message)

*#Updates text and location to be centred*

    def \_\_UpdateTextObject(self, newText):

        textSize = self.\_\_font.size(newText)

        textLocation = (int((self.\_resolution[0] \* 1920 - textSize[0]) / 2), int((self.\_resolution[1] \* 1080 - textSize[1]) / 2))

        self.\_\_textObject.location = textLocation

        self.\_\_textObject.SetText(newText)

    def \_HandleInputs(self):

*#Inputs can be handled directly in this, as there is no need to keep certain inputs to deal with in child classes*

        for event in pygame.event.get():

*#If the player quit*

            if event.type == pygame.QUIT:

                self.userQuit = True

*#If player clicks*

            elif event.type == pygame.MOUSEBUTTONDOWN:

                mousePos = pygame.mouse.get\_pos()

                if self.\_\_textBox.CheckForCollisionWithMouse(mousePos):

                    self.\_\_textBox.SetActive()

                else:

                    self.\_\_textBox.SetInactive()

            elif event.type == pygame.KEYDOWN:

                if event.key == pygame.K\_BACKSPACE:

                    self.\_backspace = True

                    self.\_timeSinceLastBackspace = -200

                    if self.\_\_textBox.isActive:

                        self.\_\_textBox.RemoveLetter(self.\_ctrl)

                elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:

                    self.\_ctrl = True

                else:

                    if self.\_\_textBox.isActive and (event.unicode.isalpha() or event.unicode == " " or event.unicode == "-"):

                        self.\_\_textBox.AddLetter(event.unicode)

                        if self.\_\_textBox.CheckIfFinished():

                            self.socket.msgsToSend.append(f"!FINALTEXT:{self.\_\_textBox.text}")

                        else:

                            self.socket.msgsToSend.append(f"!TEXT:{self.\_\_textBox.text}")

            elif event.type == pygame.KEYUP:

                if event.key == pygame.K\_BACKSPACE:

                    self.\_backspace = False

                elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:

                    self.\_ctrl = False

## Server

The server needs to send the player’s text to the other player, handle the timer and to update the player’s stats at the end of the game. This is done using another Stage object which is used in the Game object (server).

### Pseudocode

CLASS Race(Stage)

SUBROUTINE (self, player1, player2, textPlayersHaveToType, databaseHandler)

Inherit from Stage

self.raceFinished <-- False

self.\_\_waitingForPlayersText <-- False

self.timeUntilEnd <-- 30

self.\_\_timeSinceLastTimerUpdate <-- 0

self.\_\_textPlayersHaveToType <-- textPlayersHaveToType

self.\_\_player1FinalText <-- None

self.\_\_player2FinalText <-- None

self.\_\_player1TimeFinished <-- None

self.\_\_player2TimeFinished <-- None

self.\_\_dbHandler <-- databaseHandler

ENDSUBROUTINE

SUBROUTINE main(self)

Inherit from Stage

self.\_\_timeSinceLastTimerUpdate <-- self.\_\_timeSinceLastTimerUpdate + time since last frame

IF self.\_\_timeSinceLastTimerUpdate >= 1000 AND NOT self.\_\_waitingForPlayersText THEN

self.timeUntilEnd <-- self.timeUntilEnd - 1

self.\_\_timeSinceLastTimerUpdate <-- self.\_\_timeSinceLastTimerUpdate - 1000

self.\_SendMessageToBothPlayers("!TIMELEFT:" + STRING\_TO\_INT(self.timeUntilEnd))

IF self.timeUntilEnd = 0 THEN

self.\_SendMessageToBothPlayers("!GAMECOMPLETE")

self.\_\_waitingForPlayersText = True

ENDIF

ENDIF

FOR player IN self.\_players

IF LEN(player.textWritten) = LEN(self.\_\_textPlayersHaveToType) THEN

OUTPUT "A player has won"

IF self.\_\_player1FinalText IS NOT None AND self.\_\_player2FinalText IS NOT None THEN

self.\_SendMessageToBothPlayers("!GAMECOMPLETED")

self.raceFinished <-- True

self.\_\_UpdatePlayers()

