# 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 keep doing this until they type the correct one and continue from there.

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, but it needs to be a normal sentence that is not gibberish. 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. Text size.
      3. Text colour (For colour-blind players).
   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 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.
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 let’s 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.

# Entity relationship diagram

# H:\Downloads\Database ER diagram (crow's foot).png

# 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 issue that arises with using sockets in a scenario where there isn’t always information that will be sent is that sockets by default are set to “block” as in the program stops until a message is received. This is bypassed by making a non-blocking socket which doesn’t do this, which instead returns an error if there is no message to be returned.

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:

## Planning for client

Flowchart for client connecting to the server:



Flowchart for client sending a message to the server:

Diagram

Description automatically generated

Flowchart for client getting a message from the server:

Diagram

Description automatically generated

## Python code for server

import socket

import threading

HEADER = 8

PORT = 5000

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

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

FORMAT = 'utf-8'

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

server.bind(ADDRESS)

def GetMsgs(conn,addr):

    #receiving messages

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

    if msgLen:  #First message will always be empty

        msgLen = int(msgLen)

        msg = conn.recv(msgLen).decode(FORMAT) #Waits for a message with length msgLen to be received

        if msg.upper() == "!DISCONNECT":

            conn.close()

        if msg != "":

            print(str(addr) +  ":", str(msg)) #Prints the message

def SendMsg(newMsg, conn):

    encMessage = newMsg.encode(FORMAT) #encodes msg with utf-8

    msgLen = len(encMessage)

    msgLen = str(msgLen).encode(FORMAT)

    msgLen += b' ' \* (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)

    newMsg = ""

    return newMsg

def HandleClient(conn, addr):

    print(f"New Client: {addr}")  #Outputs the new client's local address

    newMsg = "Connection established"

    #create new thread for getting messages

    while True:

        GetMsgs(conn, addr)

        #sending messages

        if newMsg:

            newMsg = SendMsg(newMsg, conn)

        #put stuff here for server to do to change newMsg

def GetClient():

    server.listen() #Looks for connections

    threads = []

    while True:

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

        thread = threading.Thread(target=HandleClient, args=(conn, addr)) #Makes a new thread for the client handling so when it listens it doesn't stop the whole program

        thread.start()  #Starts the thread

        threads.append(thread)

#server starts here

print("Starting server")

GetClient()

## Python code for client

import socket

import threading

class Client:

    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

    def SendMsg(self, msg):

        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

        self.\_\_client.send(msgLen)

        self.\_\_client.send(encMessage)

    def GetMsgs(self):

        while True:

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

            if msgLen:  #First message will always be empty

                msgLen = int(msgLen)

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

                if msg != "":

                    print(f"\nMessage from server: {str(msg)}")

client = Client()

receiveThread = threading.Thread(target=client.GetMsgs)

receiveThread.start()

newMsg = "hello"

while True:

    client.SendMsg(newMsg)

    newMsg = input("Client message to server:")

## 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.

# Pygame code for getting text input

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.

import pygame

from TextBox import TextBox

from InputHandler import InputHandler

from Renderer import Renderer

from WordGeneration import WordGenerator

class Game:

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

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

        self.\_\_inputHandler = InputHandler()    #Creates an InputHandler 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()            #Creates Renderer object

        wordGen = WordGenerator()               #Creates WordGenerator object

        self.\_\_backText = wordGen.GetWordsForProgram(500)   #Generates 500 words

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

        self.\_\_textBox = TextBox(int(dispWidth - (dispWidth \* 2/5)), int(50 \* dispHeight / 1080), (int(dispWidth / 5), int(6 \* dispHeight / 20)), (40,40,40), (30,30,30), (255,144,8), int(dispHeight\*42/1080), self.\_\_backText, (160,160,160))

        self.\_\_GAMELOOP = True

    def main(self, window):

        while self.\_\_GAMELOOP:

            self.\_\_gameClock.tick()

            pygame.time.delay(30)                       #Determines max fps of game

            commands = self.\_\_inputHandler.HandleInput(self.\_\_textBox.box) #Gets list of input events

            self.TranslateInput(commands)   #Converts keyboard inputs into changes in attributes

            self.CheckForBackspace()        #Function for checking if backspace is held down

            self.\_\_timeSinceLastBackspace += self.\_\_gameClock.get\_time()    #Adds time since last frame to time since last backspace

            self.\_\_renderer.Render(window, self.\_\_textBox)  #Draws everything

        return 0

    def TranslateInput(self, commands):

        for command in commands:

            if command == "QUIT":                   #If alt + f4 pressed or quit button (in the future)

                self.\_\_GAMELOOP = False

            elif command[0] == "K":                 #K is always followed by another letter (letter that was pressed)

                command = command[1:]               #Removes K

                self.\_\_textBox.AddLetter(command)   #Adds letter to textbox

            elif command == "CLICKED ON BOX":

                self.\_\_textBox.SetActive()          #Changes colour and enables typing in the textbox

            elif command == "CLICKED OUT OF BOX":   #Changes colour and disables typing in the textbox

                self.\_\_textBox.SetDormant()

            elif command == "BACKSPACE DOWN":

                self.\_\_textBox.DeleteLetter(self.\_\_ctrl)       #Removes letter form textbox

                self.\_\_deleting = True              #True until BACKSPACE UP

                self.\_\_timeSinceLastBackspace = -200    #Gives 0.2 second delay until deleting starts

            elif command == "BACKSPACE UP":

                self.\_\_deleting = False

            elif command == "CONTROL DOWN":

                self.\_\_ctrl = True

            elif command == "CONTROL UP":

