Moving Words Typing Game
(Wspeed Knock-off)

Background:

My inspiration for this game was from a project made by a renowned programmer and musician from Finland named Bisqwit from back in 2001 called Wspeed. This is such a fascinating game where there are a bunch of words on screen moving from left to right and you must type in each word to make them disappear before they go out the window on the right-hand side of the screen. https://bisqwit.iki.fi/wspeed/

According to Bisqwit this game can make you a better typer. I can't back that claim, but the game is very fun and I want to try and make it on python. Bisqwit originally made the game in C programming language which from what I heard is very difficult because it's a lower-level programming language. I wanted to try and make this game because I thought it would be easier since python has a bunch of built in functions to help me with the word database.

The great features of Wspeed that intrigued me was that it had multiple language choices, scoring system , information about your ten-minute relative speed (10mrs) which according to Bisqwit is a good comparison value how good you are in typing, it also gives a description of your typing skill which is very neat, and the leaderboard to store your best scores which is sorted.https://www.youtube.com/watch?v=jnkOcAkLkBU&t=169s

Game feature requirements:

There will be 2 modes, Classic and Arcade. The former is exactly like wspeed and the latter is my take.

Other than the requirements given by Mr. Jude, the following are the features that will be in the game once it is fully fledged:

Classic Mode:

Functional Requirements:

- Language selector
- Timer (increasing)
- Counter for missed words (Max 10)
- 10-minute relative speed
- Input box for the user to see what they type
- Pause button
- Scoring system and a local storage method which stores the highest score and quickest time
- Length should be more than 1

Non-Functional Requirements:

- Music which increases its tempo the more mistakes you made
- Description of typing skill
- Color change for words that are nearing the right-hand side of the screen
- SFX for correct word input
- Highlight if a word matches a word on input box
- SFX for misses
- SFX for losing

Arcade Mode:

Functional Requirements:

- Selectable length of the words that will appear
- Number of lives that decreases if a word goes beyond the screen
- Scoring system and a local storage method which stores the highest score or a leaderboard
- Increasing the number of words that appear and its speed as the level increases
- Input box for the user to see what they type
- Pause button

Non-Functional Requirements:

- Music
- SFX for correct word input
- Highlight if a word matches a word on input box
- SFX for decreasing lives
- SFX for losing

Modules and libraries used:

- *PyGame (requires installation)*

This library is used to render graphics, handle user inputs, and managing audio. Pygame is built on top Simple DirectMedia Layer (SDL) library. Originally game development requires lower-level programming languages. But pygame abstracts the functions which makes it easier to write programs on the high level that is python. Which mean i don’t have to write a game engine from scratch and just focus on making the game.

- *Random*

The random module allows me to generate pseudo-random numbers without having to create my own function. Examples:

-random.randint(a, b): Returns a random integer between a and b (inclusive)

-random.choice(seq): Returns a random element from a non-empty sequence

- *Time*

For tasks that require manipulation of time. In this case it is for creating a spawn interval for the words.

- *Copy*

Provides functions to create shallow and deep copies of objects. Deepcopy is used to access the original object’s value if there are algorithms that might change it.

- *NLTK (requires installation)*

NLTK is a powerful natural language processor which provides many functions often used by data scientists. It also provides extensive language corpora mainly for english

Progress

Before i make things pretty i want to make the meat of the game first which is generating words, moving them and make the input checker logic amongst other things.

My idea is to make some sort of object that will contain a randomly selected word moving from outside the screen to the left. This should be simple enough because i remember from making the chrome dinosaur game where the cactus obstacles are moving from right to left. I just have to do the same thing for the words.

```
class Word:
    """Class to represent a word."""
    def __init__(self, text, speed, x, y):
        self.text = text
        self.speed = speed
        self.x_position = x
        self.y_position = y

    def draw(self):
        color = 'red'
        screen.blit(font.render(self.text, True, color), (self.x_position,
self.y_position))
    def update(self):
        self.x_position -= self.speed
```

Next i need a function to generate the word.

```
def generate_word():
    #Spawns a new word at a random position.
    global speed

    y_pos = random.randint(10, HEIGHT - 100)
    x_pos = random.randint(WIDTH, WIDTH + 300)
    text = random.choice(wordlist).lower()#selects the word at random

    new_word = Word(text, speed, x_pos, y_pos)
    word_objects.append(new_word) #to store whats on screen
```

-Wordlist:

I need a list of words that can be picked randomly to assign to text. For testing purposes during this stage i used a txt file containing random french word per line.

```
with open(" Algorithm and programing final/assets/lang_french.txt
", "r") as file:
    wordlist = [line.strip() for line in file.readlines()]
```

Later on I will add a feature where users can select the language. One of the solutions i had in mind was to have a txt file of words for each language, and make a function to select the languages somehow.

But i heard about this module called nltk which i heard could provide me a list of words for many languages. I'll consider what's practical later.

-Pausing:

In the game there will be a pause function because i need a way to place the menus. To do this, i need an if else statements where the game loop checks if pause condition is true or false. If it is true the the main gameplay logic, such as updating the UI, spawning words, and handling user input, is stopped and draws a pause menu.

I'm gonna use either inputs from keyboard or buttons such as esc for pausing and resuming which will be shown later in the game loop.

```
def draw_pause():

    pause_text = font.render("Paused. Press ESC to resume.", True, BLACK)
    screen.blit(pause_text, (WIDTH // 2 - 200, HEIGHT // 2 - 50))
```

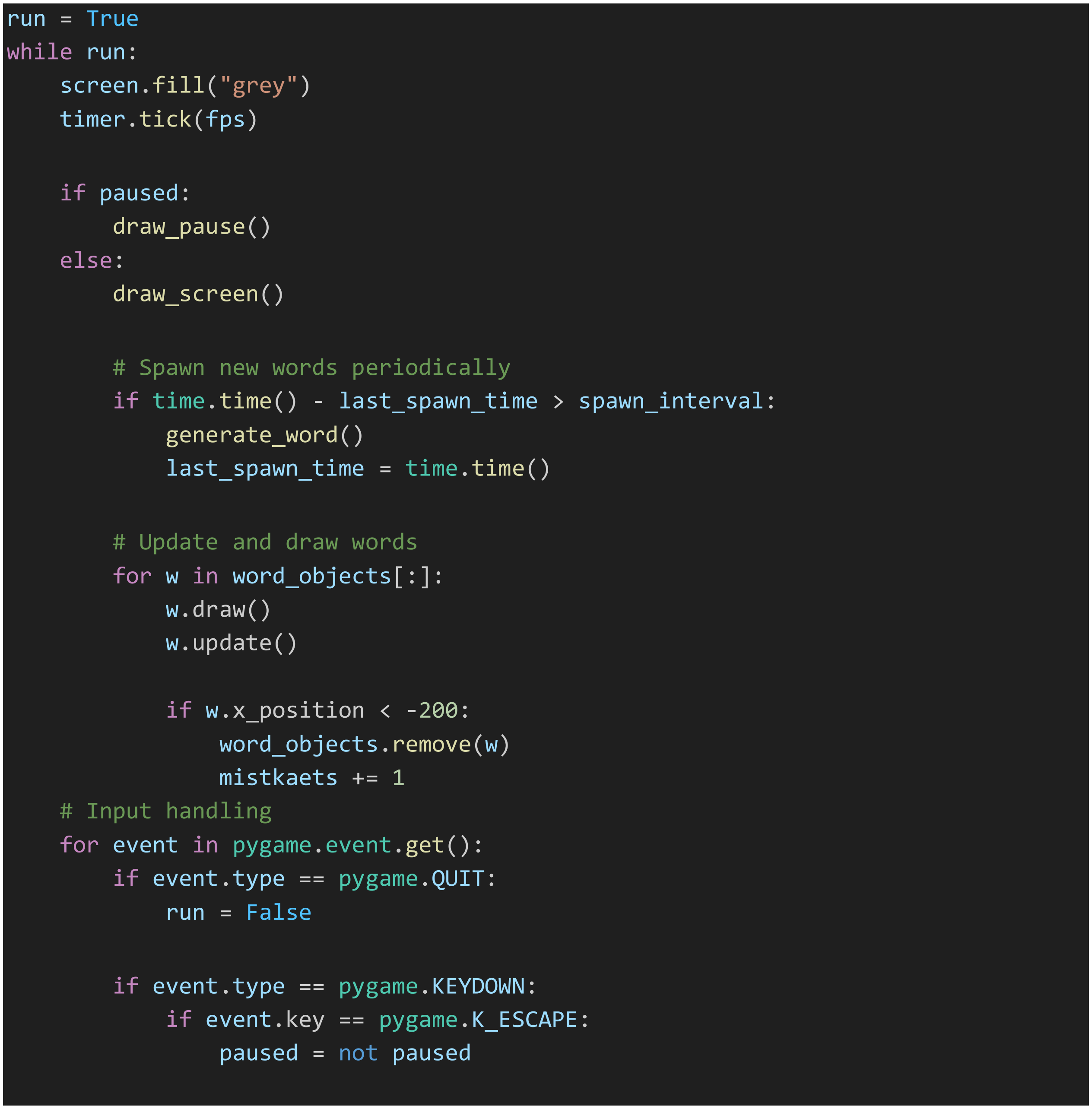
-Generating the words in game loop:

I wanted the words to appear periodically so here the time module could help with setting up the intervals. The algorithm under `# Spawn new words periodically,` has time.time() which returns the seconds since epoch time (January 1st 1970). During run time the time.time() begins counting the seconds as it increases it is subtracted by last_spawn_time which is assigned with what time.time() returns in the beginning of the game. If the result of the subtraction is bigger than spawn_interval (2 sec) then it generates and updates last_spawn_time with what the current time.time() returns.

To simplify the explanation let's say when running the game, the seconds is 100 since epoch time:

1. When `time.time() = 102`, the condition is met (`102 - 100 > 2`), and a word is generated.
2. `last_spawn_time` is updated to `102`.
3. The next word is generated at `time.time() = 104`, and the process repeats.

