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 there be a level system to show how much a player has played by giving experience when certain criteria is met?

O: This would be a good idea but it would have to be mainly about playtime, as other things might cause players to play differently in order to maximise it.

A: Should there be a reward for levelling up the account? Should there be a max level or a prestige system?

O: Maybe just a badge next to their name on the GUI that the opponent can see.

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: They should get less experience points for levelling up their account and should also be considered when adjusting the ELo for whoever won or lost.

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

O: The only thing that should decide who won is the person who wrote out the entire phrase first.

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 players lose ranked points after not playing for a set period of time?

O: After 10 days of not playing any matches they should lose a certain number of points per day, but maybe this should only apply to players above a certain ELo to prevent people using this to only play against slow typers.

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: Should there be a mode to play against certain players by searching for their names?

O: Yes

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 in plaintext, but this will be told to the user when creating their account, so they use something different to their usual password.
   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. The program should reduce the players ELo if it is both above a certain threshold and should reduce it by an amount linear to how many days they haven’t played after 10 days.
   1. This will be calculated by the server at the start of every day.
4. 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.
5. The game should last 30 seconds
6. If a player leaves, then they should lose maximum ELo and that should be given to the player who did not leave.
7. 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. A short sound should play when the opponent is at 25, 50, 75 and 100% of the way done.
   3. The opponent’s text background should change colours to reflect if they are ahead or behind.
   4. Sound volume should be able to be changed in the settings.
8. 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.
9. 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. 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. 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 the focus.

# 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 in particular 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 fixed by opening and closing the connection whenever a new request is going to be made; this however does not work when both players are in game, as it is necessary to keep the connection open due to the high frequency of inputs from the users. This is fixed by closing the connection whenever the game is finished, or early if the player presses alt + f4 or the cross in the top right. This is handled by the pygame.event.QUIT event which is triggered by the aforementioned actions. This would allow us to close the connection manually when the player quits the game.

One approach to socket programming is making new threads for each client, though I found this to be problematic as the data from the threads would be needed in the main program for things such as the matchmaking, so I have taken this approach for the client, because the client simply needs to display the information it receives, but the server will be asynchronous by making use of timeouts.

The use of timeouts is crucial here as otherwise the program would be stuck waiting for both clients to connect and for clients to send a message. Timeouts will make it so that the server will check each socket for a message and the message will be empty if there is no message sent by the client.

When a socket times out, it will return a socket error. This error can be caught and used to determine if a message was received, if so then the program can accept the message by turning the socket into a blocking socket again. If no message was received the program can then do other things.

## Planning for server

Flowchart for server finding connections to clients:

Diagram

Description automatically generated

Flowchart for server receiving messages from client:

Diagram

Description automatically generated

Flowchart for server sending messages to the client:

Diagram

Description automatically generated

Flowchart for server handling the client:

Diagram

Description automatically generated

## 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

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

import socket

import threading

from tkinter.tix import Tree

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

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Input data | Expected output | Output |
| Client connects to server | Open server file, open multiple instances of client file. | The hello message to be shown on the server console 3 times, alongside the address of the clients. |  |
| 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. | Same as above. |
| 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. | The top is the client script and the bottom is the server script. I manually typed the messages, though this will not work for the final project. Until this gets integrated into the game, this will not be as useful as it will be. |
| 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. | Same as above. | 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 wont be using inputs to send messages to the server in the end product. |

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

# TextBox

USING pygame

#resolution of game window

dispWidth 🡨 display width

dispHeight 🡨 display height

boxColourActive 🡨 tuple with RGB values

boxColourDormant 🡨 tuple with RGB values

backColour 🡨 tuple with RGB values for the background colour

textColour 🡨 tuple with RGB values for the text colour

#keeps font size the same regardless of resolution.

fontSize 🡨 FLOAT\_TO\_INT(dispHeigh\*62/1080)

window 🡨 make a pygame window with size dispWidth, dispHeight

set the window’s caption to “test for game”

gameClock 🡨 make a new pygame clock object

#sets the box to be 3/5 of the screen long

boxWidth 🡨 FLOAT\_TO\_INT(dispWidth \* 3/5)

boxHeight 🡨 FLOAT\_TO\_INT(50 \* dispHeight / 1080)

#defines box position

boxX 🡨 FLOAT\_TO\_INT(dispWidth / 5)

boxY 🡨 FLOAT\_TO\_INT(6 \* dispHeight / 20)

#defines box coordinates

boxCoords 🡨 tuple with boxX and boxY

#sets the box to be unselected

boxColour 🡨 boxColourDormant

box 🡨 pygame rectangle object with argument boxCoords, boxWidth and boxHeight

text 🡨 ‘’

#defines an empty list for the removed text stack which will be used later to make sure that the box

#contains the letters and they do not keep going after the letters reach the halfway point of the box.

removedText 🡨 []

font 🡨 pygame font object with arguments None and fontSize

timeBetweenBackspaces 🡨 50

timeSinceLastBackspace 🡨 0

typing 🡨 False

GAMELOOP 🡨 True

WHILE GAMELOOP

#progresses time by 1 tick (1 frame)

gameClock.tick()

#sets fps

pygame.time.delay(30)

#pygame.event.get() returns a list of the events that have happened

FOR event IN pygame.event.get()