ENDIF

ENDSUBROUTINE

#This method calculates the players' new stats

SUBROUTINE \_\_UpdatePlayers(self)

self.\_player1.wordsTyped <-- self.\_player1.wordsTyped + LEN(self.\_\_player1FinalText.split(" "))

self.\_player2.wordsTyped <-- self.\_player2.wordsTyped + LEN(self.\_\_player2FinalText.split(" "))

self.\_player1.timePlayed <-- self.\_player1.timePlayed + self.\_\_player1TimeFinished

self.\_player2.timePlayed <-- self.\_player2.timePlayed + self.\_\_player2TimeFinished

#Find number of letters each player got correct

player1LettersCorrect <-- 0

FOR i IN RANGE(LEN(self.\_\_player1FinalText))

self.\_player1.lettersTyped <-- self.\_player1.lettersTyped + 1

IF self.\_\_player1FinalText[i] = self.\_\_textPlayersHaveToType[i] THEN

player1LettersCorrect <-- player1LettersCorrect + 1

self.\_player1.lettersTypedCorrectly <-- self.\_player1.lettersTypedCorrectly + 1

ENDIF

ENDFOR

player2LettersCorrect <-- 0

FOR i IN RANGE(LEN(self.\_\_player2FinalText))

self.\_player2.lettersTyped <-- self.\_player2.lettersTyped + 1

IF self.\_\_player2FinalText[i] = self.\_\_textPlayersHaveToType[i] THEN

player2LettersCorrect <-- player2LettersCorrect + 1

self.\_player2.lettersTypedCorrectly <-- self.\_player2.lettersTypedCorrectly + 1

ENDIF

ENDFOR

#Determine winner

IF player1LettersCorrect = player2LettersCorrect THEN

IF self.\_\_player1TimeFinished = self.\_\_player2TimeFinished THEN

#Game is a draw

winner <-- None

loser <-- None

ELSE IF self.\_\_player1TimeFinished < self.\_\_player2TimeFinished THEN

winner <-- self.\_player1

loser <-- self.\_player2

ELSE

winner <-- self.\_player2

loser <-- self.\_player2

ELSE IF player1LettersCorrect > player2LettersCorrect THEN

winner <-- self.\_player1

winMargin <-- player1LettersCorrect - player2LettersCorrect

if winMargin > winner.largestWinMargin:

winner.largestWinMargin <-- winMargin

loser <-- self.\_player2

ELSE

winner <-- self.\_player2

winMargin <-- player2LettersCorrect - player1LettersCorrect

IF winMargin > winner.largestWinMargin THEN

winner.largestWinMargin <-- winMargin

ENDIF

loser <-- self.\_player1

ENDIF

IF winner IS NOT None THEN

winner.msgsToSend.Enqueue("!MATCHOUTCOME:WIN")

loser.msgsToSend.Enqueue("!MATCHOUTCOME:LOSS")

winner.msgsToSend.Enqueue("!MARGIN:" + winMargin)

loser.msgsToSend.Enqueue("!MARGIN:" + winMargin)

winner.gamesWon <-- winner.gamesWon + 1

winner.gamesPlayed <-- winner.gamesPlayed + 1

winner.currentWinstreak <-- winner.currentWinstreak + 1

IF winner.currentWinstreak > winner.longestStreak THEN

winner.longestStreak <-- winner.currentWinstreak

loser.gamesPlayed <-- loser.gamesPlayed + 1

loser.currentWinstreak <-- 0

winner.sumOfOpponentsElo <-- winner.sumOfOpponentsElo + loser.Elo

loser.sumOfOpponentsElo <-- loser.sumOfOpponentsElo + winner.Elo

winnerGamesLost <-- winner.gamesPlayed - winner.gamesWon

#This is the formula for calculating PR (Performance Rating) increase

EloDiff <-- (winner.sumOfOpponentsElo + 400 \* (winner.gamesWon - winnerGamesLost)) / winner.gamesPlayed - winner.Elo

winner.msgsToSend.Enqueue("!ELO:" + EloDiff)

winner.Elo <-- winner.Elo EloDiff

IF winner.Elo > winner.highestElo THEN

winner.highestElo <-- winner.Elo

ENDIF

loserGamesLost <-- loser.gamesPlayed - loser.gamesWon

EloDiff <-- loser.Elo - (loser.sumOfOpponentsElo + 400 \* (loser.gamesWon - loserGamesLost)) / loser.gamesPlayed

loser.msgsToSend.Enqueue("!ELO:" + EloDiff)

loser.Elo <-- loser.Elo - EloDiff

ELSE

self.\_SendMessageToBothPlayers("!MATCHOUTCOME:DRAW")