Game loop:



```
run = True
while run:
    screen.fill("grey")
    timer.tick(fps)

    if paused:
        draw_pause()
    else:
        draw_screen()

        # Spawn new words periodically
        if time.time() - last_spawn_time > spawn_interval:
            generate_word()
            last_spawn_time = time.time()

        # Update and draw words
        for w in word_objects[:]:
            w.draw()
            w.update()

            if w.x_position < -200:
                word_objects.remove(w)
                mistkaets += 1
    # Input handling
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                paused = not paused
```

-User text input

I then make a place for the user typing input and i need a way to check the answer after i submit it and to highlight it if it is correct before i submit it.

The idea is to assign what the user input to a variable (active_string) and if that variable has the same string as a word in the list `word_objects`, it removes the word and resets itself as empty string again.

```
word_objects = []
```

```
def check_answer():
    """Checks if the active string matches any word."""
    global word_objects
    for w in word_objects:
        if w.text == submit:
            word_objects.remove(w)
            return score + 10
    return score
```

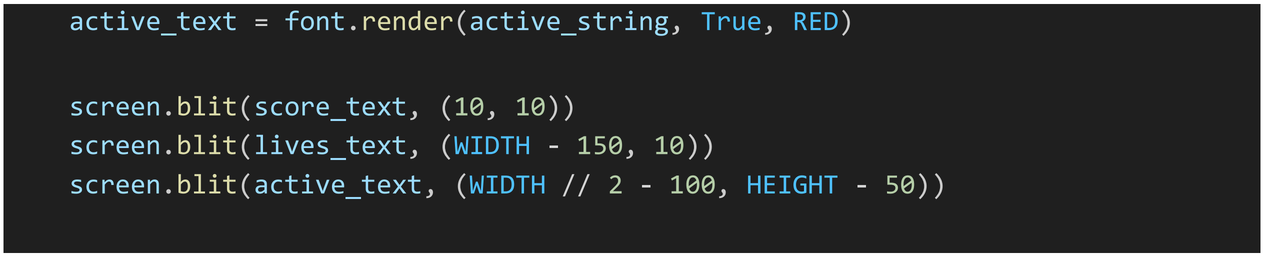
It is simpler to make a function for it to call later in the game loop when the user types and enter. This function checks whether a string in submit is the same as in a word in word_objects and removes said word and return a score the reason why I need another variable called submit is because during the game loop it acts as a sort of event listener which will be explained further later. The scoring here hasn't been defined yet so for now I'll just make it 10 for every word.

In the **Game loop**, inside "`if event.type == pygame.KEYDOWN:`" as shown above, I added this:

```
            if not paused:
                if event.unicode.isalpha():
                    active_string += event.unicode.lower()
                if event.key == pygame.K_BACKSPACE and len(active_string) > 0:
                    active_string = active_string[:-1]
                if event.key == pygame.K_RETURN:  #submitting answer
                    submit = active_string
                    active_string = ''
```

Before i explain i should mention there is a function called draw_screen. This is where i first made the overall layout for score, mistakes or lives, and the user input which is represented by active_string.

```
def draw_screen():
    """Draw the score, lives, and active string."""
    score_text = font.render(f"Score: {score}", True, BLACK)
    lives_text = font.render(f"Miss: {mistkaets}/10", True, BLACK)
```



```
    active_text = font.render(active_string, True, RED)

    screen.blit(score_text, (10, 10))
    screen.blit(lives_text, (WIDTH - 150, 10))
    screen.blit(active_text, (WIDTH // 2 - 100, HEIGHT - 50))
```

Now active_string is assigned an empty string and in the game loop *(if not paused or else it can still type during pause and the user can just cheat. The reason i put it outside else is because every other inputs like resuming is also there)*, “`if event.unicode.isalpha`” is used to check if the character associated with a keyboard event is an alphabetic character. Inside there’s a “`active_string += event.unicode.lower()`” This retrieves the Unicode character representation of the key pressed and appends that input to `active_string` with the +=. And the “`.lower()`” is just to make everything lower case. How I'd visualise this is:

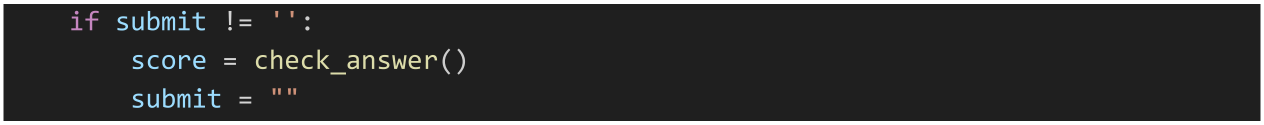
- Initially, `active_string` = "".
- After pressing A, `active_string` = "a".
- After pressing p, `active_string` = "ap".
- After pressing P, `active_string` = "app"

Then below that the “`event.key == pygame.K_BACKSPACE and len(active_string) > 0: active_string = active_string[:-1]`” just deletes whatever is inside active_string by assigning it with the version of it were without the most right-hand side letter after each press, and of course only if it has a string.

-Input Checker

Finally, how the user will submit their answer. In the **Game loop** input handling, “`if event.key == pygame.K_RETURN:`” when the user presses the enter key, it will assign whatever is inside `active_string` to `submit` and empties `active_string`.

Then, outside “`if event.type == pygame.KEYDOWN:`”, there’s a sort of event listener which is where `check_answer()` will be called:

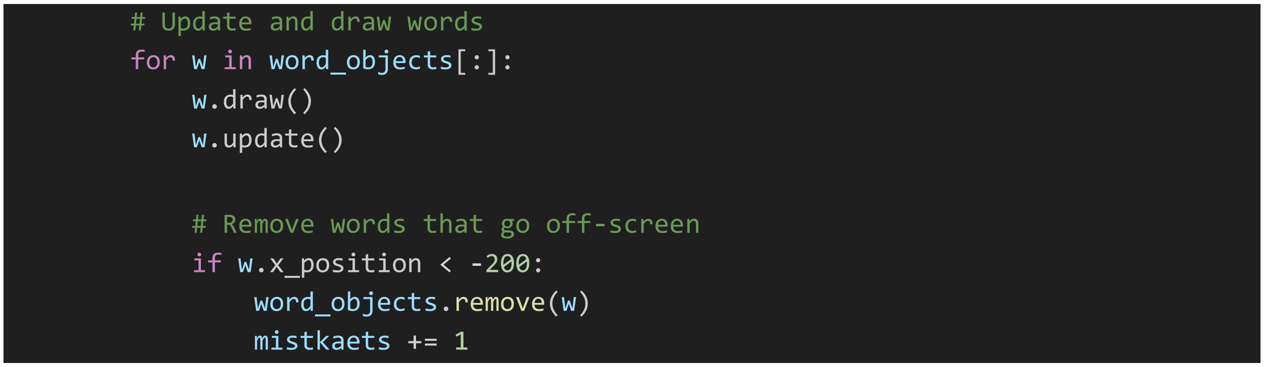


```
    if submit != '':
        score = check_answer()
        submit = ""
```

`check_answer()` will be called upon if submit contains a string. `Check_answer()` returns the score and is immediately assigned to score.

Recall that in `def check_answer` above, the function only removes a word in `word_objects` and returns a score `if submit == w.text`. Or else it does nothing (or later on the else could do the wrong sound effect).

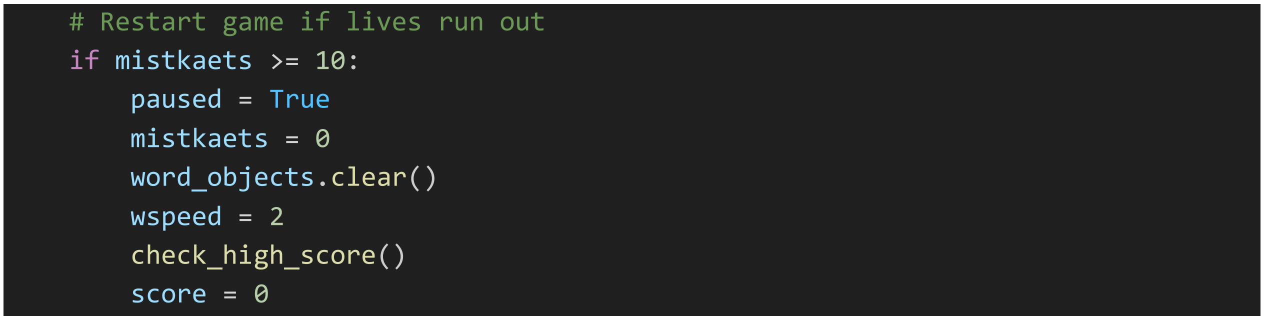
-Mistakes counter and reset



```
        # Update and draw words
        for w in word_objects[:]:
            w.draw()
            w.update()

            # Remove words that go off-screen
            if w.x_position < -200:
                word_objects.remove(w)
                mistkaets += 1
```

In the **Game loop** i call on the draw and update methods from the Word class for each word that appears in the word_objects list, so during run time it keeps updating. Without it, it won’t move nor update the colors.

Below the calls is where i put the mistakes counter, it basically checks whether an object passes through the limit of the screen and increments the mistakes until it resets.