IF event.type = quit THEN

GAMELOOP = False

ELSE IF event.type = mousebuttondown THEN

IF mouse cursor is on the box THEN

boxColour 🡨 boxColourActive

typing 🡨 True

ELSE

boxColour 🡨 boxColourDormant

typing 🡨 False

ENDIF

ELSE IF event.type = keydown AND typing THEN

IF event.key = enter THEN

OUTPUT text

ELSE IF event.key = backspace THEN

text 🡨 text – letter at the end

deleting 🡨 True

timeSinceLastBackspace 🡨 0

ELSE

#concatenates the text string with the new letter that was pressed

text 🡨 text + event.unicode

ENDIF

ELSE IF event.type = keyup THEN

IF event.key = backspace THEN

deleting 🡨 False

IF LEN(removedText) > 0 THEN

text 🡨 removedText.POP() + text

ENDIF

ENDIF

ENDIF

ENDFOR

IF deleting AND timeSinceLastBackspace > timeBetweenBackSpaces AND typing THEN

text 🡨 text[:-1]

timeSinceLastBackspcace 🡨 0

#bring back a letter from the string

IF LEN(removedText) > 0 THEN

text 🡨 removedText.POP() + text

ENDIF

ENDIF  
 timeSinceLastBackspace 🡨 timeSinceLastBackspace + the time since last frame

textRender 🡨 render the text with the font

fill window with backColour

draw text rectangle

“blit” the rectangle onto the screen

#draws it on top of the things already there

update the display

ENDWHILE

## Actual code for textbox with typing functionality

The following code has been altered since the pseudocode, it is now object oriented.

from typing import Text

import pygame

class TextBox:

    def \_\_init\_\_(self, boxWidth, boxHeight, boxCoords, boxColourActive, boxColourDormant, textColour, font, previewTextColour, textColourWrong = (196, 24, 24)):

*self*.\_\_boxSize = (boxWidth, boxHeight)  *#Box size (width, height)*

*self*.\_\_boxCoords = boxCoords    *#Box coordinates (x,y)*

*self*.\_\_boxColourActive = boxColourActive    *#Box colour when box is selected*

*self*.\_\_boxColourDormant = boxColourDormant  *#Box colour when box isn't selected*

*self*.isActive = False

*self*.boxColour = *self*.\_\_boxColourDormant

*self*.\_\_textColour = textColour  *#Text colour for correct letters*

*self*.\_\_previewTextColour = previewTextColour

*self*.\_\_textColourWrong = textColourWrong    *#Text colour for wrong letters*

*self*.\_\_text = ""    *#Empty string to add text to later*

*self*.\_\_previewText = ""

*self*.\_\_font = font

*self*.box = pygame.Rect(*self*.\_\_boxCoords, *self*.\_\_boxSize) *#Defines rectangle object (pygame)*

    def SetActive(self):

*self*.boxColour = *self*.\_\_boxColourActive

*self*.isActive = True

    def SetDormant(self):

*self*.boxColour = *self*.\_\_boxColourDormant

*self*.isActive = False

*#Sets the previewText to text*

    def SetPreviewText(self, text):

*self*.\_\_previewText = text

*self*.\_\_text = ""

*#Removes a letter from text and potentially brings back a letter that was previously taken off the screen*

    def DeleteLetter(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]

    def AddLetter(self, letter):

*self*.\_\_text += letter

*#Draws the textbox*

    def DrawBox(self, window):

*#Length of text is used a lot, this saves some function calls*

        lenText = len(*self*.\_\_text)

*#Cuts front of text so that the end is in the middle*

        lettersFromBack = 0

        while *self*.\_\_font.size(*self*.\_\_text[-(lettersFromBack + 1):])[0] <= *self*.\_\_boxSize[0] / 2 and lettersFromBack + 1 <= lenText:

            lettersFromBack += 1

*#Text that is cut to reach the middle of the box*

        cutText = *self*.\_\_text[-lettersFromBack:]

        lettersFromBack2 = lettersFromBack

*#If the preview text isnt empty (Game has started) then it will cut it so that it fits in the textbox*

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

            while *self*.\_\_font.size(*self*.\_\_previewText[lenText - lettersFromBack: lettersFromBack2 + 1])[0] <= *self*.\_\_boxSize[0] - 10:

                lettersFromBack2 += 1

*#Cuts the preview text to fit the whole box*

            cutPreviewText = *self*.\_\_previewText[lenText - lettersFromBack: lettersFromBack2]

*#Cuts the preview text so that it is the same length as cutText*

            cutCorrectText = *self*.\_\_previewText[lenText - lettersFromBack:lenText]

*#Replaces correct letters player types with spaces*

            cutText = list(cutText)

            cutCorrectText = list(cutCorrectText)

            for i in range(len(cutText)):

                if cutText[i] == cutPreviewText[i]:

                    cutText[i] = " "

                else:

                    cutText[i] = cutPreviewText[i]

                    cutCorrectText[i] = " "

            cutText = "".join(cutText)

            cutCorrectText = "".join(cutCorrectText)

*#Makes render for both*

            previewTextRender = *self*.\_\_font.render(cutPreviewText, True, *self*.\_\_previewTextColour)

            cutCorrectTextRender = *self*.\_\_font.render(cutCorrectText, True, *self*.\_\_textColour)

*#Renders text that is wrong*

            textRender = *self*.\_\_font.render(cutText, True, *self*.\_\_textColourWrong)

            textCoords = (*self*.\_\_boxCoords[0] + 5, *self*.\_\_boxCoords[1] + 5)

*#Blits the gray text, then the orange text and then the red text*

            window.blit(previewTextRender, textCoords)

            window.blit(cutCorrectTextRender, textCoords)

            window.blit(textRender, textCoords)

        elif *self*.\_\_previewText == "":

            textRender = *self*.\_\_font.render(cutText, True, *self*.\_\_previewTextColour)

            window.blit(textRender, (*self*.\_\_boxCoords[0] + 5, *self*.\_\_boxCoords[1] + 5))

Some classes are not present here, they will be covered in the Client side section (not yet on this document).

