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.

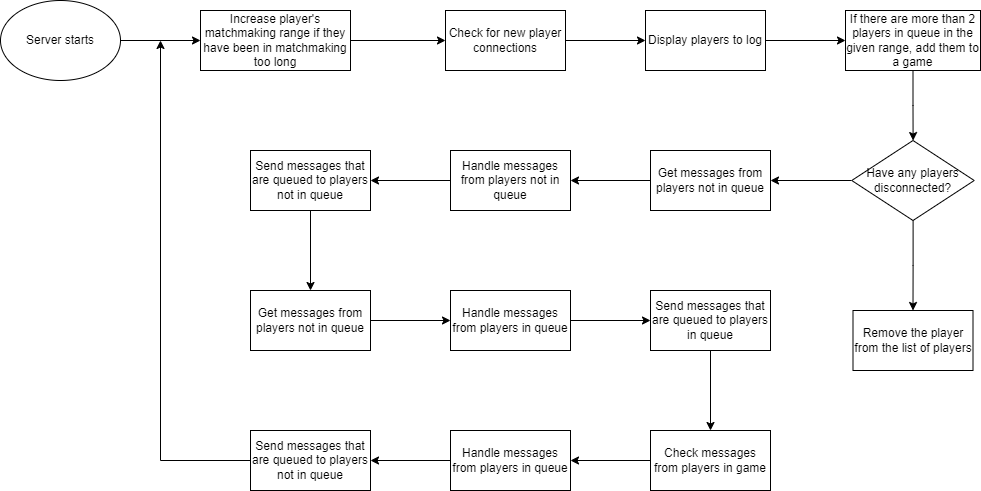
I initially tried to make them run in separate threads, but this quickly brought up issues with synchronising multiple threads. I then tried synchronous, which was because I learned of non-blocking sockets. This was a lot more successful and is in the final implementation of the final design for the 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 make 2 sections, 1 for server and 1 for client. Client will be first but I will be working on the server at the same time, but for organisation sake I have kept them separate in the documentation. 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.

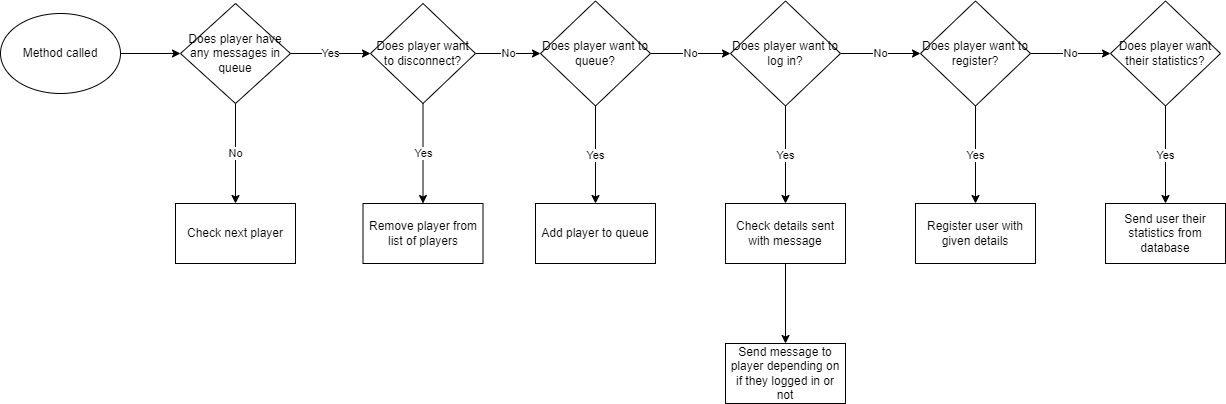
Flowchart for server finding connections to clients:



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:

## UML diagram

I would prefer to have this section below the image but to not waste a whole page I’ll put it here since it doesn’t make much difference. 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, the python code for this is below.

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.

### Python code for Play.py

*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:

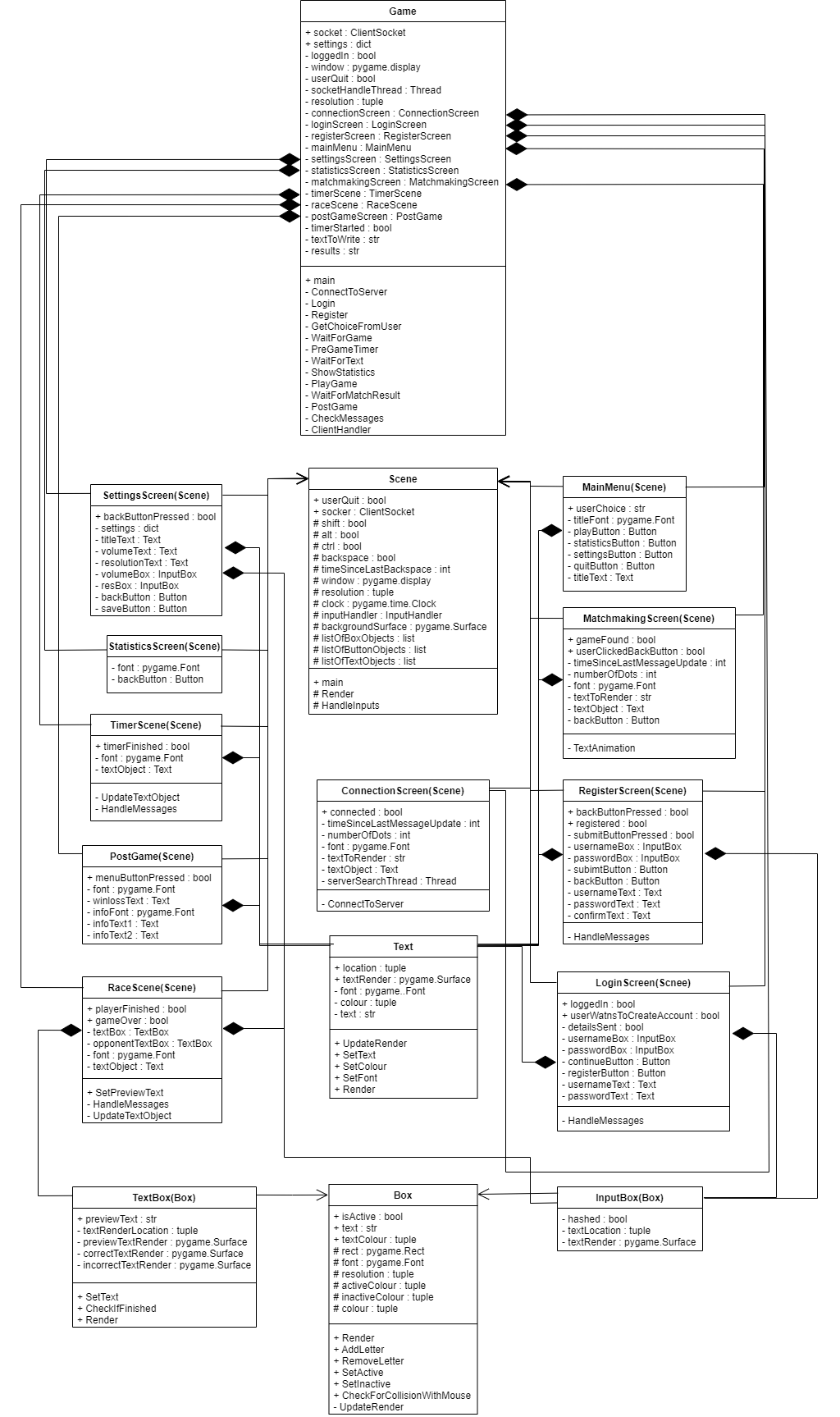
    game.socket.EndConnection()