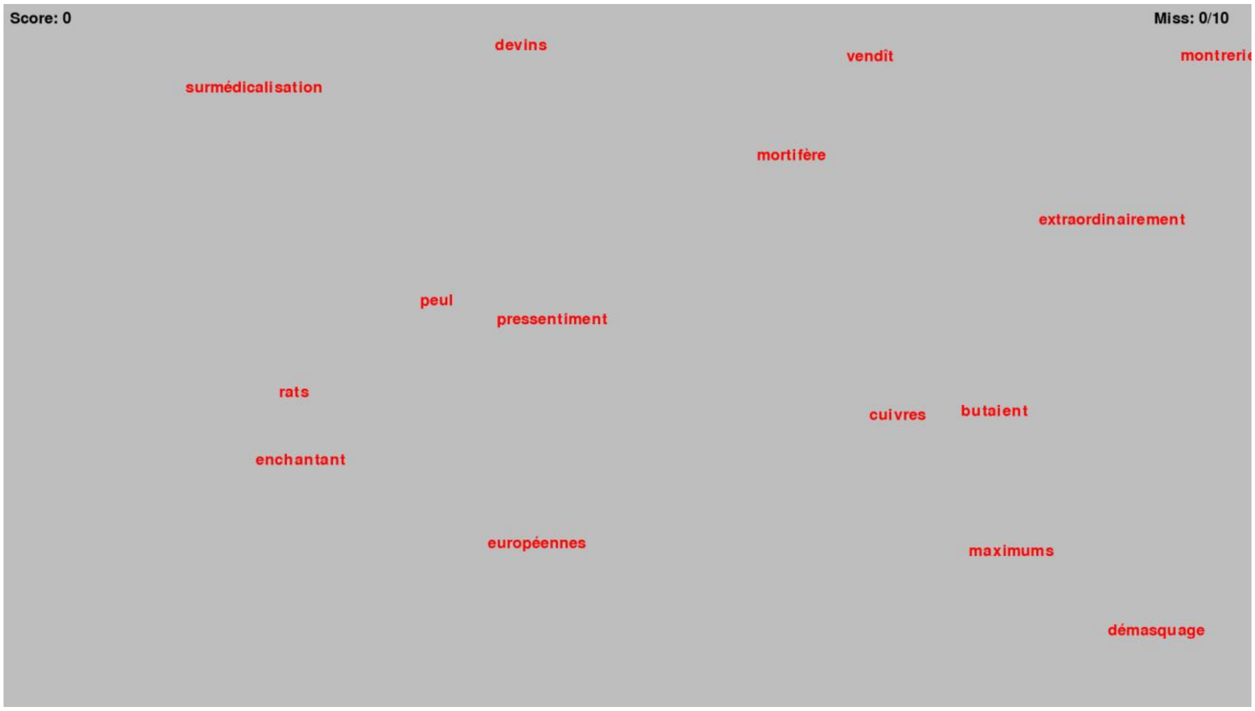


```
    # Restart game if lives run out
    if mistkaets >= 10:
        paused = True
        mistkaets = 0
        word_objects.clear()
        wspeed = 2
        check_high_score()
        score = 0
```

The resetting procedure is pretty simple. If mistake reaches 10 or more (just in case a word passes through at the right time and becomes 11), it draws the pause menu, resets the mistakes, clear the word_objects list, reset the speed, call on the high score function, and resets the score.

Test Run

Word generation:



Score: 0

Miss: 0/10

devins

vendît

montreri

surmédicalisation

mortifère

extraordinairement

peul

pressentiment

rats

cuivres

butaient

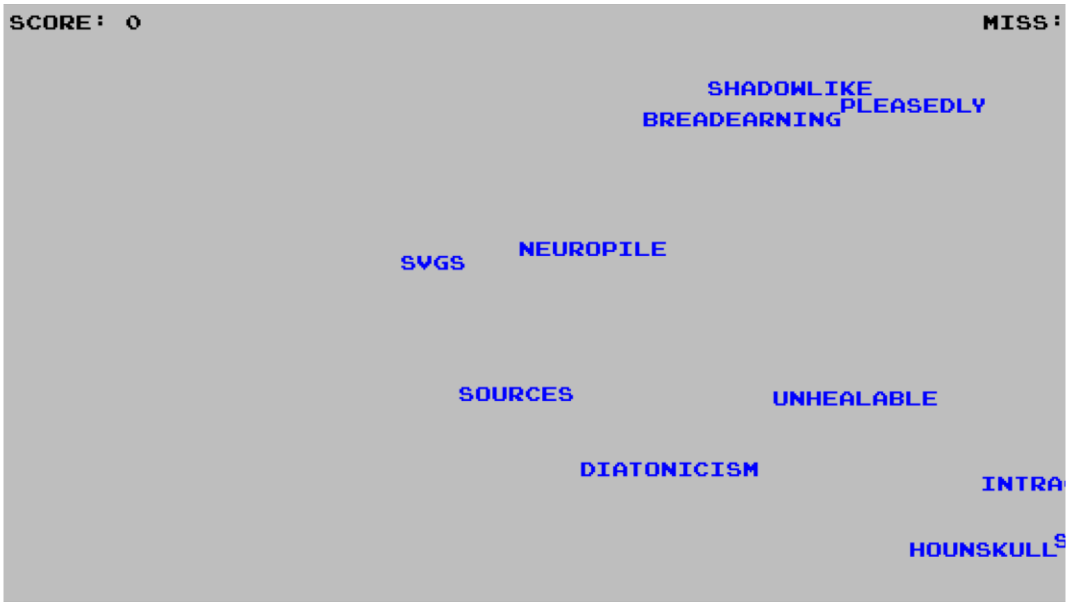
enchantant

européennes

maximums

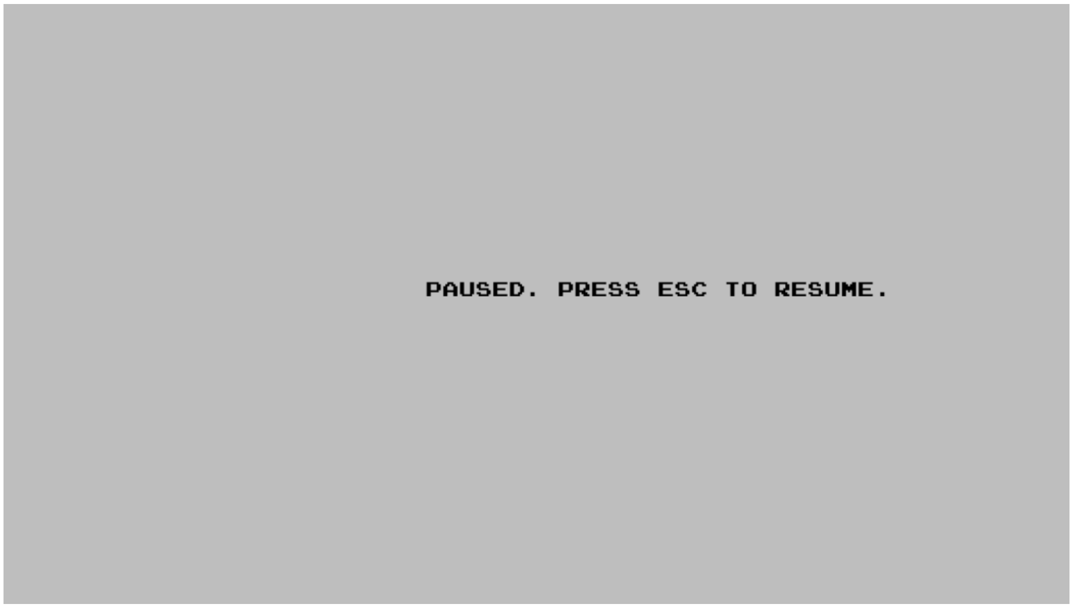
démasquage

Pausing:



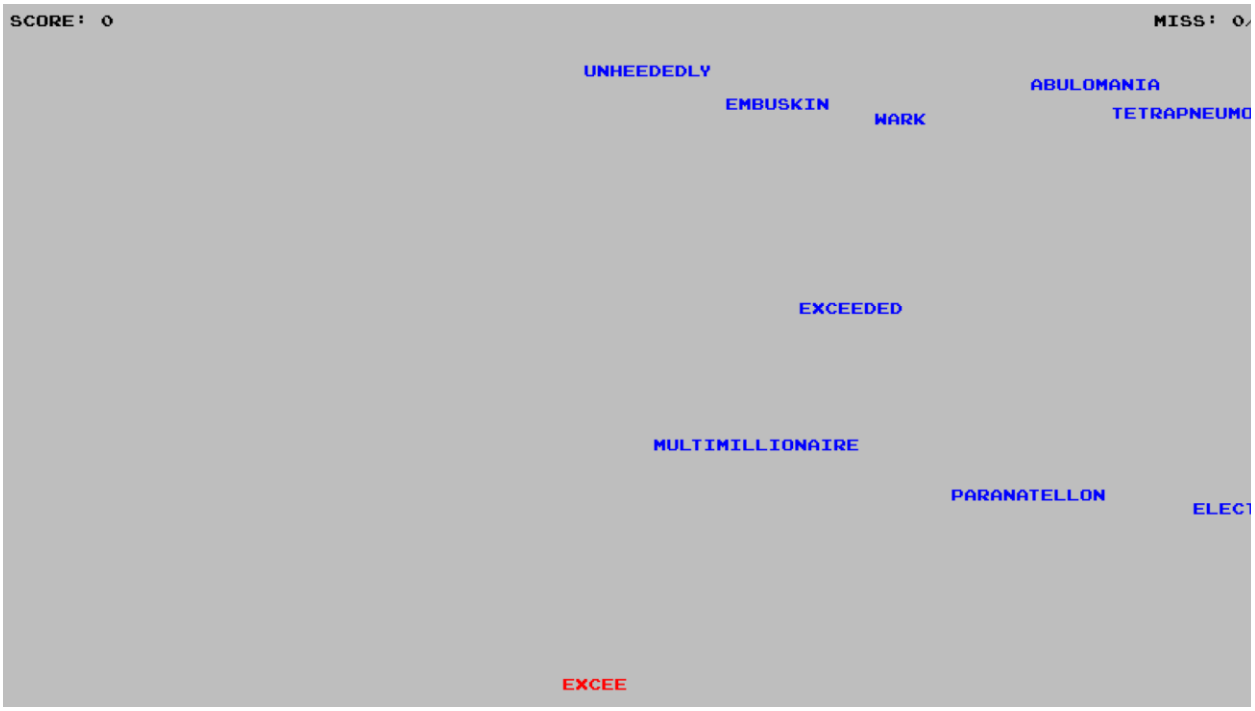
SCORE: 0

MISS:



PAUSED. PRESS ESC TO RESUME.

Input (red text):



Polishing the game layout and additional features

After knowing that everything works, i only need to make it look nicer. I want the game to have that 90s look like in Bisqwit’s game with the giant pixels for the texts and the shooting stars background.

First, I have to make the overall layout. Where to put the buttons, score, input box, etc. The tricky part will be making the languages button. So, I attempted to make that first.

-Gathering languages datasets

I first need to get the txt files for each language, i have already gathered the txt file for the French language which i acquired from lexique.org. I need it in a word per line type of format because my idea is to use a function that returns the txt file into a list using `file.read().splitlines()`.

The languages that will be featured in the game will be English, Dutch, and French. Later on, there will be many languages, but this is enough for now. For Dutch, I acquired the data from opentaal.org and for English from GitHub https://github.com/dwyl/english-words/tree/master.