## Testing for text box

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | Textbox is drawn correctly | I will run the program | A textbox with 3/5 the width of the resolution and 5/108 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. | Normal | Pass |
| 3 | Letters can be typed into the textbox | I will type “This is a test” into the textbox | The text entered will be displayed in the textbox | Normal | Pass |
| 4 | 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 |
| 5 | 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 |
| 6 | 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 |
| 7 | 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 |
| 8 | 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 |
| 9 | 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 |
| 10 | 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 |
| 11 | The game will close when Alt + f4 are pressed | I will press alt + f4 when the game has started | The game should close | Normal | Pass |
| 12 | The window can be resized by changing variables in the program | I will change the variable in charge of resolution | The game window should be a different size | Normal | Pass |
| 13 | The box should scale with the size of the window | I will change the variable in charge of resolution | The textbox should be a different size | Normal | Pass |
| 14 | Font size will change based on the size of the window | I will change the variable in charge of resolution | The text should be a different colour | Normal |  |

## Test 1

The text box will need to be drawn on the screen when the program is started. We can test this by running the program. The text box needs to be the correct shape and size relative to the screen, this was done in the program by making it a portion of the width of the screen and 50/1080th of the height.

With the resolution set to 1920x1080, the result is:

A picture containing rectangle

Description automatically generated

The textbox is draw as intended so this test is passed.

## Test 2

The textbox should change colour to indicate that the textbox is selected. The textbox being selected indicates that any key the user presses will be typed in the textbox. To test this, I have clicked on the textbox and taken a screenshot before and after with unnecessary space cropped out.

Before:



After:



Once anywhere outside the textbox is clicked then the textbox should become unselected and return to the original colour.

After clicking outside the box:



The colour difference is minimal, but it is noticeable when you click on it.

The textbox changes colour according to whether or not the textbox is selected, so this test is passed.

## Test 3

The textbox must be able to be typed in, so I will type “This is a test” into the textbox and paste a screenshot here:



This shows that the letters can be typed into the textbox so this test is passed.

## Test 4

The textbox must start removing text from the start once the written text reaches the halfway point, this is to keep the preview text (in the future) visible.

To test this, I will type things until it reaches the halfway point and paste a screenshot, I will then type another word and paste a screenshot.

At halfway point:



After another word:

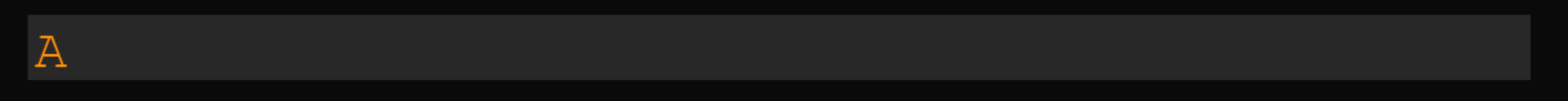


The text at the start was deleted and the typed letters are remaining in the middle of the box, so this test is passed.

## Test 5

The backspace key needs to delete text when pressed to allow for the user to fix their mistakes, to -test this I will type a letter, paste a screenshot, press backspace and paste another screenshot.

Letter A is pressed while holding shift:



Backspace is pressed:



The letter is deleted so this test is passed.

## Test 6

The backspace key should not crash the application if there is no text to delete. I will test this by opening the application and pressing backspace after selecting the textbox without typing anything.

Screenshot of before backspace is pressed:



Screenshot of after backspace is pressed:



The program didn’t crash so this test is passed.

## Test 7

The backspace key should be able to be held down to delete letters continuously. I will test this by writing a lot of letters, paste a screenshot, holding down the backspace key for 2 seconds and pasting another screenshot.

Before backspace:



After holding backspace for 2 seconds:

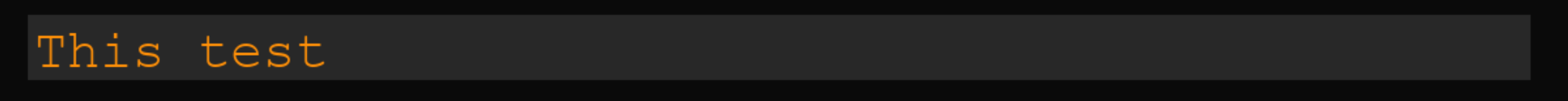


The backspace key being held down continues to delete letters at a fixed rate, so this test is passed.

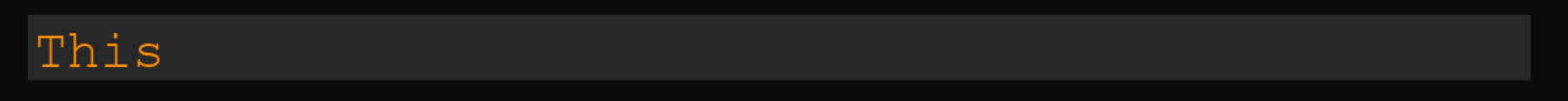
## Test 8

The backspace key should be able to delete entire words as long as the control key is held down, this is to allow easy deletion of text. To test this, I will type 2 words, paste a screenshot, press ctrl+backspace and paste another screenshot.

Before deletion:



After deletion:



The entire word was deleted so this test is passed.

## Test 9

The backspace key should be able to be held down in combination with the control key to continuously delete entire words. I will type a lot of words, paste a screenshot, hold ctrl+backspace for 2 seconds and paste another screenshot.