pygame.font.quit()

pygame.quit()

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 made and documented earlier in this document. 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.

### FlowchartDiagram 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 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.

# Class Box

This box class needs to be able to hold coordinates, states and other information about a textbox.

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

## 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.

### 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.

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 for InputBox:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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:

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:

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():

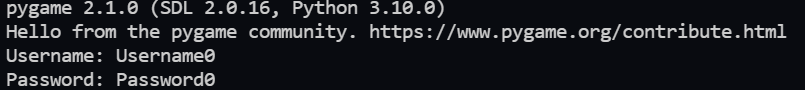
*if* event.type == pygame.QUIT:

            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.

## 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.Pressed()

*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

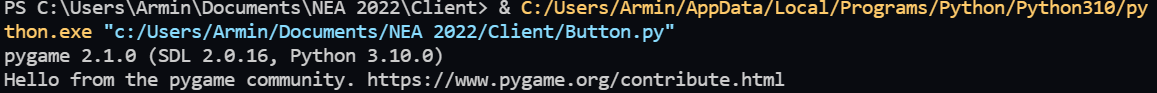
The colour has changed when I hover over the button so this test is passed.

### 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.

## 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)

# Scene

# Account Creation Screen

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

# 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

The password box will also need to be starred out when the user types in it. When the user presses continue the details should be checked with the server and then logged in if they are correct.

import pygame

from Button import Button

from InputBox import InputBox

class LoginScreen:

    def \_\_init\_\_(self, screenDimensions, socket):

*self*.\_\_socket = socket

*self*.\_\_screenDimensions = screenDimensions

        dispWidth = screenDimensions[0]

        dispHeight = screenDimensions[1]

*#Need 2 textboxes*

        boxColourActive = (35,35,35)

        boxColourDormant = (30,30,30)

*self*.\_\_textColour = (255,255,255)

        x = 999

        font = pygame.font.SysFont("Courier New", x)

        while font.size("Username")[1] > 40:

            x -= 1

            font = pygame.font.SysFont("Courier New", x)

*self*.\_\_font = font

*#Boxlocations*

        rectangleUsernameBox = pygame.rect.Rect((0,int(400 \* dispHeight/1080)), (int(1000 \* dispWidth / 1920), int(70 \* dispHeight / 1080)))

        rectangleUsernameBox.centerx = dispWidth / 2

*self*.\_\_usernameBox = InputBox((255,255,255), boxColourActive, boxColourDormant, False, rectangleUsernameBox, dispHeight)

        rectanglePasswordBox = pygame.rect.Rect((0,int(600 \* dispHeight/1080)), (int(1000 \* dispWidth / 1920), int(70 \* dispHeight / 1080)))

        rectanglePasswordBox.centerx = dispWidth / 2

*self*.\_\_passwordBox = InputBox((255,255,255), boxColourActive, boxColourDormant, True, rectanglePasswordBox, dispHeight)

*#Need confirm button*

        buttonSize = (int(400 \* dispWidth / 1920), int(60 \* dispHeight / 1080))

*self*.\_\_continueButton = Button("Continue", (0,0), buttonSize, boxColourDormant, boxColourActive, *self*.\_\_textColour, dispHeight)

        newCoords = (dispWidth / 2, 730 \* dispHeight / 1080)

*self*.\_\_continueButton.SetLocation(newCoords)

## LoginScreen.main()

This method needs to contain the main loop when the user is logging in. This is the first thing the player sees when they open the program.

    def main(self, window):

*#Variables used for conditions*

        backspace = False

        timeSinceLastBackspace = 0

        timeBetweenBackspaces = 50

        usernameBoxSelected = False

        passWordBoxSelected = False

        control = False

        clock = pygame.time.Clock()

        loggedIn = False

        playerQuit = False

        waiting = False

        while not loggedIn and not playerQuit:

*#Deletes all userinput when client is waiting for a response*

            if waiting:

                pygame.event.clear()

                control = False

                backspace = False

*#Handles all the input while the player is typing*

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    playerQuit = True

*self*.\_\_socket.EndConnection()

                elif event.type == pygame.MOUSEBUTTONDOWN:

                    clickLocation = pygame.mouse.get\_pos()

                    if *self*.\_\_usernameBox.CheckIfClicked(clickLocation):

*self*.\_\_usernameBox.SetActive()

                        usernameBoxSelected = True

                    else:

*self*.\_\_usernameBox.SetDormant()

                        usernameBoxSelected = False

                    if *self*.\_\_passwordBox.CheckIfClicked(clickLocation):

*self*.\_\_passwordBox.SetActive()

                        passWordBoxSelected = True

                    else:

*self*.\_\_passwordBox.SetDormant()

                        passWordBoxSelected = False

                    if *self*.\_\_continueButton.CheckIfHovering(clickLocation):

*self*.\_\_continueButton.Pressed()

                elif event.type == pygame.KEYDOWN:

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

                        backspace = True

                        if *self*.\_\_usernameBox.isActive:

*self*.\_\_usernameBox.RemoveLetter(control)

                        elif *self*.\_\_passwordBox.isActive:

*self*.\_\_passwordBox.RemoveLetter(control)

                        timeSinceLastBackspace = -200

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

*self*.\_\_continueButton.Pressed()

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

                        control = True

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

                        if *self*.\_\_usernameBox.isActive:

*self*.\_\_usernameBox.SetDormant()

*self*.\_\_passwordBox.SetActive()

                        else:

*self*.\_\_passwordBox.SetDormant()

*self*.\_\_usernameBox.SetActive()

                    else:

                        if *self*.\_\_usernameBox.isActive:

*self*.\_\_usernameBox.AddLetter(event.unicode)

                        elif *self*.\_\_passwordBox.isActive:

*self*.\_\_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

*#Checks if autobackspace needs to delete a letter*

            if backspace and timeSinceLastBackspace > timeBetweenBackspaces:

                if usernameBoxSelected:

*self*.\_\_usernameBox.RemoveLetter(control)

                elif passWordBoxSelected:

*self*.\_\_passwordBox.RemoveLetter(control)

                timeSinceLastBackspace = 0

*#Checks if the box should be highlighted*

            if *self*.\_\_continueButton.CheckIfHovering(pygame.mouse.get\_pos()):

*self*.\_\_continueButton.SetActive()

            else:

*self*.\_\_continueButton.SetDormant()

*#Checks if the login details need to be sent to the server*

            if *self*.\_\_continueButton.GetPressedState() and not waiting:

                username = *self*.\_\_usernameBox.text

                password = *self*.\_\_passwordBox.text

*self*.\_\_socket.msgsToSend.append(f"!LOGIN:{username},{password}")

*self*.\_\_continueButton.SetText("Loading")

                waiting = True

*#Handles messages from server*

            while *self*.\_\_socket.receivedMsgs != []:

                if *self*.\_\_socket.receivedMsgs[0] == "!PASSWORDCORRECT":

                    loggedIn = True

                elif *self*.\_\_socket.receivedMsgs[0] == "!PASSWORDINCORRECT":

*self*.\_\_continueButton.DePressed()

                    waiting = False

*self*.\_\_usernameBox.text = ""

*self*.\_\_passwordBox.text = ""

*self*.\_\_continueButton.SetText("Continue")

*self*.\_\_socket.receivedMsgs.pop(0)

*#Time*

            clock.tick()

            timeSinceLastBackspace += clock.get\_time()

*#Rendering screen*

            window.fill((10,10,10))

            textRender = *self*.\_\_font.render("Username", True, *self*.\_\_textColour)

            window.blit(textRender, (*self*.\_\_usernameBox.rectangle.x + 10 \* *self*.\_\_screenDimensions[0] / 1920, *self*.\_\_usernameBox.rectangle.y - 90 \* *self*.\_\_screenDimensions[1] / 1080))

            textRender = *self*.\_\_font.render("Password", True, *self*.\_\_textColour)

            window.blit(textRender, (*self*.\_\_passwordBox.rectangle.x + 10 \* *self*.\_\_screenDimensions[0] / 1920, *self*.\_\_passwordBox.rectangle.y - 90 \* *self*.\_\_screenDimensions[1] / 1080))

*self*.\_\_usernameBox.DrawBox(window)

*self*.\_\_passwordBox.DrawBox(window)

*self*.\_\_continueButton.Render(window)

            pygame.display.update()

*#Returns false if the player quit before logging in*

        if playerQuit:

            return False

        else:

            return True

# Main Menu

The game will need a main menu on the client’s side. This is achieved by having a separate gameloop where the player has logged in and is presented with the options available. I made a quick design for this which just shows the name of the game (SpeedTyper) and then the 4 options that were mentioned in the objectives list.

Graphical user interface, application

Description automatically generated

The Button class was used here again but instead of having a single button a list of buttons was required to make managing the buttons easier.

class MainMenu:

    def \_\_init\_\_(self, screenDimensions):

*self*.\_\_font = pygame.font.SysFont("Calibri", int(96 \* screenDimensions[1] / 1080))

*self*.\_\_textRenderSize = *self*.\_\_font.size("SpeedTyper")

*self*.\_\_screenDimensions = screenDimensions

        textColour = (160,160,160)

        titleColour = (255,144,8)

*self*.\_\_textRender = *self*.\_\_font.render("SpeedTyper", True, titleColour)

*#For the buttons*

        boxColourActive = (35,35,35)

        boxColourDormant = (30,30,30)

        buttonSize = (680 \* screenDimensions[0] / 1920, 150 \* screenDimensions[1] / 1080)

*self*.\_\_buttons = []

        screenCenter = screenDimensions[0] / 2 - buttonSize[0] / 2

*self*.\_\_buttons.append(Button("Play", (screenCenter, 250 \* screenDimensions[1] / 1080), buttonSize, boxColourDormant, boxColourActive, textColour, screenDimensions[1]))

*self*.\_\_buttons.append(Button("Statistics", (screenCenter, 450 \* screenDimensions[1] / 1080), buttonSize, boxColourDormant, boxColourActive, textColour, screenDimensions[1]))

*self*.\_\_buttons.append(Button("Settings", (screenCenter, 650 \* screenDimensions[1] / 1080), buttonSize, boxColourDormant, boxColourActive, textColour, screenDimensions[1]))

*self*.\_\_buttons.append(Button("Quit", (screenCenter, 850 \* screenDimensions[1] / 1080), buttonSize, boxColourDormant, boxColourActive, textColour, screenDimensions[1]))

*#Overrides all buttons' fonts to have same size*

        for button in *self*.\_\_buttons:

            button.SetFont(*self*.\_\_font)

In the start you can see the 3 lines of code used to make the SpeedTyper text at the top. The font size was chosen to be 96 as that was the font size I used when designing the menu and it is multiplied by the ratio of the current display height / 1080 (to keep proportions the same).

The text colour was chosen by using the colour picker tool in Paint.net and checking the RGB values.

A pygame surface object is then made called self.\_\_textRender which is able to be drawn onto the main surface during the game loop, specifically in the Render() method of this class.

Next some button colours are initialised. Active being the colour when the user is hovering over the button and dormant being whenever the user isn’t.

The button size is calculated by using Paint.net’s area select tool to determine the width and height of the boxes in the design. These were then rounded to the nearest 10 and made to be proportional to the display resolution.

4 button objects are appended to the button list that will be being used everywhere else and the button’s y coordinates are calculated while initialising the button object.

Since the button was designed to make the text within it fit automatically, to keep the font size the same we need to override the font object in the button object. This is done at the end of the initialising section.