-Drawing game layout and creating buttons

Making the game layout is very simple. I only have to use `pygame.draw.rect` and `pygame.draw.line`, tweak it to my preference and put it inside `def draw_screen` .

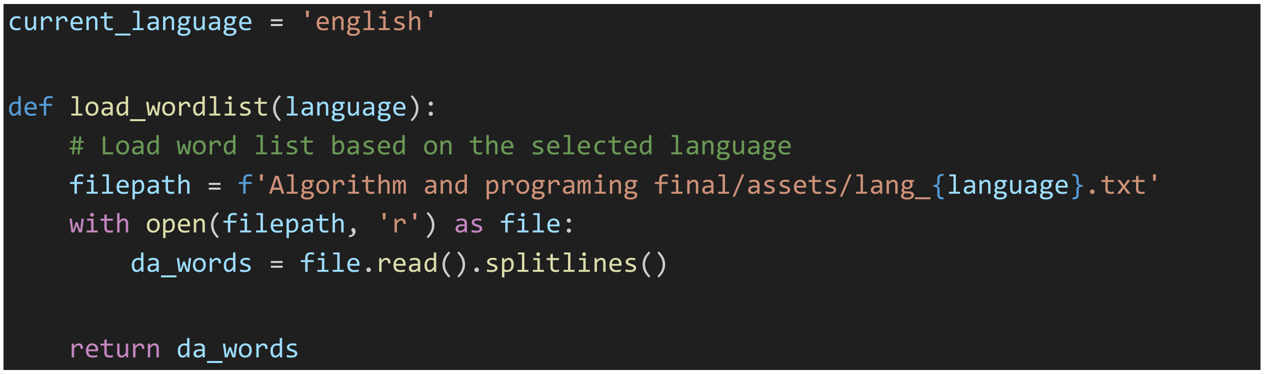
Next is the buttons. It's very simple just create a class for it along with its parameters.

```
class button:
    def __init__(self, x_position, y_position, text, clicked, surf):
        self.x_position = x_position
        self.y_position = y_position
        self.text = text
        self.clicked = clicked
        self.surf = surf

    def draw(self):
        cir = pygame.draw.circle(self.surf,(45, 89, 135), (self.x_position, self.y_position),
35) #arguments : (surface to draw, color, center position(x,y), radius)
        if cir.collidepoint(pygame.mouse.get_pos()):
            btn = pygame.mouse.get_pressed()
            if btn[0]: #in pygame button 0 is left click
                pygame.draw.circle(self.surf, (190, 35, 35), (self.x_position,
self.y_position), 35) #interface for feedback upon clicking
                self.clicked = True
            else:
                pygame.draw.circle(self.surf, (190, 89, 135), (self.x_position,
self.y_position), 35) #highlighted but not clicked

        pygame.draw.circle(self.surf,'white', (self.x_position, self.y_position), 35, 3)

        self.surf.blit(symbols_font.render(self.text, True, 'white'), (self.x_position - 15,
self.y_position - 25))
```

To make the buttons interactive, i use collidepoint to detect whether the mouse button is inside the circle. This can be used for a hovering feedback.

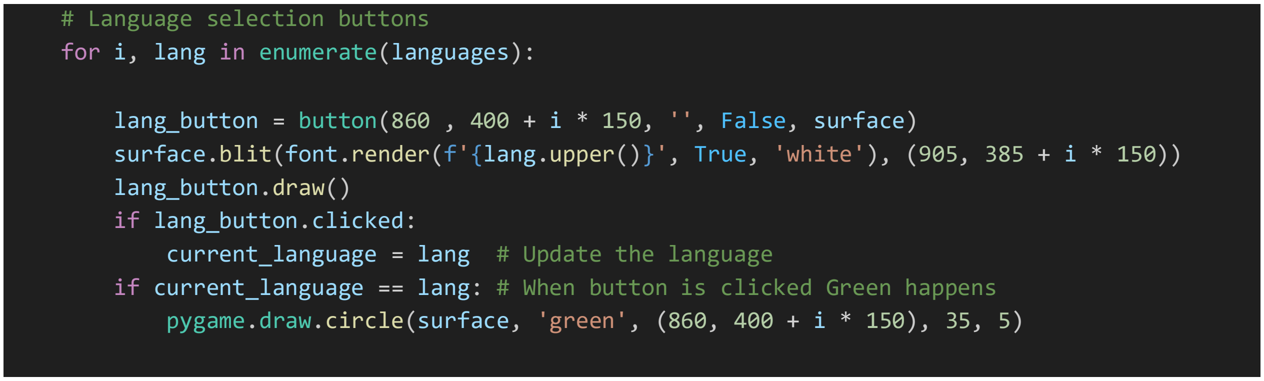
-Creating the function for accessing different languages

```
languages = ['english', 'dutch', 'french']  # Available languages
```



```
current_language = 'english'

def load_wordlist(language):
    # Load word list based on the selected language
    filepath = f'Algorithm and programing final/assets/lang_{language}.txt'
    with open(filepath, 'r') as file:
        da_words = file.read().splitlines()

    return da_words
```

For drawing the button (inside draw_pause). Using a for loop to make it easy to add more languages in the future:

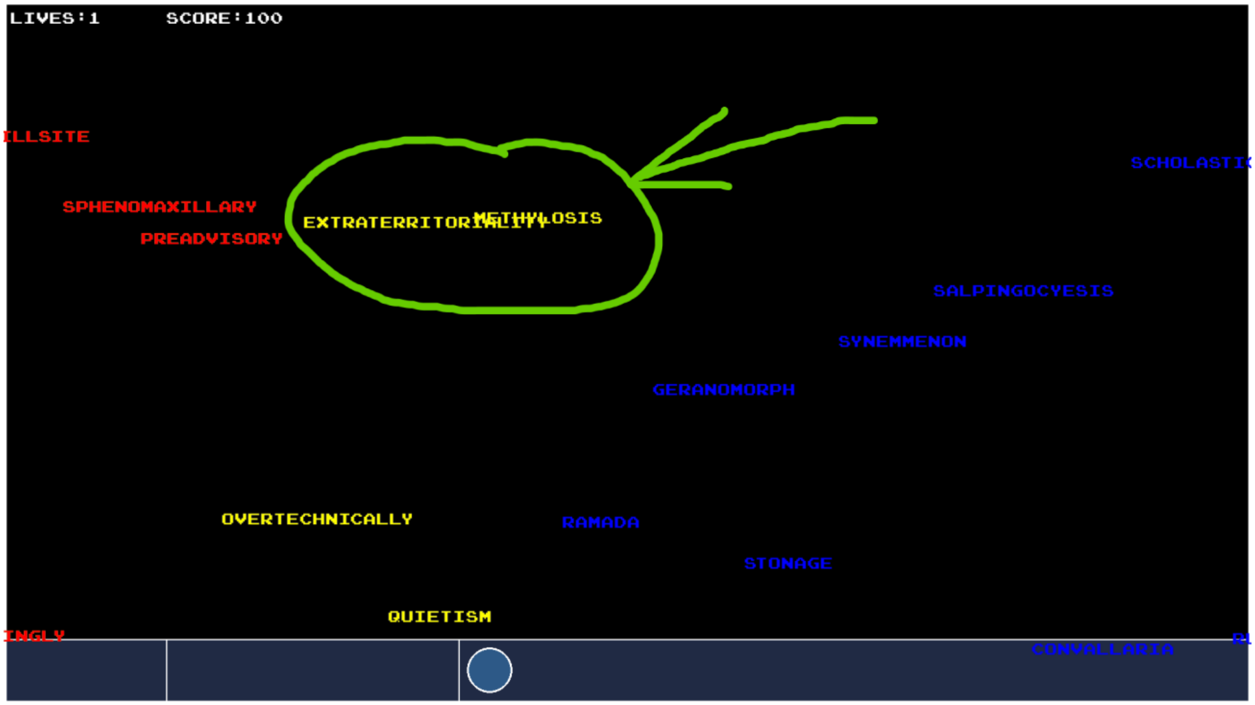


```
    # Language selection buttons
    for i, lang in enumerate(languages):

        lang_button = button(860 , 400 + i * 150, '', False, surface)
        surface.blit(font.render(f'{lang.upper()}', True, 'white'), (905, 385 + i * 150))
        lang_button.draw()
        if lang_button.clicked:
            current_language = lang  # Update the language
        if current_language == lang: # When button is clicked Green happens
            pygame.draw.circle(surface, 'green', (860, 400 + i * 150), 35, 5)
```

Arcade Mode

-Why?

Upon writing this I gotta be honest, the entire reason why I made Arcade mode (which is a different game entirely) is because of a bug that makes the words overlap in my attempt at making wspeed. The bug has been somewhat fixed, it still overlaps a little bit but not as bad, but it is the reason why I did not present the Classic mode.


LIVES:1 SCORE:100

The main cause of the problem is that sometimes the length of the word is just too long that it passes the spaces between letters (x-axis). And in the case of its y-axis it's because of the randomness which can be easily mitigated.

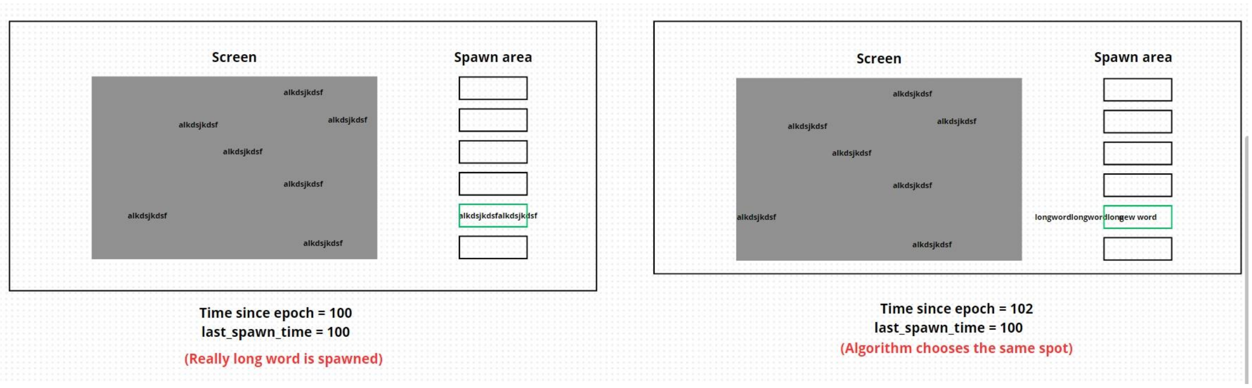
Now I could make some sort of boundary box for each words or maybe some kind of memory to store each word location and use some identifier to measure it’s length etc. But to me it is too complicated and this project has already taken it’s toll on me.