Before holding ctrl+backspace for 2 seconds:



After holding ctrl+backspace for 2 seconds:



The words continue to be deleted if both keys are held down, so this test is passed.

## Test 10

The text removed after reaching the halfway point should return when letters are deleted by the user. I will test this by typing a lot of words so that the halfway point is surpassed, pasting a screenshot, deleting some letters and pasting another screenshot.

Before deleting letters:



After deleting letters:



The letters previously removed are now returned so this test is passed.

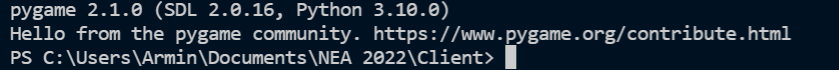
## Test 11

The program should close when the appropriate input is made. To test this I will press alt + f4 to close the application.

Before pressing alt + f4:



After pressing alt + f4:

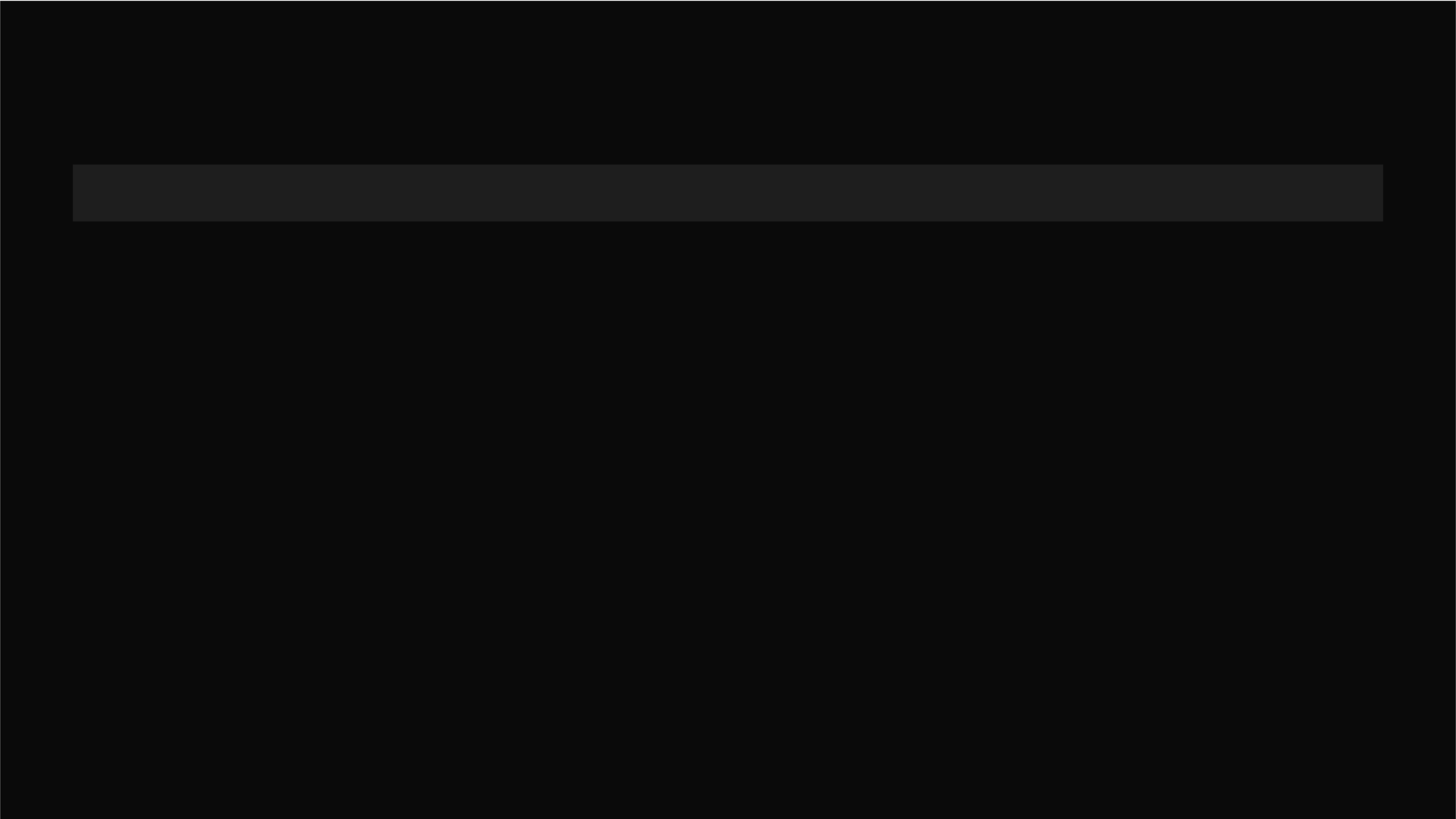


The application closes so the test is successful.