## MainMenu.Render()

This needs to simply display the text and the buttons on the screen.

    def Render(self, window):

        window.blit(*self*.\_\_textRender, (*self*.\_\_screenDimensions[0] / 2 - *self*.\_\_textRenderSize[0] / 2, 110 \* *self*.\_\_screenDimensions[1] / 1080))

        for button in *self*.\_\_buttons:

            button.Render(window)

## MainMenu.Run()

This is the section of the code that is run when the player has logged in successfully and so has it’s own input handling and such.

    def Run(self, window, settings):

        running = True

        playerQuit = False

        while running:

            mousePos = pygame.mouse.get\_pos()

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    running = False

                    playerQuit = True

                elif event.type == pygame.MOUSEBUTTONDOWN:

                    for button in *self*.\_\_buttons:

                        if button.CheckIfHovering(mousePos):

                            button.Pressed()

            for button in *self*.\_\_buttons:

                if button.CheckIfHovering(mousePos):

                    button.SetActive()

                else:

                    button.SetDormant()

*#If button is pressed*

                if button.GetPressedState():

*#Goes through button text to determine what button it is*

                    if button.GetText() == "Play":

*#Goes back to Game object and continues to the main game*

                        running = False

                    elif button.GetText() == "Statistics":

*#Go to statistics*

                        pass

                    elif button.GetText() == "Settings":

*#Go to settings*

                        pass

*#Makes it so that the Game object stops looping and the game closes*

                    elif button.GetText() == "Quit":

                        running = False

                        playerQuit = True

*#Rendering*

            window.fill((10,10,10))

*self*.Render(window)

            pygame.display.update()

        if playerQuit:

            return False

        else:

            return True

This will handle all the things needed to allow the user to click on a button and then calls another method that will do what the button is meant to do. This method will return False if the player has quit and True if they haven’t. This is used in the Game object to determine if the player should be put into matchmaking or not.

# Game (Client)

This is the object that the game occurs in, this will handle the actual “race” itself, where the players go head-to-head with each other to type the words the fastest. Currently the game has the functionality to allow players to type the same phrase as the other person, and to connect to each other also.

This object takes all other objects in this section and combines them and allows for communication with the server to be integrated into the game. This means that the game will be able to get messages from the server and translate those into reactions for the program to show to the user. This sets up other features to work much more easily and allows for other methods to be created for different things such as the menu, countdown to game start or anything else that the client needs to do.

## Python code for game

import pygame

import threading

from TextBox import TextBox

from Renderer import Renderer

from ClientSocket import ClientSocket

from LoginScreen import LoginScreen

class Game:

    def \_\_init\_\_(self, dispWidth, dispHeight):

*self*.\_\_gameClock = pygame.time.Clock()  *#Makes a clock object*

*self*.\_\_timeBetweenBacspaces = 50        *#Delay between backspaces when backspace is held down*

*self*.\_\_timeSinceLastBackspace = 0

*self*.\_\_deleting = False

*self*.\_\_ctrl = False                     *#Boolean that is true for the duration of the backspace key being held down*

*self*.\_\_renderer = Renderer(dispHeight)            *#Creates Renderer object*

*self*.\_\_backText = " "

*self*.connected = True

*self*.userQuit = False

*self*.\_\_timerUntilGameStart = 0

*self*.\_\_gameTimer = 30

*self*.\_\_timeSinceLastCountdown = 0

*self*.\_\_dispWidth = dispWidth

*self*.\_\_dispHeight = dispHeight

*self*.ConnectToServer()

*self*.\_\_loginScreen = LoginScreen((dispWidth, dispHeight), *self*.clientSocket)

        font = pygame.font.SysFont("Courier New", int(dispHeight\*42/1080))  *#sets font to Courier New (font with constant letter size)*

*self*.\_\_textBox = TextBox(int(*self*.\_\_dispWidth - (*self*.\_\_dispWidth \* 2/5)), int(50 \* *self*.\_\_dispHeight / 1080), (int(*self*.\_\_dispWidth / 5), int(6 \* *self*.\_\_dispHeight / 20)), (40,40,40), (30,30,30), (255,144,8), font, (160,160,160))

*#A thread that will get messages and send messages to the server*

*self*.\_\_SocketHandleThread = threading.Thread(target=*self*.\_\_HandleSocket, daemon=True)

*self*.\_\_SocketHandleThread.start()

    def main(self, window):

*self*.\_\_window = window

*#Ends program if no server was found before player quit*

        if not *self*.\_\_serverFound:

            return "Player quit while looking for server"

        else:

*self*.clientSocket.msgsToSend.append("[Connection established with client]")

            print("Connected to server")

        if not *self*.\_\_loginScreen.main(*self*.\_\_window):

            return "Player quit while logging in"

*self*.timerActive = False

*#Queues into matchmaking*

*self*.clientSocket.msgsToSend.append("!QUEUE")

        timeSinceLastTimerUpdate = 0

*#Main loop starts here*

        while *self*.\_\_gameTimer >= 0:

*#Checks if user is still connected*

            if *self*.userQuit:

*self*.clientSocket.EndConnection()

                break

*#Draws the background and empty textbox*

*self*.\_\_renderer.Render(*self*.\_\_window, *self*.\_\_textBox)

*self*.\_\_gameClock.tick()

*#Handles userinput*

*self*.\_\_HandleInput()

*self*.\_\_CheckForBackspace()

*self*.\_\_timeSinceLastBackspace += *self*.\_\_gameClock.get\_time()

*#Handles messages from server*

*self*.\_\_HandleMessages()

            if *self*.\_\_backText == " ":

*#If game has not started*

                if *self*.timerActive:

*#Display seconds left until start and removes time since last frame from timer*

*self*.\_\_renderer.RenderTimer(*self*.\_\_window, (*self*.\_\_dispWidth, *self*.\_\_dispHeight), *self*.\_\_timerUntilGameStart)

*self*.\_\_timeSinceLastCountdown += *self*.\_\_gameClock.get\_time()

                    if *self*.\_\_timeSinceLastCountdown >= 1000:

*self*.\_\_timerUntilGameStart -= 1

*self*.\_\_timeSinceLastCountdown -= 1000

                    if *self*.\_\_timerUntilGameStart <= 0:

*self*.timerActive = False

                else:

*self*.\_\_renderer.RenderWaitingText(*self*.\_\_window, (*self*.\_\_dispWidth, *self*.\_\_dispHeight))

*#When game has started*

            elif *self*.\_\_gameTimer >= 0:

*self*.\_\_renderer.RenderTimer(*self*.\_\_window, (*self*.\_\_dispWidth, *self*.\_\_dispHeight), *self*.\_\_gameTimer)

                timeSinceLastTimerUpdate += *self*.\_\_gameClock.get\_time()

*#Every second displays the current time left*

                if timeSinceLastTimerUpdate >= 1000:

*self*.\_\_gameTimer -= 1

                    timeSinceLastTimerUpdate -= 1000

            pygame.display.update()

*self*.clientSocket.EndConnection()

    def ConnectToServer(self):

*#Attempts to connect to the server, will continue indefinitely until a server is found*

*self*.\_\_serverFound = False

*self*.\_\_serverSearchThread = threading.Thread(target=*self*.\_\_SearchForServer)

*self*.\_\_serverSearchThread.start()

        while not *self*.\_\_serverFound and not *self*.userQuit:

*self*.\_\_CheckIfUserQuit()

    def \_\_HandleMessages(self):

        for msg in *self*.clientSocket.receivedMsgs:

            if msg == "!DISCONNECT":