To make up for it i set up many other algorithms to set the lengths by playing around with the index of a sorted list of words. It will most likely be less fun to play than original wspeed and not as useful to improve typing.

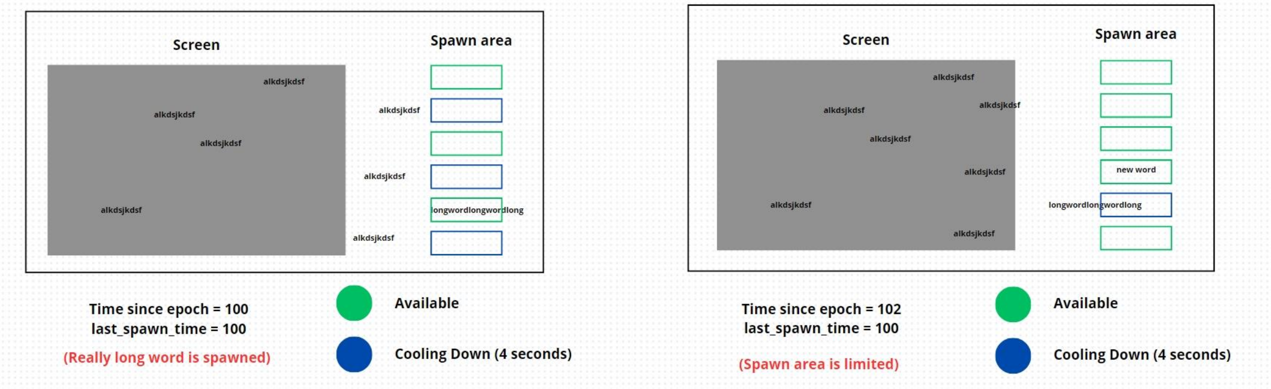
So in some ways I cheated by creating another game that can set a maximum length. But to be fair Wspeed was made by a genius using C.

-Problem Visualised + Potential solution

Problem:



Potential Solution:



Here, i could implement a cooling down system for certain areas after spawning. This solution however requires me to create a grid area each spawns a word and needs to cool down before spawning again, which probably requires a memory system and is very difficult for me so it could be another project on it’s own.

-Word length selection algorithm

```
def load_wordlist(language):
    # Load word list based on the selected language
    filepath = f'Algorithm and programing final/assets/lang_{language}.txt'
    with open(filepath, 'r') as file:
        da_words = file.read().splitlines()
    da_words.sort(key=len)  # Ensure words are sorted by length
    len_indexes = []    #[26, 453, 2583, 9769, 25689, 55563, 97561, ...]

    length = 1

    for i in range(len(da_words)):
        if len(da_words[i]) > length:
            length += 1
            len_indexes.append(i)
    len_indexes.append(len(da_words))
    print(len_indexes)
    return da_words, len_indexes
```

Inside draw_pause

```
    # define buttons for letter length selection
    for i in range(len(choices)):
        butn = button(360 + (i * 80), 700, str(i + 2), False, surface) #parameters : (x
position, y position, string starts at 2, status false or unoressed, where the button is drawn
which is pause menu in this case)
        butn.draw()
        if butn.clicked:
            if choice_commits[i]:
                choice_commits[i] = False

            else:
                choice_commits[i] = True
        if choices[i]:
            pygame.draw.circle(surface, 'green', (360 + (i * 80), 700), 35, 5)
```

Inside generate_level

```
    include = []
    for i in range(len(choices)): #iterates over choices list
        if choices [i] == True: #for choices that are true, do the following
            include.append((len_indexes[i], len_indexes[i+1])) #append 'include' list with
tuples of len_indexes of index i and i+1 (on the start of the level it would be (54,120) to
add to 'include' for example)
```

-New level logic

First i need to create a boolean object to create the condition for generating a new level. And another variable that is assigned with the level as it increases.

```
new_level = True
level = 1
```

Inside generate_level():

```
    for i in range(level): #so it generates for depending on the level, it iterates according to the level
        speed = random.randint(2 + level//3 , 3 + level//3 )#speed is random either 2 or 3 but increases as the level increases though i added //3 bcause it got too fast
        y_pos = random.randint(10 + (i * vertical_spacing), (i + 1) * vertical_spacing)
        x_pos = random.randint(WIDTH, WIDTH + 500) #so it generates from outside of the rightside of the screen

        index_sel = random.choice(include) #the choice of the tuples which is picked from 'include' is random
        index = random.randint(index_sel[0], index_sel[1])#for picking from wordlist using index with the range of the tuples from index_sel
        text = wordlist[index].lower()#this is why wordlist needed to be sorted

        new_word = Word(text, speed,x_pos ,y_pos) #draw the words using Word class
        word_obj.append(new_word) #adds Word to the word_obj list
    return word_obj
```

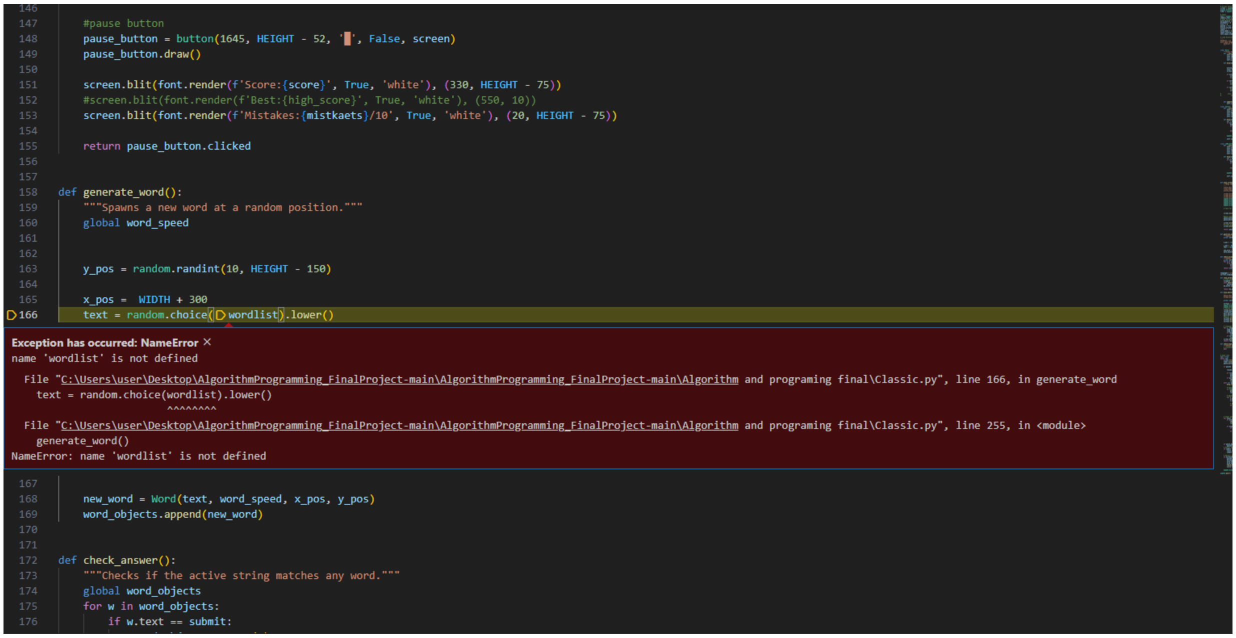
Inside **Game Loop**:

```
    if new_level and not paused: #if new_level is true and its not paused generate the words
        word_objects = generate_level()
        new_level = False
    if len(word_objects) <= 0 and not paused: #if there are no words left on the screen,
generate a new level
        level += 1
        new_level = True
```

-Using NLTK

Originally i was planning on avoiding using NLTK because it’s not included by default in all IDE’s, my concern was that something might go wrong if the installation is not set up correctly.

However there was a problem with using the reading files method. When starting up the game i seem to get an error if i click play too quickly. I have to wait just a few seconds before i can start playing or else this comes up:



```
146
147        #pause button
148        pause_button = button(1645, HEIGHT - 52, '█', False, screen)
149        pause_button.draw()
150
151        screen.blit(font.render(f'Score:{score}', True, 'white'), (330, HEIGHT - 75))
152        #screen.blit(font.render(f'Best:{high_score}', True, 'white'), (550, 10))
153        screen.blit(font.render(f'Mistakes:{mistkaets}/10', True, 'white'), (20, HEIGHT - 75))
154
155        return pause_button.clicked
156
157
158    def generate_word():
159        """Spawns a new word at a random position."""
160        global word_speed
161
162
163        y_pos = random.randint(10, HEIGHT - 150)
164
165        x_pos =  WIDTH + 300
166        text = random.choice(wordlist).lower()
```

```
Exception has occurred: NameError ×
name 'wordlist' is not defined

  File "C:\Users\user\Desktop\AlgorithmProgramming_FinalProject-main\AlgorithmProgramming_FinalProject-main\Algorithm and programing final\Classic.py", line 166, in generate_word
    text = random.choice(wordlist).lower()
                         ^^^^^^^^
  File "C:\Users\user\Desktop\AlgorithmProgramming_FinalProject-main\AlgorithmProgramming_FinalProject-main\Algorithm and programing final\Classic.py", line 255, in <module>
    generate_word()
NameError: name 'wordlist' is not defined
```

```
167
168        new_word = Word(text, word_speed, x_pos, y_pos)
169        word_objects.append(new_word)
170
171
172    def check_answer():
173        """Checks if the active string matches any word."""
174        global word_objects
175        for w in word_objects:
176            if w.text == submit:
```

My guess is that because my program has to read all the contents of the txt files first and convert them into a list which might slow it down, or it could be a hardware issue.

The great thing about NLTK amongst other things, is that it provides language corpus in the form of lists made tailored to python. All i have to do is import their words corpora and assign it to `wordlist` which i learned from: https://stackoverflow.com/questions/28339622/is-there-a-corpus-of-english-words-in-nltk and chapter 2 section 4.1 from their documentation for general wordlists.