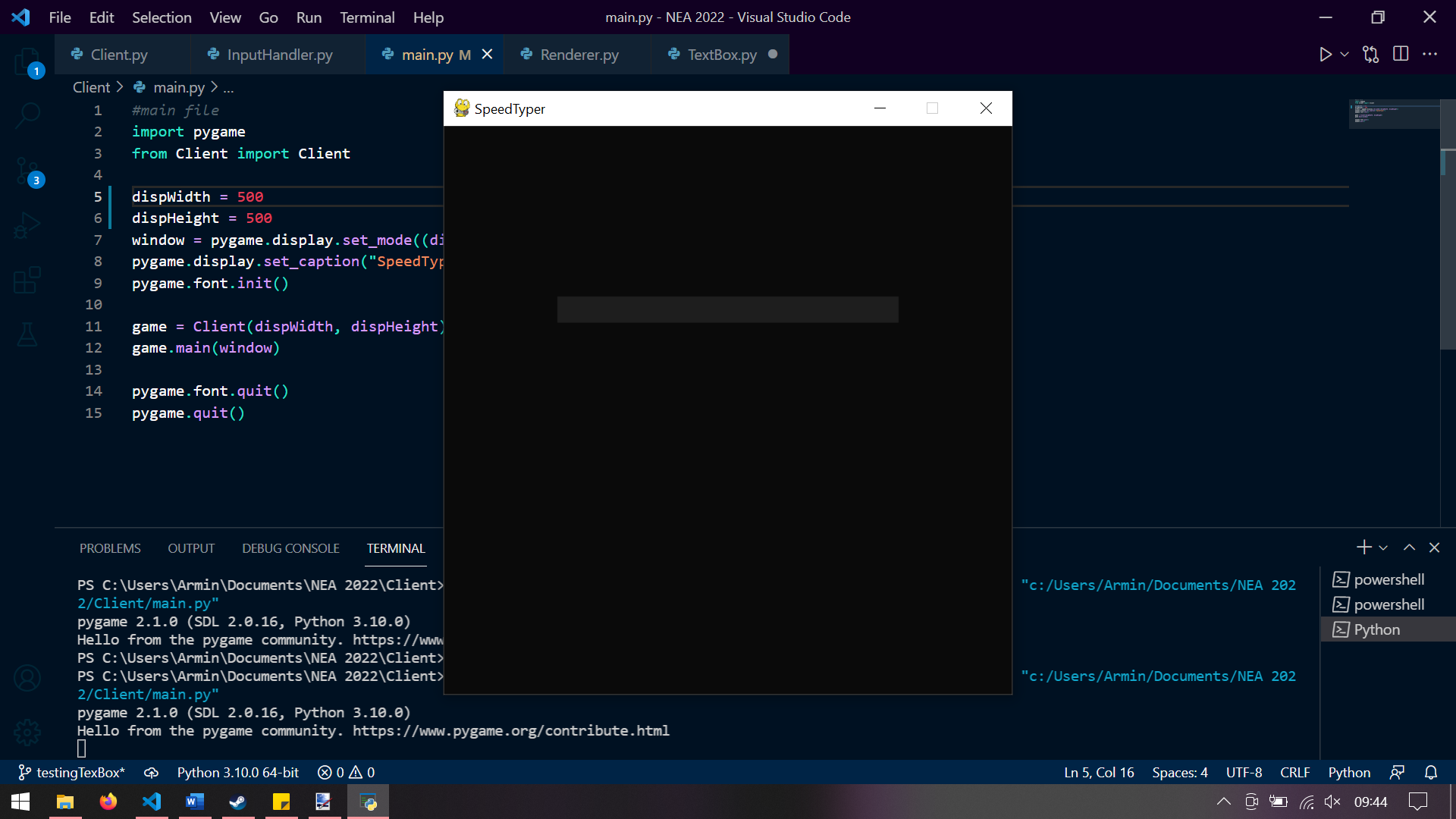
## Test 12

The window can be resized by changing the values in the main file for the client, this will be done automatically in the future by detecting the user’s screen resolution, however for now I will open the game, paste a screenshot, close the game and change the resolution from 1920x1080 to 500x500 and paste another screenshot.

1920x1080:



500x500:



The window size is changed so this test is passed.

## Test 13

Refer to test 12, this is shown in that test to be working as intended.   
This test is passed.

## Test 14

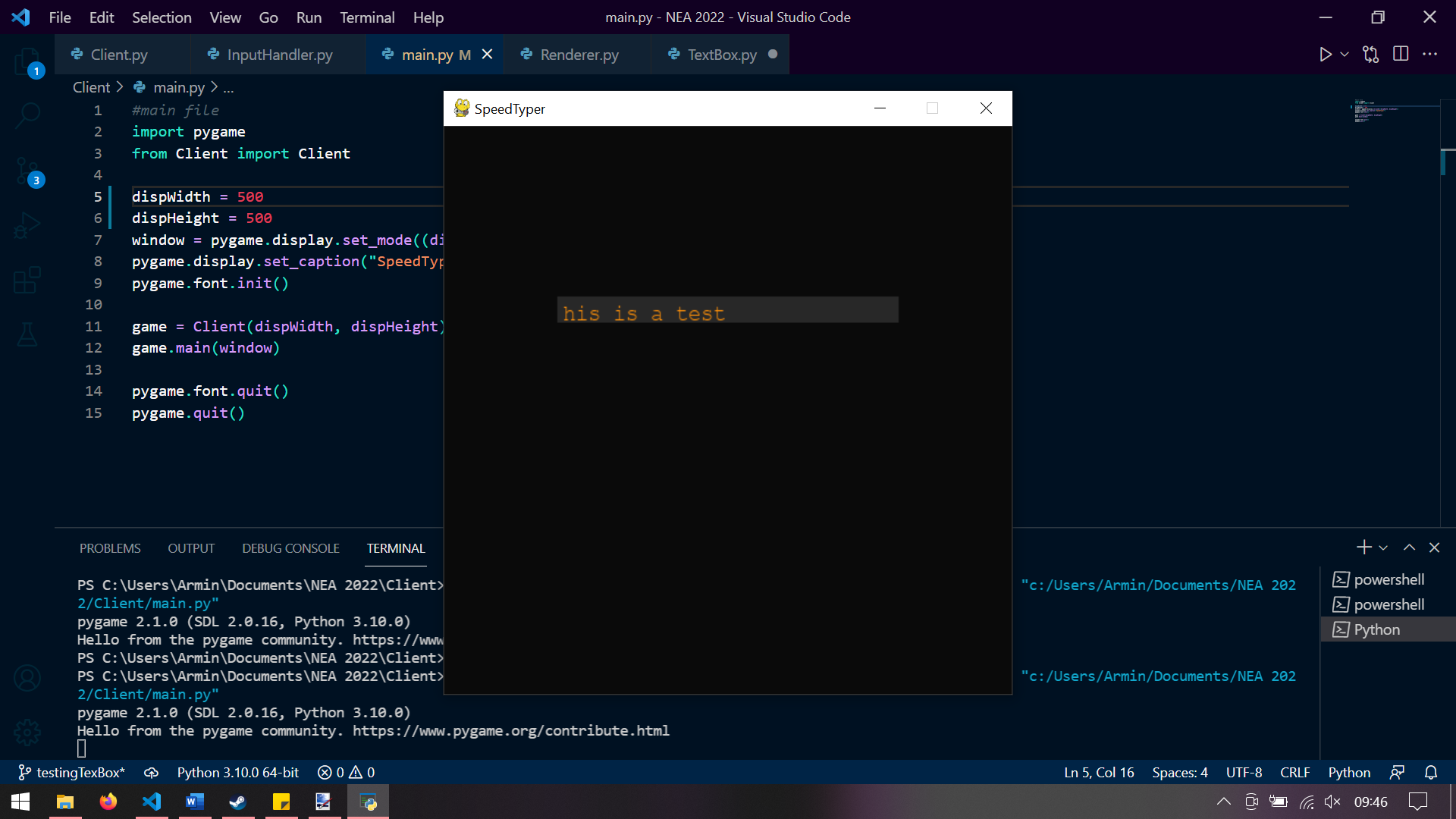
The font size should change to be within the textbox. I will type “This is a test” at 1920x1080 resolution, paste a screenshot, do the same at 500x500 and paste a screenshot.