*self*.clientSocket.connected = False

            elif msg[:10] == "!BACKTEXT:":

*self*.\_\_backText = msg[10:]

*#Creates a textbox object and passes arguments through it // refer to TextBox.py*

*self*.\_\_textBox.SetPreviewText(*self*.\_\_backText)

*self*.timerActive = False

            elif msg[:23] == "!SECONDSLEFTUNTILSTART:":

*self*.timerActive = True

*self*.\_\_timerUntilGameStart = int(msg[23:]) - 2

*self*.clientSocket.receivedMsgs = []

*#Tries to connect to server, used in init to allow instant quitting when user alt+f4*

*#This runs in another thread*

    def \_\_SearchForServer(self):

        while not *self*.\_\_serverFound and not *self*.userQuit:

            try:

*self*.clientSocket = ClientSocket()

*self*.\_\_serverFound = True

            except:

                print("Failed to connect to server, trying again")

*#Returns true if the player tries to quit the game*

*#Used in the init for Client to allow player to quit while it is searching for a server*

    def \_\_CheckIfUserQuit(self):

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

*self*.userQuit = True

*#Used to translate player input into actions on screen, such as typing a letter or deleting a letter*

    def \_\_HandleInput(self):

        for event in pygame.event.get():

            if event.type == pygame.QUIT:           *#If alt + f4 pressed or quit button (in the future)*

*self*.userQuit = True

            elif event.type == pygame.MOUSEBUTTONDOWN:  *#When mouse is clicked*

                clickLocation = pygame.mouse.get\_pos()

                if *self*.\_\_textBox.box.collidepoint(clickLocation):

*self*.\_\_textBox.SetActive()  *#Sets textbox to be active if it was clicked on*

                else:

*self*.\_\_textBox.SetDormant() *#Sets textbox to be dormant if anywhere else clicked*

            elif event.type == pygame.KEYDOWN:  *#When button is pressed*

                if event.key == pygame.K\_BACKSPACE: *#When backspace pressed*

*self*.\_\_deleting = True

                    if *self*.\_\_textBox.isActive():

*self*.\_\_textBox.DeleteLetter(*self*.\_\_ctrl)    *#Deletes letter*

*self*.\_\_timeSinceLastBackspace = -200    *#Causes delay until letters are deleted automatically*

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

                    pass

                elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:   *#Used for deleting entire letters*

*self*.\_\_ctrl = True

                else:

                    if *self*.\_\_textBox.isActive():

*self*.\_\_textBox.AddLetter(event.unicode) *#Adds letter to textbox if anything else is pressed*

            elif event.type == pygame.KEYUP:

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

*self*.\_\_deleting = False

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

                    pass

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

*self*.\_\_ctrl = False

    def \_\_CheckForBackspace(self):

*#Deletes text while backspace being held down*

        if *self*.\_\_deleting and *self*.\_\_timeSinceLastBackspace > *self*.\_\_timeBetweenBacspaces and *self*.\_\_inputHandler.typing:

*self*.\_\_textBox.DeleteLetter(*self*.\_\_ctrl)

*self*.\_\_timeSinceLastBackspace = 0

    def \_\_HandleSocket(self):

        while *self*.clientSocket.connected or *self*.clientSocket.msgsToSend != []:

*self*.clientSocket.SendMsgs()

*self*.clientSocket.GetMsgs()

The client can connect to a server, get messages from the server and do things with the information. In the code above, the client waits for a connection to the server, takes a phrase from the server and displays it to allow the player to type it in. There were some issues with implementing the ClientSocket class into this, because the ClientSocket class would freeze the program until a connection was made with the server. To fix this I made it so that the program continually checks for a connection with the server in a different thread, and while that is happening the player will be able to close the program.

It was also a problem that the player would not be able to quit while the server is being connected to, but this was fixed by running that function in another thread and allowing the client to check for inputs simultaneously.

This also had the effect of causing the player to run the game anyway, because of how the main function for the client worked, so I made it so that the object had a Boolean attribute dependant on if the player has quit or not, and if so then the main function would simply return a pre-determined message.

*#Ends program if no server was found before player quit*

        if not *self*.\_\_serverFound:

            return "Player quit while looking for server"

## Background Text Retrieval from server

The game will constantly check for messages from the server. The game will also check if the message starts with “!BACKTEXT:” and if so then it will set the background text to anything that comes after that. This only works if the textbox has functionality to act differently depending on if there is background text present or not, otherwise typing anything would simply crash the game. This is easily solved by checking the length of the background text before rendering or changing it.

## Game.\_\_SearchForServer()

The client needs to be able to find the server, this calls for a subroutine that will search for the server and try again until it finds one. It also needs to be able to stop if the user has quit. Diagram

Description automatically generated

*#Tries to connect to server, used in init to allow instant quitting when user alt+f4*

*#This runs in another thread*

    def \_\_SearchForServer(self):

        while not *self*.\_\_serverFound and not *self*.userQuit:

            try:

*self*.clientSocket = ClientSocket()

*self*.\_\_serverFound = True

            except:

                print("Failed to connect to server, trying again")

## Game.\_\_CheckIfUserQuit()

The client will need to be able to close properly while searching for a game. This means that a script will need to be running that checks the user’s input to determine if they have quit.

Diagram

Description automatically generated

*#Returns true if the player tries to quit the game*

*#Used in the init for Client to allow player to quit while it is searching for a server*

    def \_\_CheckIfUserQuit(self):

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

*self*.userQuit = True

## Game.\_\_HandleInput()

The client will need to be able to change things based off of certain inputs. This can be achieved by checking the pygame.event.get() function to retrieve a list of all the events that have happened in that frame. This can then be iterated through to make something happen when specific inputs occur.

Diagram

Description automatically generated

*#Used to translate player input into actions on screen, such as typing a letter or deleting a letter*

    def \_\_HandleInput(self):

        for event in pygame.event.get():

            if event.type == pygame.QUIT:           *#If alt + f4 pressed or quit button (in the future)*

*self*.userQuit = True

            elif event.type == pygame.MOUSEBUTTONDOWN:  *#When mouse is clicked*

                clickLocation = pygame.mouse.get\_pos()

                if *self*.\_\_textBox.box.collidepoint(clickLocation):

*self*.\_\_textBox.SetActive()  *#Sets textbox to be active if it was clicked on*

                else:

*self*.\_\_textBox.SetDormant() *#Sets textbox to be dormant if anywhere else clicked*

            elif event.type == pygame.KEYDOWN:  *#When button is pressed*

                if event.key == pygame.K\_BACKSPACE: *#When backspace pressed*

*self*.\_\_deleting = True

                    if *self*.\_\_textBox.isActive():

*self*.\_\_textBox.DeleteLetter(*self*.\_\_ctrl)    *#Deletes letter*

*self*.\_\_timeSinceLastBackspace = -200    *#Causes delay until letters are deleted automatically*

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

                    pass

                elif event.key == pygame.K\_LCTRL or event.key == pygame.K\_RCTRL:   *#Used for deleting entire letters*

*self*.\_\_ctrl = True

                else:

                    if *self*.\_\_textBox.isActive():

*self*.\_\_textBox.AddLetter(event.unicode) *#Adds letter to textbox if anything else is pressed*

            elif event.type == pygame.KEYUP:

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

*self*.\_\_deleting = False

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

                    pass

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

*self*.\_\_ctrl = False

## Game.\_\_CheckForBackspace()

The game will need to check if the backspace key is held down and will then delete letters every set amount of time while it is being held down.