                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

## Testing for text box

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Input | Expected output | Output |
| Textbox is drawn correctly | None | A textbox with 3/5 the width of the resolution and 5/108 x the resolution height. |  |
| Textbox will change colour when clicked on | Click on the box. | The colour will change to the chosen one. |  |
| Typing letters into the textbox | This is a test. | “This is a test” will show up on the box in the chosen font and colour. |  |
| The letters in the front will start being deleted when the text is too long | This is a test to see how the letters are deleted and so this needs to be somewhat long enough so that it reaches at least halfway. | The letters should not go past halfway through the textbox. |  |
| The letters in the front will reappear after being deleted and the backspace key is pressed | This is a test to see how the letters are deleted and so this needs to be somewhat long enough so that it reaches at least halfway. | The letters should come back in the correct order they were before they were deleted. | This test was carried on from the one above it: |
| The backspace key will delete letters when pressed | The backspace key will be pressed and depressed. | The letters should disappear. | Test passed. |
| The backspace key being held down will continuously delete letters. | The backspace key will be held down. | The letters will keep being deleted at an appropriate rate. | Test passed. |
| The game will close | Alt + f4 and the X button in the top right of the window. | The window will close and the game will terminate. | Test passed. |
| The window can be resized by changing dispWidth and dispHeight | Changing the size from 500x500 to 1920x1080. | The game will be fullscreen. | Test passed. |
| The box will scale with how big the window is | Increasing the size of the window. | The box will be bigger. | Test passed. |
| Font size will change depending on the size of the window | Increasing the size of the window. | The font will be bigger. | Test 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.

Text

Description automatically generated with low confidence

\_\_GetSong() needs to make an API call to request a list of songs, and then select one at random to be passed to \_\_GetLyrics().

\_\_GetLyrics() needs to make an API call to request the lyrics of a song, and then pass the request back to the main subroutine, GetWordsForProgram().

\_\_CutLyrics will cut the lyrics down to the correct length while removing capitalisation and removing paragraphs.

\_\_GetSong() is very easy to implement as it is just an API call, a flowchart for it is as follows:

## \_\_GetSong():

Pseudo-Code:

payload 🡨 request message

payload 🡨 payload encoded with utf-8

response 🡨 make API call

response 🡨 response.json()

tracknames 🡨 []

append track names to tracknames

randomTrack 🡨 tracknames[RANDOM\_INT(0, LEN(tracknames))]

return trackname of randomTrack

## \_\_GetLyrics():

This is the same as the one above but without the code after the call is made

## \_\_CutLyrics():

This needs to format lyrics to be in the appropriate format. This includes length and capitalisation. In this there will not be any capitalisation, and the length will be chosen by an integer passed as an argument.

Pseudo-Code:

remove watermark at the end of the lyrics

replace “\n” with “ “

newLyrics 🡨 “”

x 🡨 0

spaces 🡨 0

WHILE length >= spaces

IF x = LEN(lyrics) THEN

x 🡨 0

ENDIF

newLyrics 🡨 newLyrics + lyrics[x]

IF lyrics[x] = “ “ THEN

spaces 🡨 spaces + 1

ENDIF

ENDWHILE

## Python code for word generation

import requests

import json

import random

class wordGenerator:

    def \_\_init\_\_(self):

        #private so API key cant be accessed outside this object

        self.\_\_API\_KEY = "031542d08a5d97e13560b759e3094416"

    #function to get a response object with 100 songs

    def \_\_GetSong(self):

        #private as it does not need to be accessed outside the object

        #string with the request for list of top 100 songs that have lyrics is made

        payload = f"chart.tracks.get?apikey={self.\_\_API\_KEY}&country=us&f\_has\_lyrics=1&explicit=0&page\_size=100"

        #encoded using UTF-8 (default for encode())

        str(payload).encode()

        #uses response library to make request to the correct domain

        response = requests.get("https://api.musixmatch.com/ws/1.1/" + payload)

        #converts list of songs to json format

        response = response.json()

        #gets a random track\_id

        tracks = response["message"]["body"]["track\_list"]

        randomTrack = tracks[random.randint(0, len(tracks))]["track"]["track\_id"]

        print(randomTrack)

        return randomTrack

    #function to get lyrics of a specific track

    def \_\_GetLyrics(self, trackID):

        #private as it does not need to be accessed outside the object

        #string with API request to retrieve lyrics for given trackID

        payload = f"track.lyrics.get?apikey={self.\_\_API\_KEY}&track\_id={trackID}"

        str(payload).encode()

        response = requests.get("https://api.musixmatch.com/ws/1.1/" + payload).json()

        response = response["message"]["body"]["lyrics"]["lyrics\_body"]

        return response

    #cuts lyrics down to certain length and removes newlines

    def \_\_CutLyrics(self, lyrics, length):

        #private as it does not need to be accessed outside the object

        #cuts out watermark at the end of string so that if the length required is longer than the actual lyrics it will loop

        lyrics = lyrics.split("...")[0]

        lyrics = lyrics.lower()

        lyrics = list(lyrics)

        for i in range(len(lyrics)):

            if lyrics[i] == '\n':

                lyrics[i] = " "

        newLyrics = ""

        #adds length number of words to a string

        x = 0

        spaces = 0

        #counts spaces

        while length >= spaces:

            #loops to the front of the string if lyrics runs out of words

            if x == len(lyrics):

                x = 0

            #appends word from lyrics to newLyrics

            newLyrics += lyrics[x]

            if lyrics[x] == " ":

                spaces += 1

            x += 1

        return newLyrics

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

    def GetWordsForProgram(self, numberOfWords):

        #public as it is effectively the main() of this class

        #returns string with numberOfWords words

        return self.\_\_CutLyrics(self.\_\_GetLyrics(self.\_\_GetSong()), numberOfWords)

wordGen = wordGenerator()

print(wordGen.GetWordsForProgram(50))

# TODO testing for this section