I didn’t want to give myself headache with their data installation and file paths rerouting. Fortunately, i didn’t need to do that and went with their interactive installer route. https://www.nltk.org/data.html

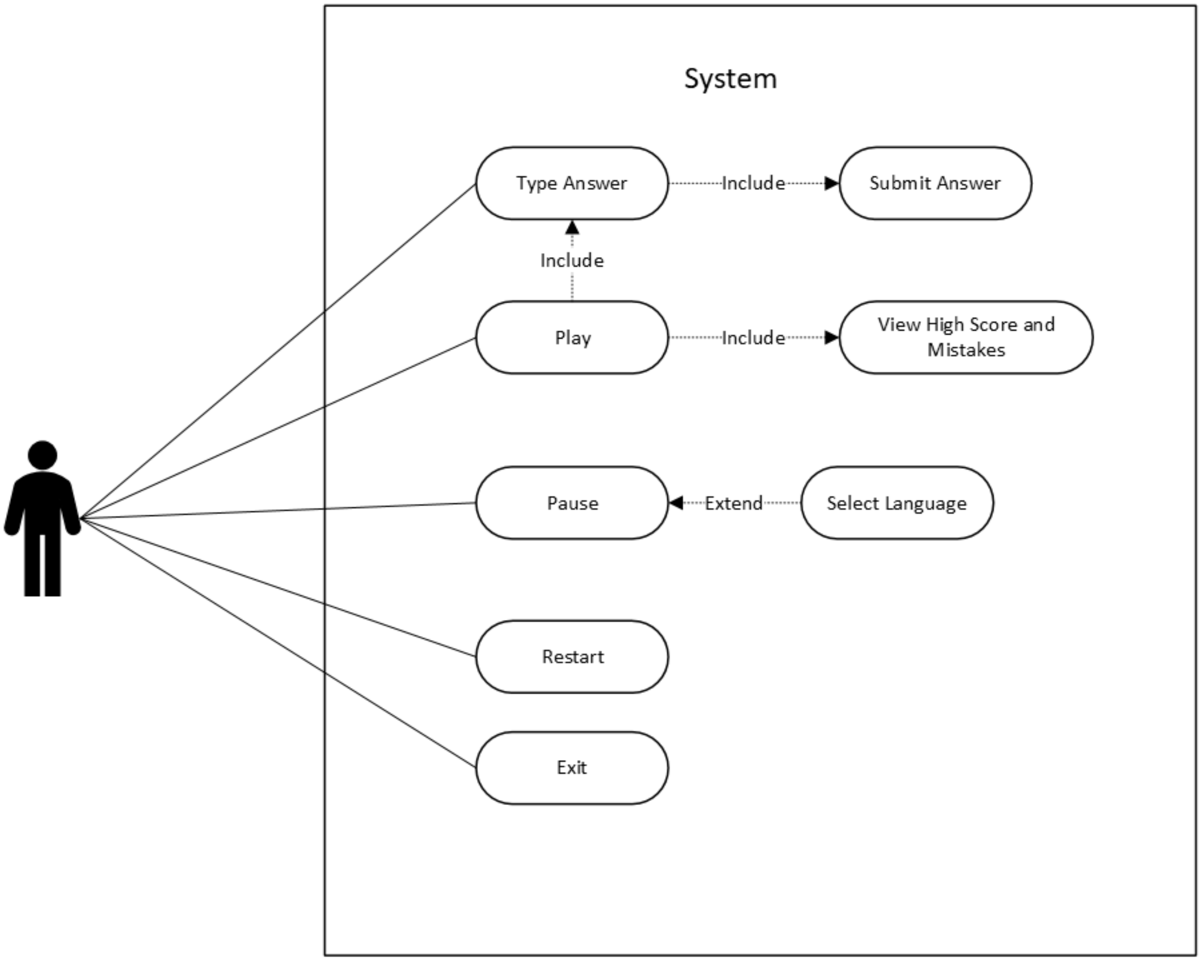
```
import nltk
from nltk.corpus import words
nltk.download('words')

wordlist = words.words()
```

Unfortunately, NLTK only provides language corpora for english which means i cannot implement a language selection in this version without using other, more advanced natural language library.

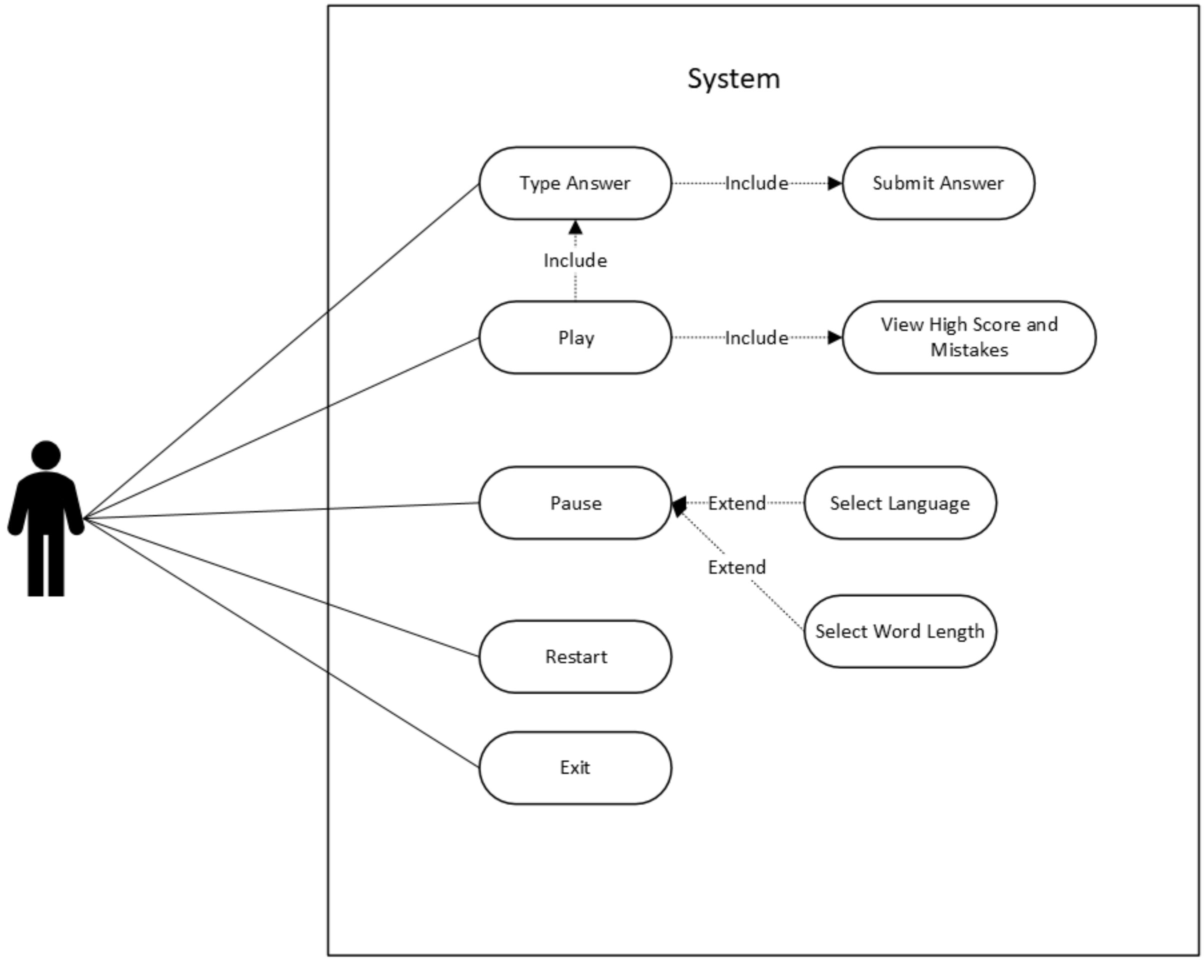
Diagrams

-UML Use Case Diagram

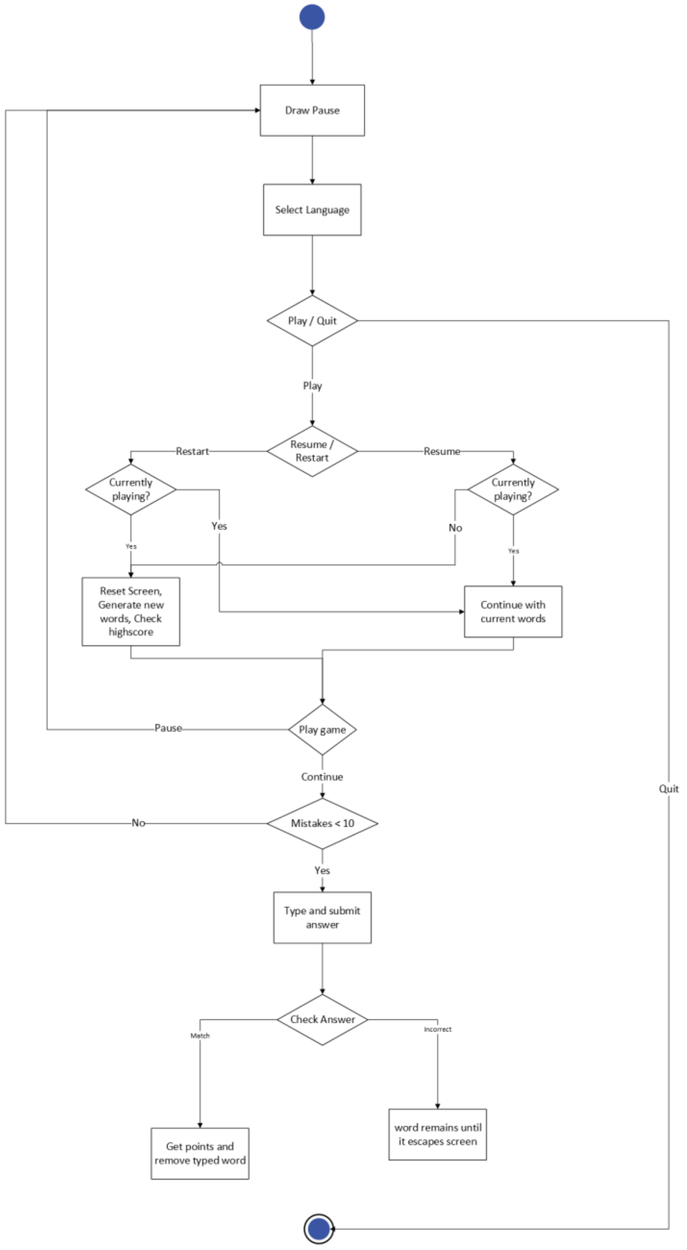
Classic


System
Type Answer
Include
Submit Answer
Include
Play
Include
View High Score and Mistakes
Pause
Extend
Select Language
Restart
Exit