1920x1080:

Text

Description automatically generated

500x500:



The font size changes to match the size of the textbox so this test is passed.

# 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()

Diagram

Description automatically generated

## WordGenerator.\_\_MakeWordsCorrectLength()

Diagram

Description automatically generated

## WordGenerator.GetWordsForProgram()

This method simply calls the other two with the correct parameters and returns the value.

## Python code for word generation

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*

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

        newWords = []

        while len(words) < length:

            newWords += words

            length -= len(words)

        newWords += words[:length]

        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)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1\* | Returns words | I will run the class’s file | It will return words | 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 |

\* Some tests were not shown due to them being too similar to those around them, and so being proven by those.

## Test 2

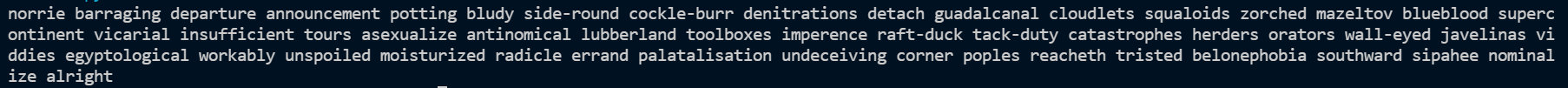
The program needs to return words with the correct length to fulfil the criteria that the length of the text presented to the player must be 30 seconds of their average WPM. In this test the number 50 was chosen as a word count and the text generator was run.

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

generator = WordGenerator()

print(generator.GetWordsForProgram(50))

The output was as follows:



When counted this was 50 words, and so the test passed. This also passes test 1.

## 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 a main game loop and things will be executed depending on what is happening in the program. The client is made up of many classes, a UML diagram for it is shown below:

Diagram

Description automatically generated

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.

## 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. Diagram

Description automatically generated

## Python code for handling input:

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

import pygame

class InputHandler:

    def \_\_init\_\_(self):

*self*.typing = False

*#Gets input, converts it to commands and returns a list of commands*

    def HandleInput(self, box):

        commands = []

        clicked = False

        for event in pygame.event.get():

*#Alt + f4*

            if event.type == pygame.QUIT:

                commands.append("QUIT")

*#On click changes if textbox is selected or not*

            elif event.type == pygame.MOUSEBUTTONDOWN:

                if box.collidepoint(pygame.mouse.get\_pos()):

*self*.typing = True

                    clicked = True

                else:

*self*.typing = False

                    clicked = True

*#Checks keypresses*

            elif event.type == pygame.KEYDOWN and *self*.typing:

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

                    commands.append("BACKSPACE DOWN")

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

                    pass

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

                    commands.append("CONTROL DOWN")

*#Adds letter pressed down as a commands*

                else:

                    commands.append(f"K{event.unicode}")

*#Detects key being depressed*

            elif event.type == pygame.KEYUP:

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

                    commands.append("BACKSPACE UP")

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

                    commands.append("CONTROL UP")

*#Determines if box should be selected or not*

        if *self*.typing and clicked:

            commands.append("CLICKED ON BOX")

        elif not *self*.typing and clicked:

            commands.append("CLICKED OUT OF BOX")

        return commands

## 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="")

## Test 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.

## Test 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.

## Test 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.

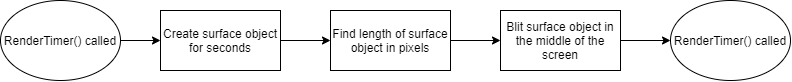
Text

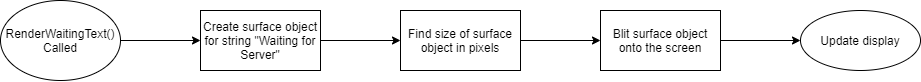
Description automatically generated

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

## Renderer

This object needs to draw the relevant things on screen. 