Diagram

Description automatically generated

    def \_\_CheckForBackspace(self):

*#Deletes text while backspace being held down*

        if *self*.\_\_deleting and *self*.\_\_timeSinceLastBackspace > *self*.\_\_timeBetweenBacspaces and *self*.\_\_inputHandler.typing:

*self*.\_\_textBox.DeleteLetter(*self*.\_\_ctrl)

*self*.\_\_timeSinceLastBackspace = 0

## Testing for Game:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | Client should be able to connect to the server | I will run the main.py file | The server terminal should show that a user has connected | Normal | Pass |
| 2 | Client should be able to get a phrase back from the server | I will run the main.py file twice to emulate 2 players joining the game | The clients should both display the same preview message in the textbox | Normal | Pass |
| 3 | Client should be able to close during matchmaking | I will run the main.py file once to prevent a game from starting, and then I will attempt to close the application | The client should close and should disconnect from the server | Normal | Pass |

## Test 1

The client needs to be able to connect to the server. This is tested by running the main.py file and checking if the server has established a connection with the client.

Text

Description automatically generated

The connection has been established and therefore this test is passed.

## Test 2

The client needs to be able to connect to the server and get a phrase back from the server to start the game. This is tested by running the main.py file twice and seeing if the textbox has a preview on it.

Graphical user interface, text

Description automatically generated

This test is passed.

## Test 3

The client needs to be able to be closed at any time so that the player does not need to use task manager to close it during matchmaking, so to test this I will be running main.py to check if the player can close the application when matchmaking.

The application closed when I pressed the red cross in the top right and so this test is passed.

## Main

Main.py will cause everything else to run and will also have the window resolution defined in it. Currently it is 2 variables to allow me to easily test things, but in the final product it will be automatically changed.

This script is very simple and was made when making the other parts, and so it didn’t have any planning.

*#main file*

import pygame

from Game import Game

dispWidth = 500

dispHeight = 500

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

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

pygame.font.init()

game = Game()

game.main(window, dispWidth, dispHeight)

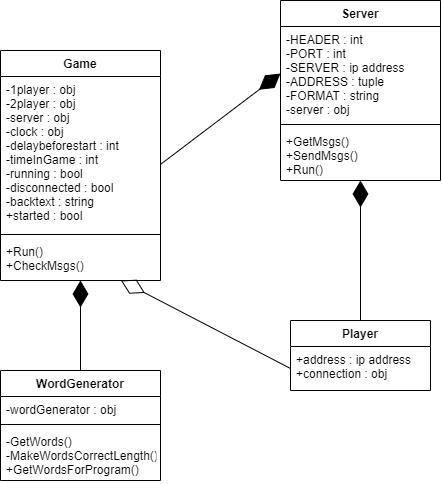
pygame.font.quit()

pygame.quit()

The testing for this section has been effectively done in the previous sections.

# Server

The server will take multiple connections from clients, store all the clients in a list and then allow them to play against each other in a game object. A UML diagram for this is below:



The WordGenerator object is documented in this previously and has not changed.

## Player

The player object is simply an object with 4 attributes, that allows for the retrieval of the socket object used to communicate between the server and client and the IP address. It also allows for accessing the messages that need to be sent to a specific player or the messages received from these players. This object is usually found in a list of players in the Server object and will likely be iterated through in a separate thread to get messages and to send them.

## Python code for player:

class Player:

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

*self*.address = address

*self*.connection = connection

*self*.msgsToSend = []

*self*.msgsReceived = []

    def SendMsg(self, msg):

*self*.msgsToSend.append(msg)

## Server

The server is the same as before, in that it was documented in the connecting a client and server section, however the Run method has been amended to allow players to get into a game together.

This was done by adding players to a list, and the server constantly checking whether it needs to add, remove or create a new game based on who’s in that list. Currently the matchmaking system isn’t in place, but it would be easy to implement.

## Python code for Server.Run()

    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()

## Testing for Server:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | The server needs to add players who connect to a list of players waiting for a game | I will add a print statement that prints the list at every frame, and I will then launch the client’s main.py | The player object will be printed after they have connected | Normal | Pass |
| 2 | The server needs to create a new game object for players when there are at least 2 of them waiting | I will launch the client’s main.py file twice | The client should display the random phrases and the game should start | Normal | Pass |
| 3 | The server needs to remove players from the queue after they have been sent to a game | I will add a print statement that prints the list at every frame, and I will then launch the client’s main.py twice | The list will be empty after both players’ game have started | Normal | Pass |
| 4 | The server will remove a player from the matchmaking queue if they close the application before a game starts | I will add a print statement that prints the list at every frame, and I will then launch the client’s main.py. I’ll then close the client file. | The list should become empty | Normal | Pass |

## Test 1

The server will need to add new players to a queue in order to find them an opponent. To test this I will add the following print statement to the main loop for the server and check if the client shows up in the list.

print(playersInMatchmaking)

Before launching client main.py:

Icon

Description automatically generated

After launching client main.py:

Text

Description automatically generated

The client is added to the queue and so this test is passed.

## Test 2

The server needs to make a new instance of a game, a game object, and so I can test this by seeing if the players are in game. This can be done easily by checking if once 2 players have launched the game, whether their phrase shows up in the textbox.

## Test 3

## Test 4

If a player leaves the game early then they must be removed from the matchmaking queue, to test this I will be doing the same thing as in test 1 but closing the client after. The list should become empty.

Before client opened:

Icon

Description automatically generated

After client opened:

Graphical user interface, text, application, chat or text message

Description automatically generated

After client closed:

Icon

Description automatically generated with low confidence

The player was removed from the queue and so this test is passed.

## Game (Server)

The Game object is an object which the game will be happening in for the server. This will include everything that happens in the actual game. This object is usually run in a separate thread for every 2 players that are playing.