ENDIF

#Updates database

self.\_\_dbHandler.SaveUser(self.\_player1)

self.\_\_dbHandler.SaveUser(self.\_player2)

self.\_player1.gameFinished <-- True

self.\_player2.gameFinished <-- True

ENDSUBROUTINE

SUBROUTINE \_HandleMessages(self)

Inherit from Stage

unusedMessages <-- []

WHILE self.\_player1.msgsReceived.GetLength() <> 0

message <-- self.\_player1.msgsReceived.Dequeue()

IF message[:6] = "!TEXT:" THEN

text <-- message[6:]

self.\_player1.textWritten <-- text

self.\_player2.msgsToSend.Enqueue("!OTHERPLAYERTEXT:" + self.\_player1.textWritten)

ELSE IF message[:11] = "!FINALTEXT:" THEN

self.\_\_player1FinalText <-- message[11:]

self.\_\_player1TimeFinished <-- 30 - self.timeUntilEnd

ELSE:

unusedMessages.APPEND(message)

ENDIF

ENDWHILE

FOR message IN unusedMessages

self.\_player1.msgsReceived.Enqueue(message)

ENDFOR

#For player2

unusedMessages <-- []

WHILE self.\_player2.msgsReceived.GetLength() <> 0

message = self.\_player2.msgsReceived.Dequeue()

IF message[:6] = "!TEXT:" THEN

text <-- message[6:]

self.\_player2.textWritten <-- text

self.\_player1.msgsToSend.Enqueue("!OTHERPLAYERTEXT:" + self.\_player2.textWritten)

ELSE IF message[:11] = "!FINALTEXT:" THEN

self.\_\_player2FinalText <-- message[11:]

self.\_\_player2TimeFinished <-- 30 - self.timeUntilEnd

ELSE:

unusedMessages.APPEND(message)

ENDIF

ENDWHILE

FOR message IN unusedMessages:

self.\_player2.msgsReceived.Enqueue(message)

ENDFOR

ENDSUBROUTINE

ENDCLASS

### Python code

class Race(Stage):

    def \_\_init\_\_(self, player1: Player, player2: Player, textPlayersHaveToType, databaseHandler : DatabaseHandler) -> None:

        super().\_\_init\_\_(player1, player2)

        self.raceFinished = False

        self.\_\_waitingForPlayersText = False

        self.timeUntilEnd = 30

        self.\_\_timeSinceLastTimerUpdate = 0

        self.\_\_textPlayersHaveToType = textPlayersHaveToType

        self.\_\_player1FinalText : str = None

        self.\_\_player2FinalText : str = None

        self.\_\_player1TimeFinished : int = None

        self.\_\_player2TimeFinished : int = None

        self.\_\_dbHandler = databaseHandler

    def main(self):

        super().main()

        self.\_\_timeSinceLastTimerUpdate += self.\_clock.get\_time()

        if self.\_\_timeSinceLastTimerUpdate >= 1000 and not self.\_\_waitingForPlayersText:

            self.timeUntilEnd -= 1

            self.\_\_timeSinceLastTimerUpdate -= 1000

            self.\_SendMessageToBothPlayers(f"!TIMELEFT:{self.timeUntilEnd}")

            if self.timeUntilEnd == 0:

                self.\_SendMessageToBothPlayers("!GAMECOMPLETE")

                self.\_\_waitingForPlayersText = True

*#Checking if player has finished*

        for player in self.\_players:

            if len(player.textWritten) == len(self.\_\_textPlayersHaveToType):

                print("A player has won")

        if self.\_\_player1FinalText is not None and self.\_\_player2FinalText is not None:

            self.\_SendMessageToBothPlayers("!GAMECOMPLETED")

            self.raceFinished = True

            self.\_\_UpdatePlayers()

    def \_\_UpdatePlayers(self):