## Python code for Renderer:

import pygame

class Renderer:

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

*self*.\_\_backColour = (10,10,10)

*self*.\_\_font = pygame.font.SysFont("Courier New", int(dispHeight \* 60/1080))

*#Renders a background and a textbox*

    def Render(self, window, textBox):

        window.fill(*self*.\_\_backColour)

        pygame.draw.rect(window, textBox.boxColour, textBox.box)

        textBox.DrawBox(window)

*#Used in part where client is waiting for the game to start*

*#Renders a number*

    def RenderTimer(self, window, screenDimensions, seconds):

        numRender = *self*.\_\_font.render(str(seconds), True, (255,255,255))

        renderSize = *self*.\_\_font.size(str(seconds))

        window.blit(numRender, ((screenDimensions[0] - renderSize[0]) / 2, (screenDimensions[1] - renderSize[1]) / 2))

    def RenderWaitingText(self, window, screenDimensions):

        textRender = *self*.\_\_font.render("Waiting for Server", True, (255,255,255))

        renderSize = *self*.\_\_font.size(str("Waiting for Server"))

        window.blit(textRender, ((screenDimensions[0] - renderSize[0]) / 2, (screenDimensions[1] - renderSize[1]) / 2))

## Testing for Renderer:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | Purpose | Test | Expected result | Type | Result |
| 1 | Draws the background with the appropriate colour | I will run the program | The background should be the colour with RGB values 10,10,10 | Normal | Pass |
| 2 | Draws the textbox | I will run the program | The textbox should be drawn | Normal | Pass |
| 3 | Updates the screen after something changes | I will click on the textbox so that it changes colour | The textbox should change colour | Normal | Pass |

## Test 1

The renderer object needs to draw the background with the colour defined during initialisation. In this case it needs to be the colour with RGB values 10,10,10 (almost black).

To test this I will run the program and paste a screenshot of the window here.

Text

Description automatically generated

The background colour is the correct colour and so this test is passed.

## Test 2

The renderer object needs to draw the textbox. To test this I will run the program and paste a screenshot here.

Text

Description automatically generated

The textbox is drawn correctly and so this test is passed.

## Test 3

The screen needs to update after something happens. To test this, I will click on the textbox, causing the textbox colour to change. I will paste before and after pictures here:

Before:

Text

Description automatically generated

After:

Text

Description automatically generated

The background colour of the textbox is slightly lighter in the second picture because it is selected. This test is passed.

## TextBox

The textbox object is the same as shown in the previous section. This object was used in the client side of the program.

## ClientSocket

The ClientSocket object is being used in this section aswell, though it was covered in the connecting client and server section of this document.

I have however, added an EndConnection method to this object in order to allow the connection to be ended from outside the object.

  def EndConnection(self):

*self*.connected = False

*self*.msgsToSend.append("!DISCONNECT")

# Input Box

This class needs to be able to make a box, render it on screen and show the text.

import pygame

class InputBox:

    def \_\_init\_\_(self, textColour, colourActive, colourDormant, hashedOut, rectangle, dispHeight):

*self*.text = ""

*self*.\_\_textColour = textColour

*self*.\_\_colour = colourDormant

*self*.isActive = False

*self*.\_\_colourActive = colourActive

*self*.\_\_colourDormant = colourDormant

*self*.\_\_hashedOut = hashedOut

*self*.\_\_size = rectangle.size

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

*self*.rectangle = rectangle

        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[1] > *self*.\_\_size[1]:

            fontSize -= 1

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

            fontRenderSize = font.size(*self*.text)

*self*.\_\_font = font

## InputBox.DrawBox()

This is a method that will draw the box on the screen. It will draw the box, the text and cut the text to fit.

    def DrawBox(self, window):

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

        if *self*.\_\_hashedOut:

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

        else:

            text = *self*.text

        lettersFromBack = len(text)

*#While the rendered version of text with lettersFromBack number of letters from the back is bigger than the textbox*

        while *self*.\_\_font.size(text[-lettersFromBack:])[0] > *self*.\_\_size[0] - 10:

            lettersFromBack -= 1

        textRender = *self*.\_\_font.render(text[-lettersFromBack:], True, *self*.\_\_textColour)

        window.blit(textRender, (*self*.\_\_location[0] + 5, *self*.\_\_location[1] + 5))

## InputBox.CheckIfClicked()

This will take a tuple of (x,y) and return whether or not the location is in the box. This will be used when the MOUSEBUTTONDOWN event happens.

    def CheckIfClicked(self, location):

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

            return True

        else:

            return False

## InputBox.AddLetter()

This method will simply add a letter to the \_\_text attribute.

    def AddLetter(self, letter):

*self*.text += letter

## InputBox.RemoveLetter()

This method will remove a letter or word depending on if the control key is being held down. This is done by removing any trailing spaces and then deleting letters until a space is found or the text has run out.

    def RemoveLetter(self, control):

        if *self*.text != "":

            if control:

                if *self*.\_\_hashedOut:

*self*.text = ""

                else:

*#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]

## InputBox.SetActive()

Sets \_\_isActive to True and changes the colour of the box.

    def SetActive(self):

*self*.\_\_colour = *self*.\_\_colourActive

*self*.isActive = True

## InputBox.SetDormant()

Sets \_\_isActive to False and changes the colour of the box.

    def SetDormant(self):

*self*.\_\_colour = *self*.\_\_colourDormant

*self*.isActive = False

# 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.

import pygame

*#This object is a button that can be clicked to do certain actions*

class Button:

    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

This sets some attributes and then calculates the max font size that will fit in the box.