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

*self*.player1 = player1

*self*.player2 = player2

*self*.\_\_server = server

*self*.\_\_clock = pygame.time.Clock()  *#Pygame clock object*

*self*.started = False

*self*.\_\_backTextSent = False

*self*.\_\_timerSent = False

*self*.\_\_delayBeforeStart = 5 *#Seconds before game starts*

*self*.\_\_timeInGame = 30      *#Seconds before game ends*

*self*.\_\_running = True

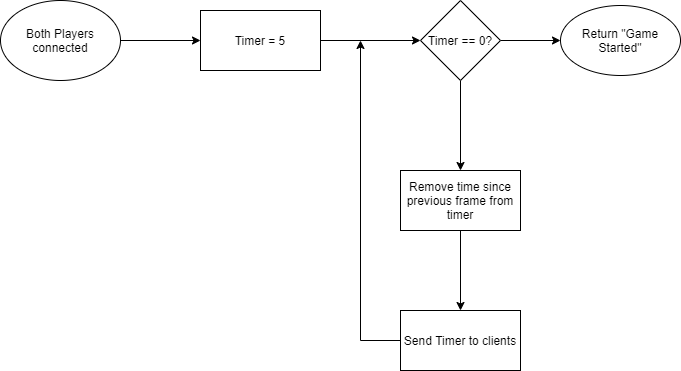
*self*.\_\_disconnected = ""

*self*.\_\_timeSinceLastMessage = 1000 *#milliseconds*

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

## Game.\_\_Countdown()

The game object will have the method Countdown which will count down from a given time and update the clients until it has reached 0. This is going to be used before the words are generated to make sure both players start at the same time.



*#This method counts down from timer seconds and updates the client on this*

    def \_\_Countdown(self):

        if not *self*.\_\_timerSent:

*#Sends message to clients to start timer*

            msg = f"!SECONDSLEFTUNTILSTART:{*self*.\_\_delayBeforeStart}"

*self*.\_\_SendMsgToBothPlayers(msg)

*self*.\_\_delayBeforeStart \*= 1000

*self*.\_\_timerSent = True

*#Waits until the timer has run out*

*self*.\_\_delayBeforeStart -= *self*.\_\_clock.get\_time()

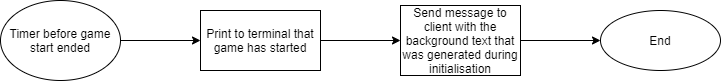
*#If the time has run out*

        if *self*.\_\_delayBeforeStart <= 0:

*self*.started = True

## Game.\_\_SendBackgroundText

The server will need to send the background text to the client once the game has started. This is handled in the SendBackgroundText() method.



def \_\_SendBackgroundText(self):

    print("Got to game countdown")

    msg = f"BACKTEXT:{*self*.\_\_backText}"

*self*.\_\_SendMsgToBothPlayers(msg)

## Game.\_\_SendMsgToBothPlayers()

The server will usually need to send the same message to both players and it is easier to do this than to type it out every time.

*#Sends the same message to both players*

    def \_\_SendMsgToBothPlayers(self, msg):

*self*.player1.SendMsg(msg)

*self*.player2.SendMsg(msg)

## Game.CheckMsgs()

The server needs to check through every message the players have sent. This method will check through every message for both players.

    def CheckMsgs(self):

        for msg in *self*.player1.msgsReceived:

            if msg == " ":

                pass

            elif msg == "!DISCONNECT":

*self*.\_\_running = False

*self*.\_\_disconnected = "player1"

            elif msg[:8] == "!LETTER:":

*#Do things with letter*

                pass

        for msg in *self*.player2.msgsReceived:

            if msg == " ":

                pass

            elif msg == "!DISCONNECT":

*self*.\_\_running = False

*self*.\_\_disconnected = "player2"

            elif msg[:8] == "!LETTER:":

*#Do things with letter*

                pass

## Game.\_\_Run()

This is the main function of the game and it can be private as it does not need to be called outside the program. This is because it is called in the \_\_init\_\_() method that is called when the object is first created.

    def \_\_Run(self):

*self*.\_\_backText = WordGenerator().GetWordsForProgram(500)

*#Main Loop for the game*

        while *self*.\_\_running:

*self*.\_\_clock.tick()

            if not *self*.started:

*#Manages countdown for clients*

*self*.\_\_Countdown()

*#Sends background text if game has started but only does this once*

            elif not *self*.\_\_backTextSent and *self*.started:

*self*.\_\_SendBackgroundText()

*#Checks messages of both players*

*self*.CheckMsgs()

*#End of game*

        if *self*.\_\_disconnected != "":

            if *self*.\_\_disconnected == "player1":

                pass

*#Do something when player 1 has disconnected*

            else:

                pass

*#Do something when player 2 has disconnected*

        else:

            pass

*#Do something when the game ended normally*

## Game.StartThread()

    def StartThread(self):

*self*.\_\_gameThread.start()

This is used in the Server object to start the main gameloop.

# Button

The button will be used in the menus and in the login screen to allow the player to submit their details or to go to another menu.

This button class will have support for choosing location of the button, size and colours.

    def \_\_init\_\_(self, text, location, size, backColourDormant, backColourActive, textColour, dispHeight):

*self*.\_\_pressed = False

*self*.\_\_text = text

*self*.\_\_location = location

*self*.\_\_size = size

*self*.rectangle = pygame.rect.Rect(*self*.\_\_location, *self*.\_\_size)

*self*.\_\_backColourDormant = backColourDormant

*self*.\_\_backColourActive = backColourActive

*self*.\_\_backColour = *self*.\_\_backColourDormant

*self*.\_\_textColour = textColour

        fontSize = 999

        font = pygame.font.SysFont("Courier New", int(dispHeight \* fontSize/1080))

        fontRenderSize = font.size(*self*.\_\_text)

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

        while fontRenderSize[0] > *self*.\_\_size[0] or fontRenderSize[1] > *self*.\_\_size[1]:

            fontSize -= 1

            font = pygame.font.SysFont("Courier New", int(dispHeight \* fontSize/1080))

            fontRenderSize = font.size(*self*.\_\_text)

*self*.\_\_font = font

## Button.SetLocation()

This method is used to move the rectangle that will act as collision for the box and the text of the box.