*#For player1*

*#Calculate words typed*

        self.\_player1.wordsTyped += len(self.\_\_player1FinalText.split(" "))

        self.\_player2.wordsTyped += len(self.\_\_player2FinalText.split(" "))

        self.\_player1.timePlayed += self.\_\_player1TimeFinished

        self.\_player2.timePlayed += self.\_\_player2TimeFinished

*#Find number of letters each player got correct*

        player1LettersCorrect = 0

        for i in range(len(self.\_\_player1FinalText)):

            self.\_player1.lettersTyped += 1

            if self.\_\_player1FinalText[i] == self.\_\_textPlayersHaveToType[i]:

                player1LettersCorrect += 1

                self.\_player1.lettersTypedCorrectly += 1

        player2LettersCorrect = 0

        for i in range(len(self.\_\_player2FinalText)):

            self.\_player2.lettersTyped += 1

            if self.\_\_player2FinalText[i] == self.\_\_textPlayersHaveToType[i]:

                player2LettersCorrect += 1

                self.\_player2.lettersTypedCorrectly += 1

*#Determine winner*

*#If both players wrote same number of letters correctly*

        if player1LettersCorrect == player2LettersCorrect:

            if self.\_\_player1TimeFinished == self.\_\_player2TimeFinished:

*#Game is a draw*

                winner = None

                loser = None

            elif self.\_\_player1TimeFinished < self.\_\_player2TimeFinished:

                winner = self.\_player1

                loser = self.\_player2

            else:

                winner = self.\_player2

                loser = self.\_player2

        elif player1LettersCorrect > player2LettersCorrect:

            winner = self.\_player1

            winMargin = player1LettersCorrect - player2LettersCorrect

            if winMargin > winner.largestWinMargin:

                winner.largestWinMargin = winMargin

            loser = self.\_player2

        else:

            winner = self.\_player2

            winMargin = player2LettersCorrect - player1LettersCorrect

            if winMargin > winner.largestWinMargin:

                winner.largestWinMargin = winMargin

            loser = self.\_player1

        if winner is not None:

            winner.msgsToSend.Enqueue("!MATCHOUTCOME:WIN")

            loser.msgsToSend.Enqueue("!MATCHOUTCOME:LOSS")

            winner.msgsToSend.Enqueue(f"!MARGIN:{winMargin}")

            loser.msgsToSend.Enqueue(f"!MARGIN:{winMargin}")

            winner.gamesWon += 1

            winner.gamesPlayed += 1

            winner.currentWinstreak += 1

            if winner.currentWinstreak > winner.longestStreak:

                winner.longestStreak = winner.currentWinstreak

            loser.gamesPlayed += 1

            loser.currentWinstreak = 0

            winner.sumOfOpponentsElo += loser.Elo

            loser.sumOfOpponentsElo += winner.Elo

            winnerGamesLost = winner.gamesPlayed - winner.gamesWon

            EloDiff = (winner.sumOfOpponentsElo + 400 \* (winner.gamesWon - winnerGamesLost)) / winner.gamesPlayed - winner.Elo

            winner.msgsToSend.Enqueue(f"!ELO:{EloDiff}")

            winner.Elo += EloDiff

            if winner.Elo > winner.highestElo:

                winner.highestElo = winner.Elo

            loserGamesLost = loser.gamesPlayed - loser.gamesWon

            EloDiff = loser.Elo - (loser.sumOfOpponentsElo + 400 \* (loser.gamesWon - loserGamesLost)) / loser.gamesPlayed

            loser.msgsToSend.Enqueue(f"!ELO:{EloDiff}")

            loser.Elo -= EloDiff

        else:

            self.\_SendMessageToBothPlayers("!MATCHOUTCOME:DRAW")

*#Updates database*

        self.\_\_dbHandler.SaveUser(self.\_player1)

        self.\_\_dbHandler.SaveUser(self.\_player2)

        self.\_player1.gameFinished = True

        self.\_player2.gameFinished = True

    def \_HandleMessages(self):

        super().\_HandleMessages()