## Button.SetFont()

A setter for the \_\_font attribute

    def SetFont(self, font):

*self*.\_\_font = font

## Button.SetLocation()

This method will set the location of the centre of the button to the pos argument.

*#Sets center to the new position*

    def SetLocation(self, pos):

*self*.rectangle.center = pos

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

## Button.SetText()

This method will set the \_\_text attribute to something else.

    def SetText(self, text):

*self*.\_\_text = text

## Button.GetText()

A getter for the \_\_text attribute.

    def GetText(self):

        return *self*.\_\_text

## Button.Render()

Draws everything on 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()

Sets the button to be active, this changes the colour of the button.

    def SetActive(self):

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

## Button.SetDormant()

Sets the button to be inactive, this changes the colour of the button.

    def SetDormant(self):

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

## Button.CheckIfHovering()

Returns whether or not the mousePos argument (x,y) is in the button’s area.

    def CheckIfHovering(self, mousePos):

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

            return True

        else:

            return False

## Button.GetPressedState

A getter for the \_\_pressed attribute.

    def GetPressedState(self):

        return *self*.\_\_pressed

## Button.Pressed()

A setter for the \_\_pressed attribute.

    def Pressed(self):

*self*.\_\_pressed = True

## Button.Depressed()

A setter for the \_\_pressed attribute.

    def DePressed(self):

*self*.\_\_pressed = False

# 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)

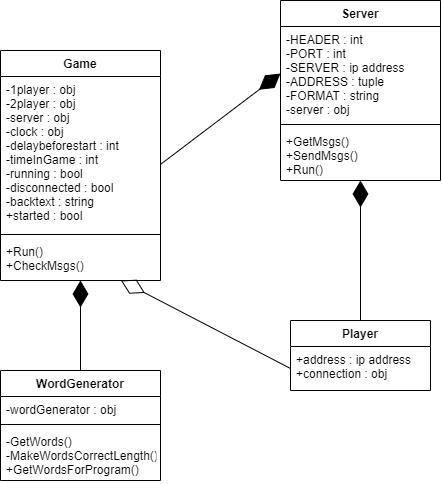
pygame.font.quit()

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

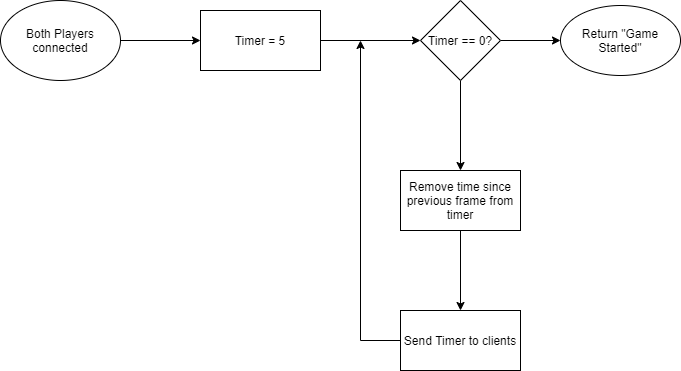
*self*.\_\_disconnected = ""

*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()

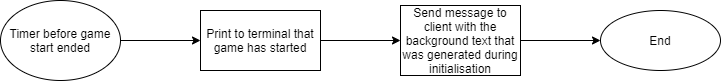
*#If the time has run out*

        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()

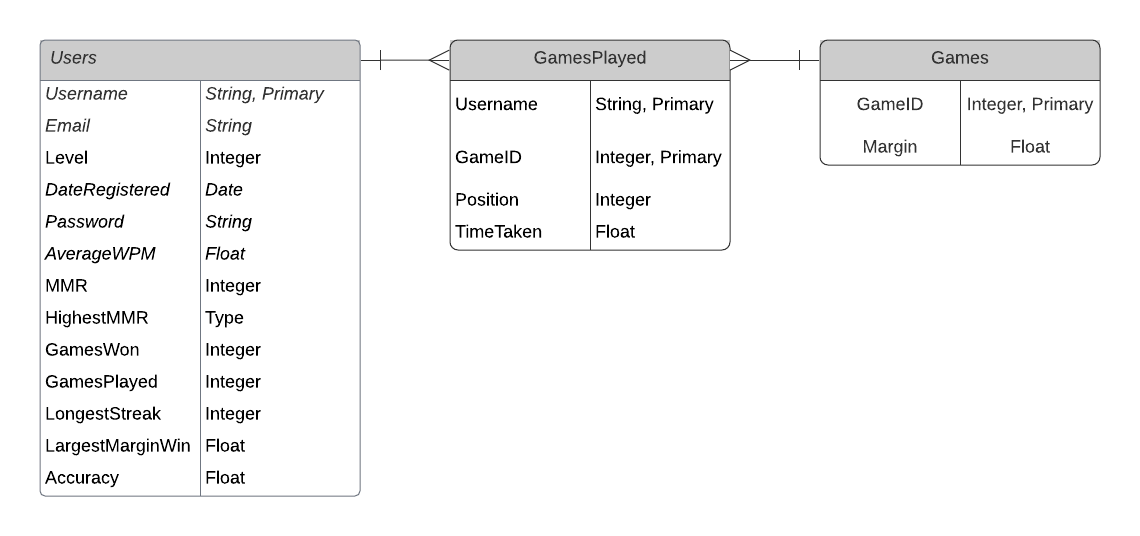
    def Pressed(self):

*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()