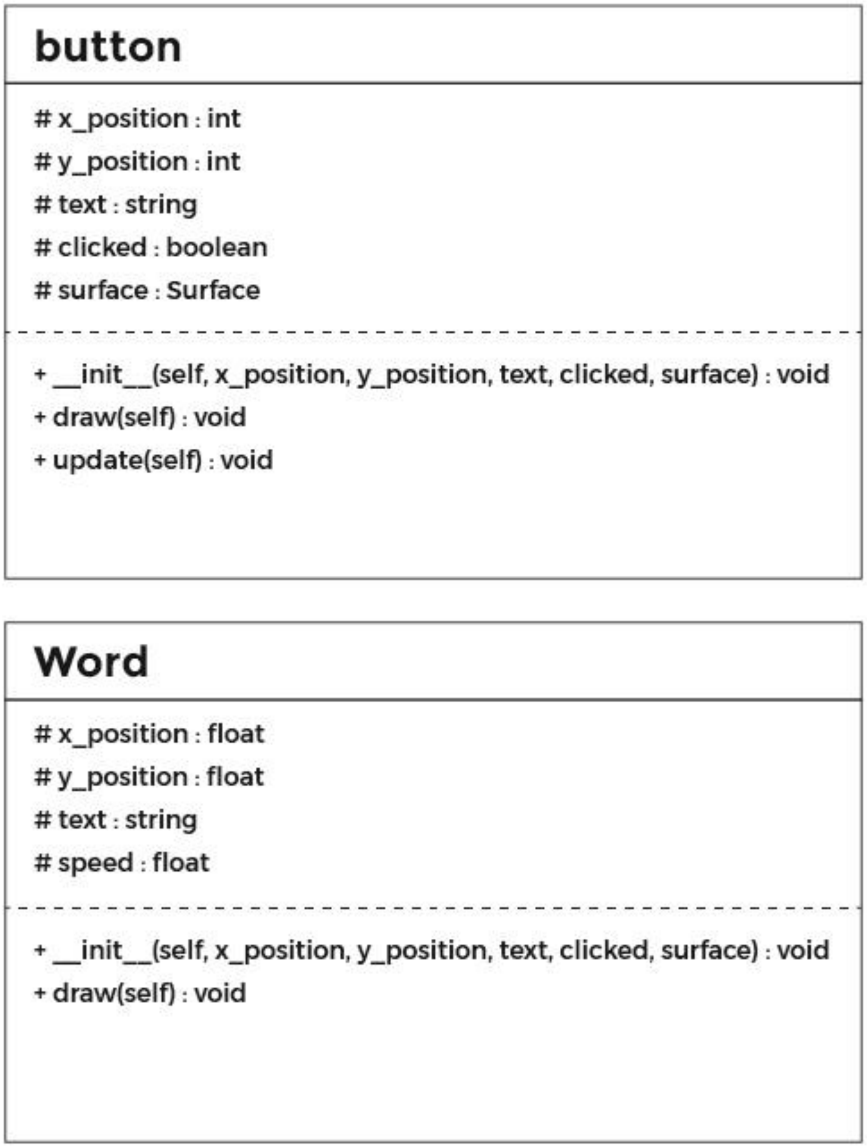
Arcade


System
Type Answer
Include
Submit Answer
Include
Play
Include
View High Score and Mistakes
Pause
Extend
Select Language
Extend
Select Word Length
Restart
Exit

-UML Activity Diagram



-UML Class Diagram



button

x_position : int
y_position : int
text : string
clicked : boolean
surface : Surface

+ __init__(self, x_position, y_position, text, clicked, surface) : void
+ draw(self) : void
+ update(self) : void

Word

x_position : float
y_position : float
text : string
speed : float

+ __init__(self, x_position, y_position, text, clicked, surface) : void
+ draw(self) : void

Classic vs Arcade Mode

Features	**Classic**	**Arcade(ML)**	**Arcade(NLTK)**
Word length selection	No	Yes	Yes
Language selection	Yes	Yes	No
Variable word speed	No	Yes	Yes
Word missed	Increasing Mistakes	Decreasing Lives	Decreasing Lives
New level logic	No	Yes	Yes
Color code for words location	Yes	No	No
Scoring system based on word length	Yes	Yes + Speed	Yes + Speed
Restart	Yes	Yes	No
Screen	Full Screen	Full Screen	Small Window

Known Bugs

Classic:

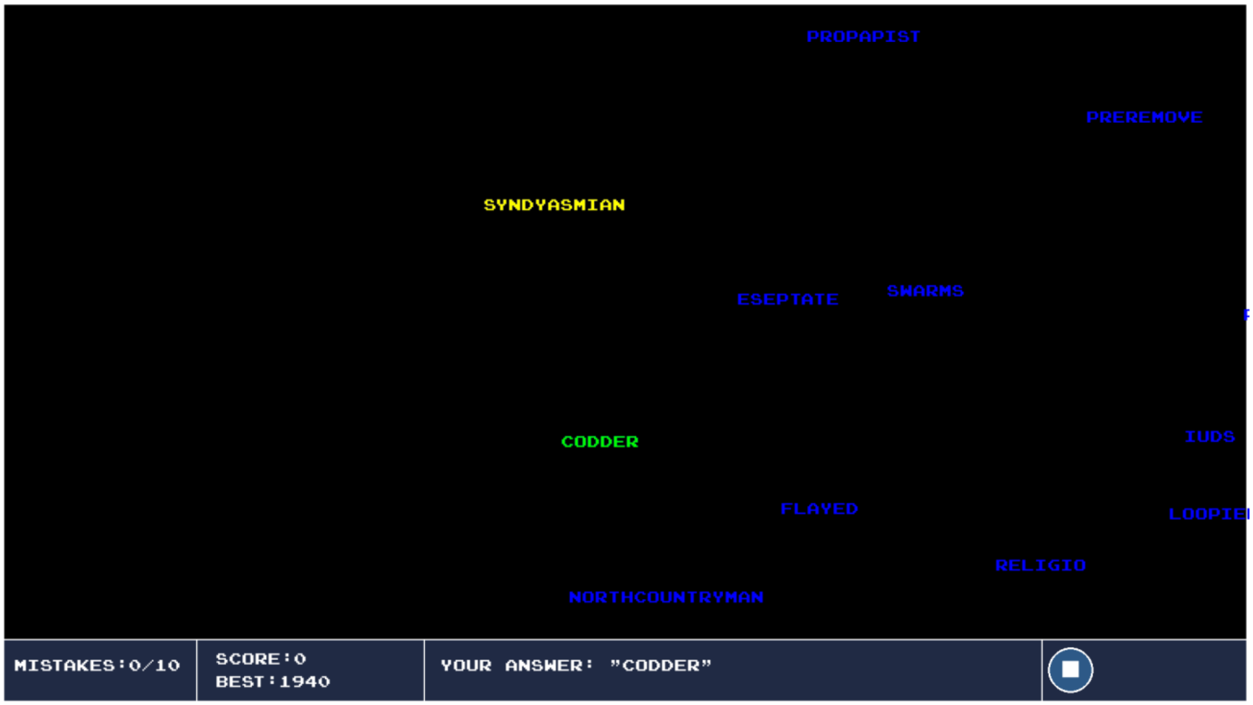
- Cannot hold backspace to delete letters (Gotta delete one by one)
- Word Overlapping
- Word matching highlighter (green) is overlapped by word location color code

Arcade:

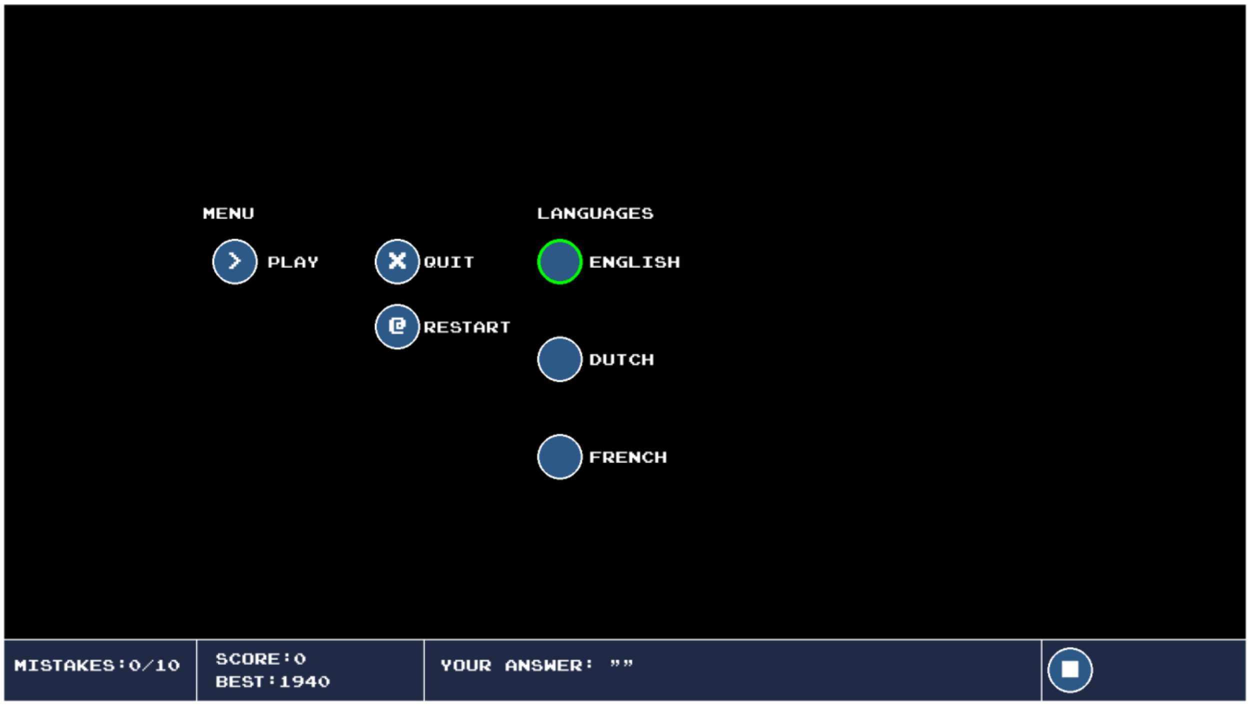
- Cannot hold backspace to delete letters (Gotta delete one by one)
- Loading wordlist is slow on multi language version