*#For player1*

        unusedMessages = []

        while self.\_player1.msgsReceived.GetLength() != 0:

            message = self.\_player1.msgsReceived.Dequeue()

            if message[:6] == "!TEXT:":

                text = message[6:]

                self.\_player1.textWritten = text

                self.\_player2.msgsToSend.Enqueue(f"!OTHERPLAYERTEXT:{self.\_player1.textWritten}")

            elif message[:11] == "!FINALTEXT:":

                self.\_\_player1FinalText = message[11:]

                self.\_\_player1TimeFinished = 30 - self.timeUntilEnd

            else:

                unusedMessages.append(message)

        for message in unusedMessages:

            self.\_player1.msgsReceived.Enqueue(message)

*#For player2*

        unusedMessages = []

        while self.\_player2.msgsReceived.GetLength() != 0:

            message = self.\_player2.msgsReceived.Dequeue()

            if message[:6] == "!TEXT:":

                text = message[6:]

                self.\_player2.textWritten = text

                self.\_player1.msgsToSend.Enqueue(f"!OTHERPLAYERTEXT:{self.\_player2.textWritten}")

            elif message[:11] == "!FINALTEXT:":

                self.\_\_player2FinalText = message[11:]

                self.\_\_player2TimeFinished = 30 - self.timeUntilEnd

            else:

                unusedMessages.append(message)

        for message in unusedMessages:

            self.\_player2.msgsReceived.Enqueue(message)

## Game

The game object needs to check if the previewtext has been received and then send the user to the race scene.

### Pseudocode

self.\_\_WaitForText()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

self.\_\_raceScene <-- RaceScene object

self.\_\_PlayGame()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

self.\_\_WaitForMatchResult()

### Python code

self.\_\_WaitForText()

if self.\_\_userQuit:

    return 0

*#Plays game while timer is more than 0*

self.\_\_raceScene = RaceScene(self.\_\_window, self.\_\_resolution, self.\_\_textToWrite, self.socket)

self.\_\_PlayGame()

if self.\_\_userQuit:

    return 0

WaitForText()

def \_\_WaitForText(self):

    while self.\_\_textToWrite is None:

        self.\_\_CheckMessages()

CheckMessages() was covered in the previous

PlayGame()

def \_\_PlayGame(self):

    while not self.\_\_raceScene.gameOver:

        self.\_\_raceScene.main()

        if self.\_\_raceScene.userQuit:

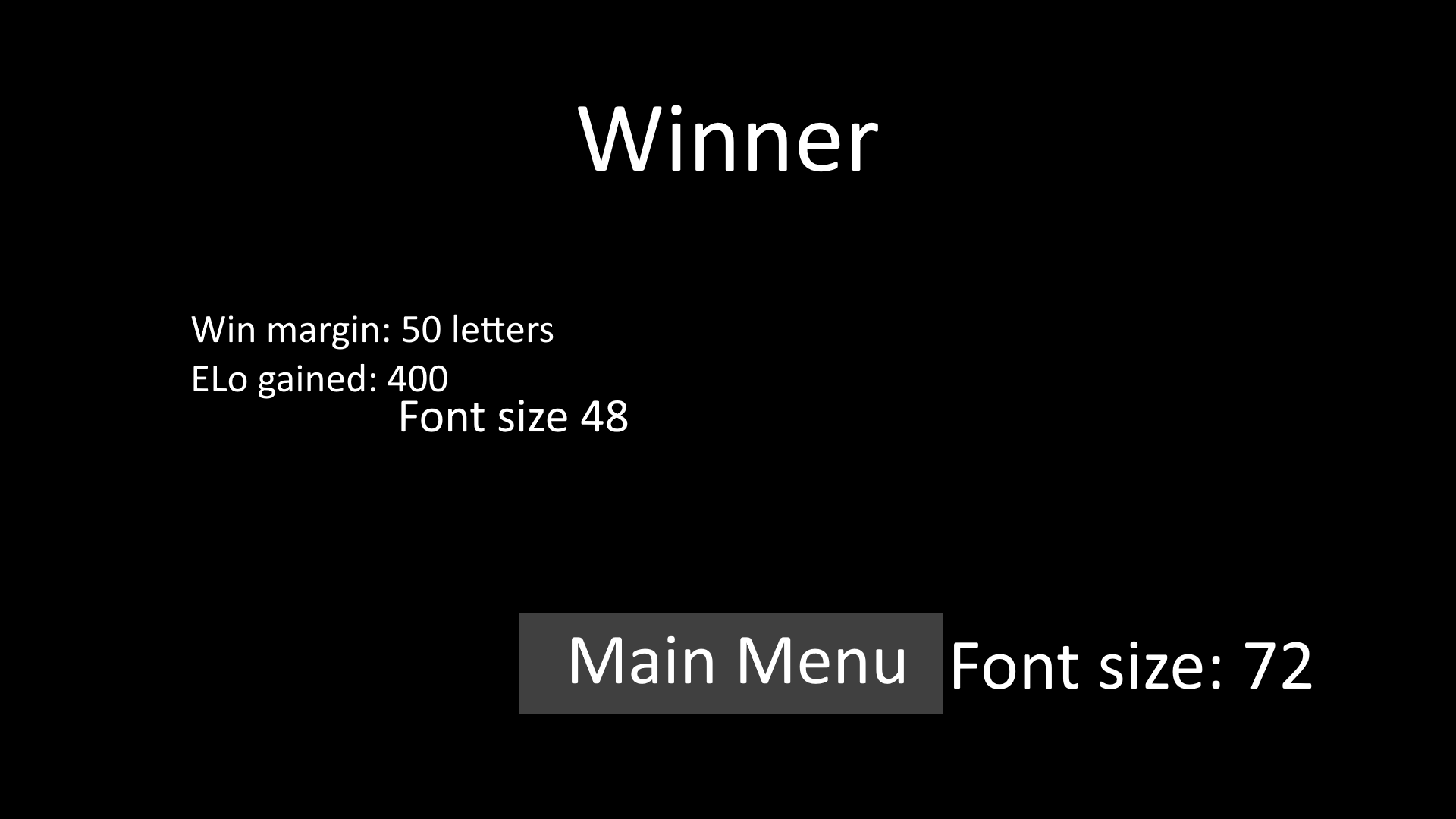
            self.\_\_userQuit = True

            return 0

        pygame.display.update()

# Post game screen

The post game screen needs to show the win margin, the elo gained/lost and whether the player won, lost or drew and a button to let the player go to the main menu.



The font sizes are not meant to be on there and were only there for me to know what size to program them in the program.

## Pseudocode

CLASS PostGame(Scene)

SUBROUTINE init(self, window, resolution, winloss, winmargin, ELodiff, socket)

Inherit from Scene

colourActive <-- (40,40,40)

colourInactive <-- (25,25,25)

#For button objects

self.menuButtonPressed <-- False

menuButtonSize <-- (560, 130)

menuButtonLocation <-- centred x and y = 800

menuButtonRect <-- pygame Rect object

self.\_\_menuButton <-- Button object

self.\_listOfButtonObjects <-- [self.\_\_menuButton]

#For text objects

self.\_\_font <-- pygame Font object

winlossTextSize <-- size of winloss when rendered

winlossTextLocation <-- centred x and y = 140

self.\_\_winlossText <-- Text object

self.\_\_infoFont <-- pygame Font object

IF winloss = "WIN" THEN

infoText1 <-- "Win margin: + winmargin + "letters"

infoText2 <-- "ELo gained:" + ELodiff

ELSE IF winloss <-- "LOSS" THEN

infoText1 <-- "Loss margin: {winmargin} letters"

infoText2 <-- "ELo lost: {ELodiff}"

else:

infoText1 <-- "Win margin:" + winmargin + "letters"

infoText2 <-- "ELo gained:" + ELodiff

infoText1Location <-- (250,400)

infoText2Location <-- (250, 480)

self.\_\_infoText1 <-- Text object

self.\_\_infoText2 <-- Text object

self.\_listOfTextObjects <-- [self.\_\_winlossText, self.\_\_infoText1, self.\_\_infoText2]

ENDSUBROUTINE

SUBROUTINE \_HandleInputs(self)

Inherit from Scene

i <-- 0

WHILE i < LEN(self.\_inputHandler.inputsList)

IF self.\_inputHandler.inputsList[i][:6] = "CLICK:" THEN

clickLocation <-- split with "," self.\_inputHandler.inputsList[i][6]