*#Sets center to the new position*

    def SetLocation(self, pos):

*self*.rectangle.center = pos

*self*.\_\_location = (*self*.rectangle.x, *self*.rectangle.y)

## Button.SetText()

    def SetText(self, text):

*self*.\_\_text = text

## Button.Render()

This does the drawing of the button onto the screen.

    def Render(self, window):

        pygame.draw.rect(window, *self*.\_\_backColour, *self*.rectangle)

        textRenderSize = *self*.\_\_font.size(*self*.\_\_text)

*#location of text = top right of box + half of the difference of width, top right of box + half of the difference of height (between the box and text)*

        textRenderLocation = (*self*.\_\_location[0] + (*self*.\_\_size[0] - textRenderSize[0]) / 2, *self*.\_\_location[1] + (*self*.\_\_size[1] - textRenderSize[1]) / 2)

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

        window.blit(textRender, textRenderLocation)

## Button.SetActive() and Button.SetDormant()

These change the colour of the textbox.

    def SetActive(self):

*self*.\_\_backColour = *self*.\_\_backColourActive

    def SetDormant(self):

*self*.\_\_backColour = *self*.\_\_backColourDormant

## Button.CheckIfHovering()

This will check if the mouse is on top of the button.

    def CheckIfHovering(self, mousePos):

        if *self*.rectangle.collidepoint(mousePos):

            return True

        else:

            return False

## Button.GetPressedState()

Returns whether or not the button is pressed down

    def GetPressedState(self):

        return *self*.\_\_pressed

## Button.Pressed() and Button.DePressed()

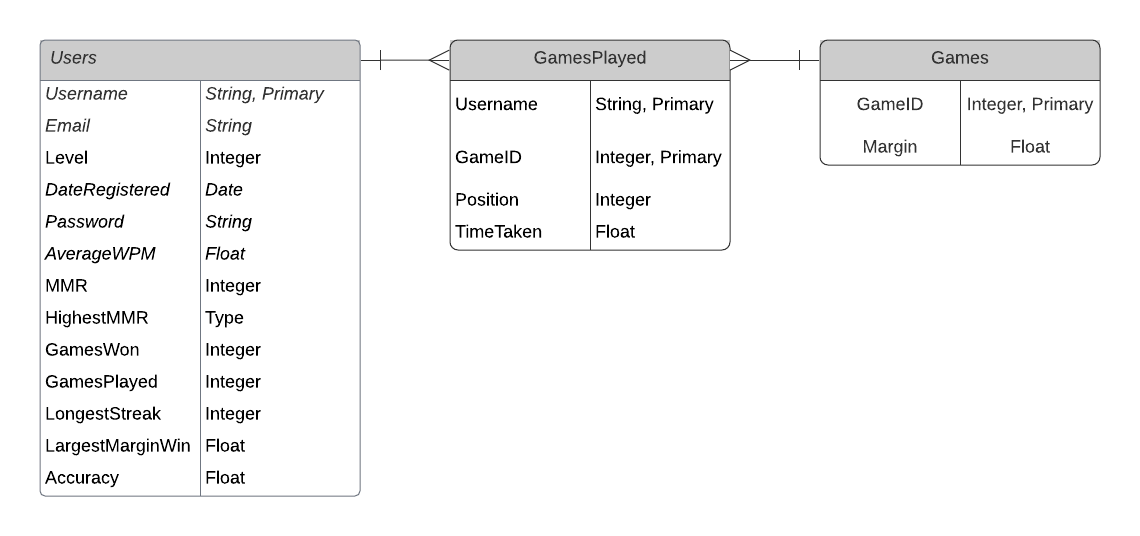
    def Pressed(self):

*self*.\_\_pressed = True

    def DePressed(self):

*self*.\_\_pressed = False

# Database

The database is not a focus of the program as it is not very complex, but it is required to store the user’s information. This ER diagram is used however I have left the Email section out for now because it is for a feature that’s not necessary right now (resetting passwords). The database will be handled by an object called a DatabaseHandler. This object needs to be able to connect to the database and make modifications to it.

This is done by using the sqlite3 built in module that comes with python. The object needs to make new players, get details for players and to return an updated Player object.

import sqlite3

*#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 = sqlite3.connect("db.db")

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

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

        command = """CREATE TABLE IF NOT EXISTS

        Users(Username TEXT PRIMARY KEY, Level INTEGER, Password TEXT, AverageWPM FLOAT, Elo INTEGER, HighestElo INTEGER, GamesWon INTEGER, GamesPlayed INTEGER, LongestStreak INTEGER, LargestWinMargin FLOAT, Accuracy FLOAT)"""

*self*.\_\_cursor.execute(command)

        command = """CREATE TABLE IF NOT EXISTS

        GamesPlayed(Username TEXT, GameID INTEGER, Position INTEGER, TimeTaken FLOAT, PRIMARY KEY(Username, GameID))"""

*self*.\_\_cursor.execute(command)

        command = """CREATE TABLE IF NOT EXISTS

        Games(GameID INTEGER PRIMARY KEY, Margin FLOAT)"""

*self*.\_\_cursor.execute(command)

## Database.CreateNewUser()

This method will take a player object and create a new user based on it. By default, a player will start at an Elo rating of 1000.

    def CreateNewUser(self, player):

        username = player.username

        password = player.password

        level = player.level

        avgWPM = player.avgWPM

        Elo = player.Elo

        highestElo = player.highestElo

        gamesWon = player.gamesWon

        gamesPlayed = player.gamesPlayed

        longestStreak = player.longestStreak

        largestWinMargin = player.largestWinMargin

        accuracy = player.accuracy

        params = (username, level, password, avgWPM, Elo, highestElo, gamesWon, gamesPlayed, longestStreak, largestWinMargin, accuracy)

        command = f"""INSERT INTO Users

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

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

As you can see the SQL query is

INSERT INTO Users

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

And the parameters used are the player object’s initial attributes.

## Database.LoadUser()

This method is going to select all columns in a row where the player’s username matches the one passed as an argument.

*#Takes player object with username already defined and returns player object with values from database*

    def LoadUser(self, player):

        params = (player.username,)

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

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

        valuesReturned = *self*.\_\_cursor.fetchall()   *#Gives a list of tuples with the correct values in them*

        valuesReturned = valuesReturned[0]

        player.username = valuesReturned[0]

        player.password = valuesReturned[2]

        player.level = valuesReturned[1]

        player.avgWPM = valuesReturned[3]

        player.Elo = valuesReturned[4]

        player.highestElo = valuesReturned[5]

        player.gamesWon = valuesReturned[6]

        player.gamesPlayed = valuesReturned[7]

        player.longestStreak = valuesReturned[8]

        player.largestWinMargin = valuesReturned[9]

        player.accuracy = valuesReturned[10]

        return player

The valuesReturned variable is a list of tuples. In this case it will only ever return 1 tuple as there will only be 1 player. This means we can simple set valuesReturned to the first item in the list.

The values in the tuple are then assigned to the attributes of the Player object and then the Player object is returned.

## Database.Close()

This method not only closes the database but also commits any changes. Without this any changes made to the database since opening will be lost.

    def Close(self):

*self*.\_\_connection.commit()

*self*.\_\_connection.close()