IF self.\_\_menuButton.CheckForCollision(clickLocation) THEN

self.menuButtonPressed <-- True

self.\_\_menuButton.clicked <-- True

self.\_inputHandler.inputsList.POP(i)

ELSE

i <-- i + 1

ENDIF

ENDWHILE

ENDCLASS

## Python code

class PostGame(Scene):

    def \_\_init\_\_(self, window, resolution, winloss, winmargin, ELodiff, socket: ClientSocket = None) -> None:

        super().\_\_init\_\_(window, resolution, socket)

        colourActive = (40,40,40)

        colourInactive = (25,25,25)

*#For button objects*

        self.menuButtonPressed = False

        menuButtonSize = (int(560 \* self.\_resolution[0]), int(130 \* self.\_resolution[1]))

        menuButtonLocation = (int((self.\_resolution[0] \* 1920 - menuButtonSize[0]) / 2), int(800 \* self.\_resolution[1]))

        menuButtonRect = pygame.Rect(menuButtonLocation[0], menuButtonLocation[1], menuButtonSize[0], menuButtonSize[1])

        self.\_\_menuButton = Button(menuButtonRect, colourActive, colourInactive, (255,255,255), text="Main Menu")

        self.\_listOfButtonObjects = [self.\_\_menuButton]

*#For text objects*

        self.\_\_font = pygame.font.SysFont("Calibri", int(72 \* self.\_resolution[1]))

        winlossTextSize = self.\_\_font.size(winloss)

        winlossTextLocation = (int((self.\_resolution[0] \* 1920 - winlossTextSize[0]) / 2), int(140 \* self.\_resolution[1]))

        self.\_\_winlossText = Text(self.\_\_font, text=winloss, location=winlossTextLocation)

        self.\_\_infoFont = pygame.font.SysFont("Calibri", int(48 \* self.\_resolution[1]))

        if winloss == "WIN":

            infoText1 = f"Win margin: {winmargin} letters"

            infoText2 = f"ELo gained: {ELodiff}"

        elif winloss == "LOSS":

            infoText1 = f"Loss margin: {winmargin} letters"

            infoText2 = f"ELo lost: {ELodiff}"

        else:

            infoText1 = f"Win margin: {winmargin} letters"

            infoText2 = f"ELo gained: {ELodiff}"

        infoText1Location = (int(250 \* self.\_resolution[0]), int(400 \* self.\_resolution[1]))

        infoText2Location = (int(250 \* self.\_resolution[0]), int(480 \* self.\_resolution[1]))

        self.\_\_infoText1 = Text(self.\_\_infoFont, text=infoText1, location=infoText1Location)

        self.\_\_infoText2 = Text(self.\_\_infoFont, text=infoText2, location=infoText2Location)

        self.\_listOfTextObjects = [self.\_\_winlossText, self.\_\_infoText1, self.\_\_infoText2]

    def \_HandleInputs(self):

        super().\_HandleInputs()

        i = 0

        while i < len(self.\_inputHandler.inputsList):

            if self.\_inputHandler.inputsList[i][:6] == "CLICK:":

                clickLocation = self.\_inputHandler.inputsList[i][6:].split(",")

                clickLocation = (int(clickLocation[0]), int(clickLocation[1]))

                if self.\_\_menuButton.CheckForCollision(clickLocation):

                    self.menuButtonPressed = True

                    self.\_\_menuButton.clicked = True

                self.\_inputHandler.inputsList.pop(i)

            else:

                i += 1

## Server

The server needs to send the player how much Elo they gained or lost, the win/loss margin and whether they won or lost.

The section that is responsible for this is in the Race stage of the server Game object.

### Pseudocode

IF winner IS NOT None THEN

winner.msgsToSend.Enqueue("!MATCHOUTCOME:WIN")

loser.msgsToSend.Enqueue("!MATCHOUTCOME:LOSS")

winner.msgsToSend.Enqueue("!MARGIN:" + winMargin)

loser.msgsToSend.Enqueue("!MARGIN:" + winMargin)

winner.gamesWon <-- winner.gamesWon + 1

winner.gamesPlayed <-- winner.gamesPlayed + 1

winner.currentWinstreak <-- winner.currentWinstreak + 1

IF winner.currentWinstreak > winner.longestStreak THEN

winner.longestStreak <-- winner.currentWinstreak

loser.gamesPlayed <-- loser.gamesPlayed + 1

loser.currentWinstreak <-- 0

winner.sumOfOpponentsElo <-- winner.sumOfOpponentsElo + loser.Elo

loser.sumOfOpponentsElo <-- loser.sumOfOpponentsElo + winner.Elo

winnerGamesLost <-- winner.gamesPlayed - winner.gamesWon

#This is the formula for calculating PR (Performance Rating) increase

EloDiff <-- (winner.sumOfOpponentsElo + 400 \* (winner.gamesWon - winnerGamesLost)) / winner.gamesPlayed - winner.Elo

winner.msgsToSend.Enqueue("!ELO:" + EloDiff)

winner.Elo <-- winner.Elo EloDiff

IF winner.Elo > winner.highestElo THEN

winner.highestElo <-- winner.Elo

ENDIF

loserGamesLost <-- loser.gamesPlayed - loser.gamesWon

EloDiff <-- loser.Elo - (loser.sumOfOpponentsElo + 400 \* (loser.gamesWon - loserGamesLost)) / loser.gamesPlayed

loser.msgsToSend.Enqueue("!ELO:" + EloDiff)

loser.Elo <-- loser.Elo - EloDiff

ELSE

self.\_SendMessageToBothPlayers("!MATCHOUTCOME:DRAW")

ENDIF

### Python code

        if winner is not None:

            winner.msgsToSend.Enqueue("!MATCHOUTCOME:WIN")

            loser.msgsToSend.Enqueue("!MATCHOUTCOME:LOSS")

            winner.msgsToSend.Enqueue(f"!MARGIN:{winMargin}")

            loser.msgsToSend.Enqueue(f"!MARGIN:{winMargin}")

            winner.gamesWon += 1

            winner.gamesPlayed += 1

            winner.currentWinstreak += 1

            if winner.currentWinstreak > winner.longestStreak:

                winner.longestStreak = winner.currentWinstreak

            loser.gamesPlayed += 1

            loser.currentWinstreak = 0

            winner.sumOfOpponentsElo += loser.Elo

            loser.sumOfOpponentsElo += winner.Elo

            winnerGamesLost = winner.gamesPlayed - winner.gamesWon

            EloDiff = (winner.sumOfOpponentsElo + 400 \* (winner.gamesWon - winnerGamesLost)) / winner.gamesPlayed - winner.Elo

            winner.msgsToSend.Enqueue(f"!ELO:{EloDiff}")

            winner.Elo += EloDiff

            if winner.Elo > winner.highestElo:

                winner.highestElo = winner.Elo

            loserGamesLost = loser.gamesPlayed - loser.gamesWon

            EloDiff = loser.Elo - (loser.sumOfOpponentsElo + 400 \* (loser.gamesWon - loserGamesLost)) / loser.gamesPlayed

            loser.msgsToSend.Enqueue(f"!ELO:{EloDiff}")

            loser.Elo -= EloDiff

        else:

            self.\_SendMessageToBothPlayers("!MATCHOUTCOME:DRAW")

## Game

The game object needs to check if the user pressed the back button in order to take them back to the main menu.

### Pseudocode

self.\_\_WaitForMatchResult()

self.\_\_postGameScreen = PostGame object

self.\_\_PostGame()

IF self.\_\_userQuit THEN RETURN 0

ENDIF

### Python code

self.\_\_WaitForMatchResult()

*#Goes to post game screen*

self.\_\_postGameScreen = PostGame(self.\_\_window, self.\_\_resolution, self.\_\_results, self.\_\_margin, self.\_\_ELodiff, self.socket)

self.\_\_PostGame()

if self.\_\_userQuit:

    return 0

WaitForMatchResult()

def \_\_WaitForMatchResult(self):

    while self.\_\_results is None or self.\_\_margin is None or self.\_\_ELodiff is None:

        self.\_\_CheckMessages()

PostGame()

def \_\_PostGame(self):

    while not self.\_\_postGameScreen.menuButtonPressed:

        self.\_\_postGameScreen.main()

        if self.\_\_postGameScreen.userQuit:

            self.\_\_userQuit= True

            return 0

        pygame.display.update()