Gameplay

Classic:

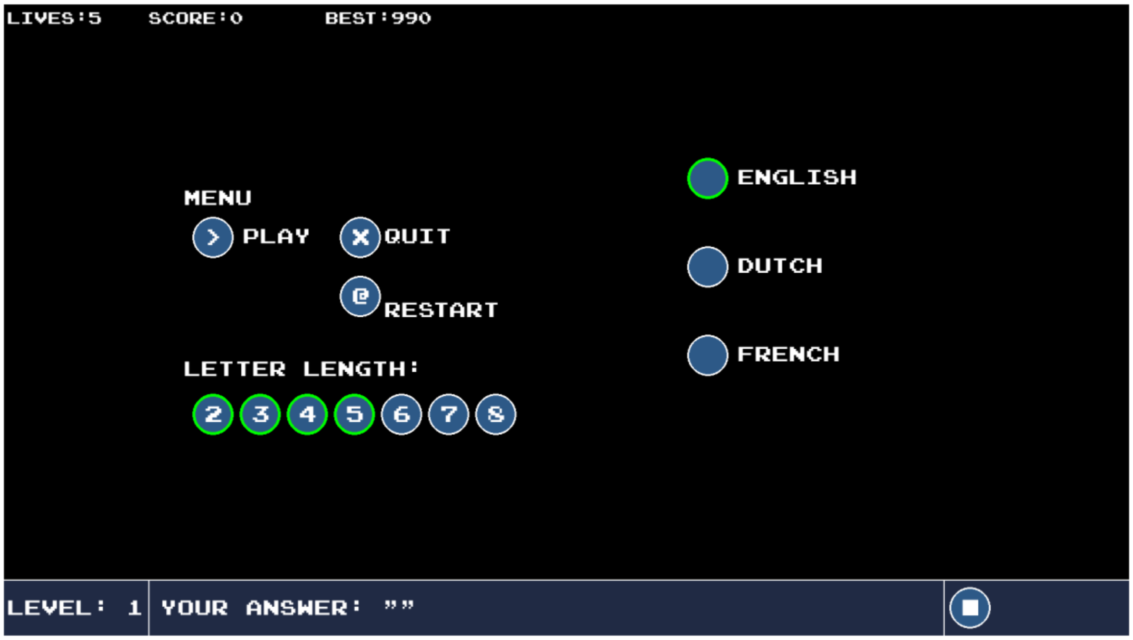


MISTAKES:0/10
SCORE:0
BEST:1940
YOUR ANSWER: "CODDER"



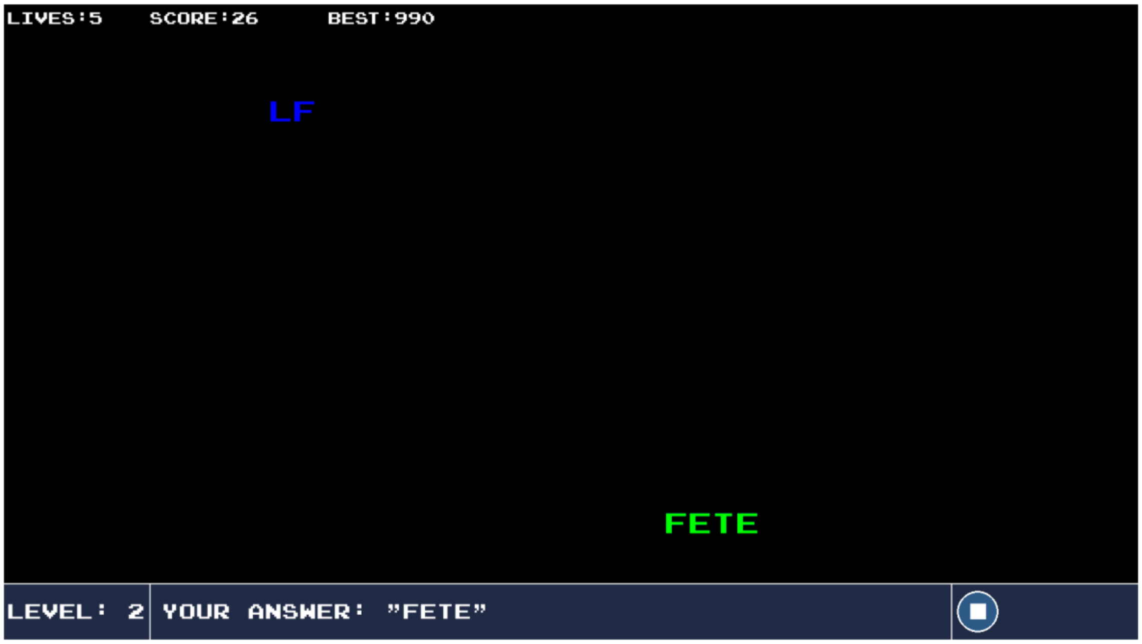
MISTAKES:0/10
SCORE:0
BEST:1940
YOUR ANSWER: ""

Arcade (Multi Language):



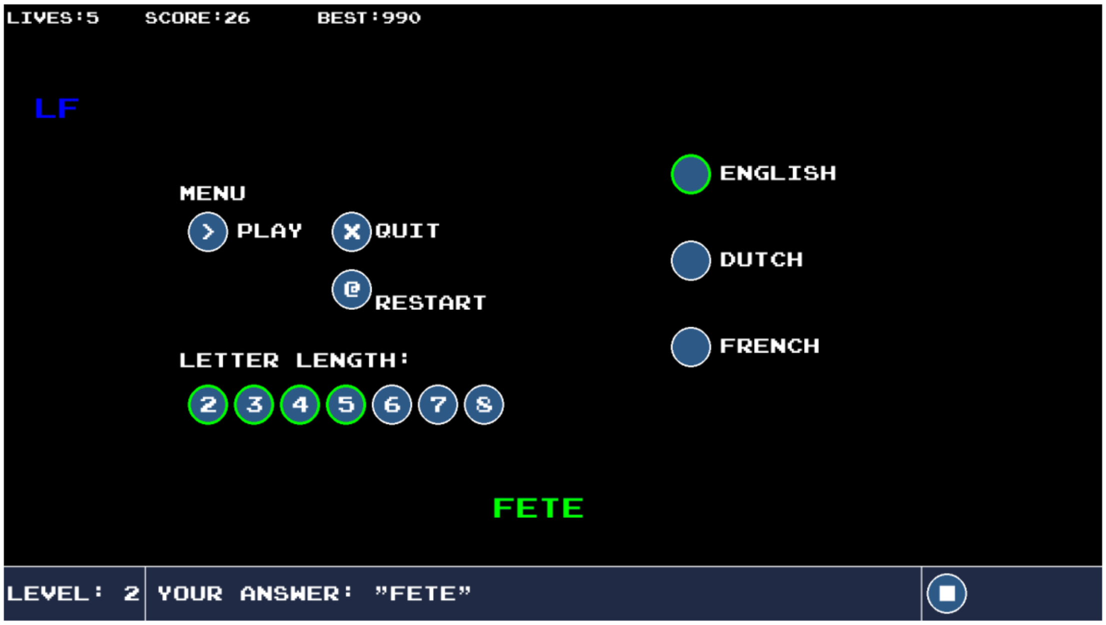
LIVES:5 SCORE:0 BEST:990

LEVEL: 1 YOUR ANSWER: ""

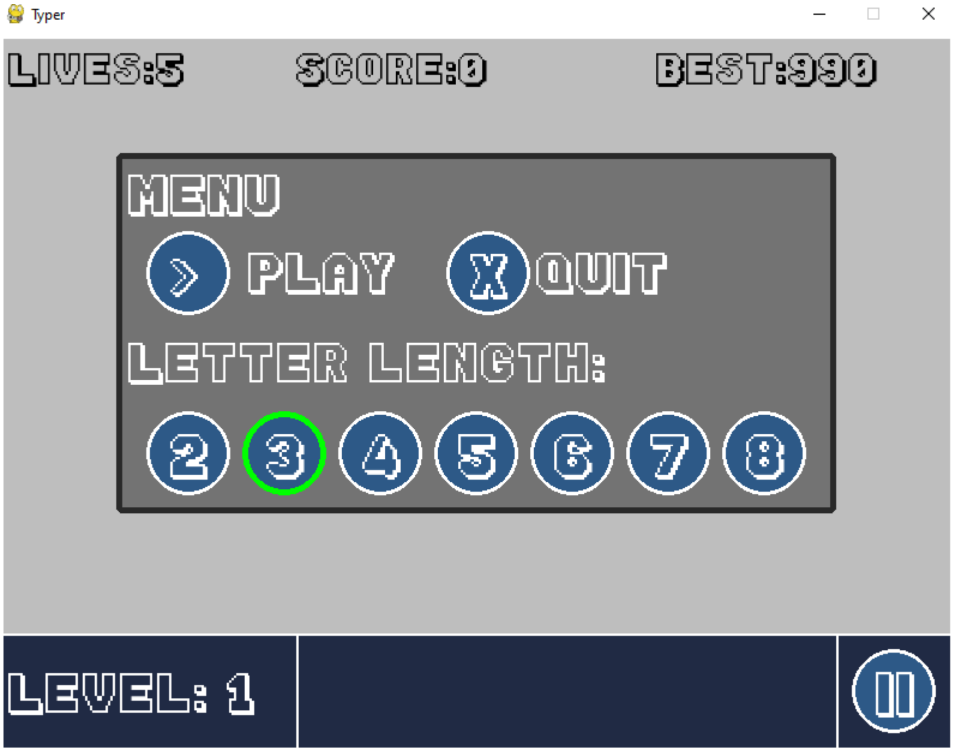


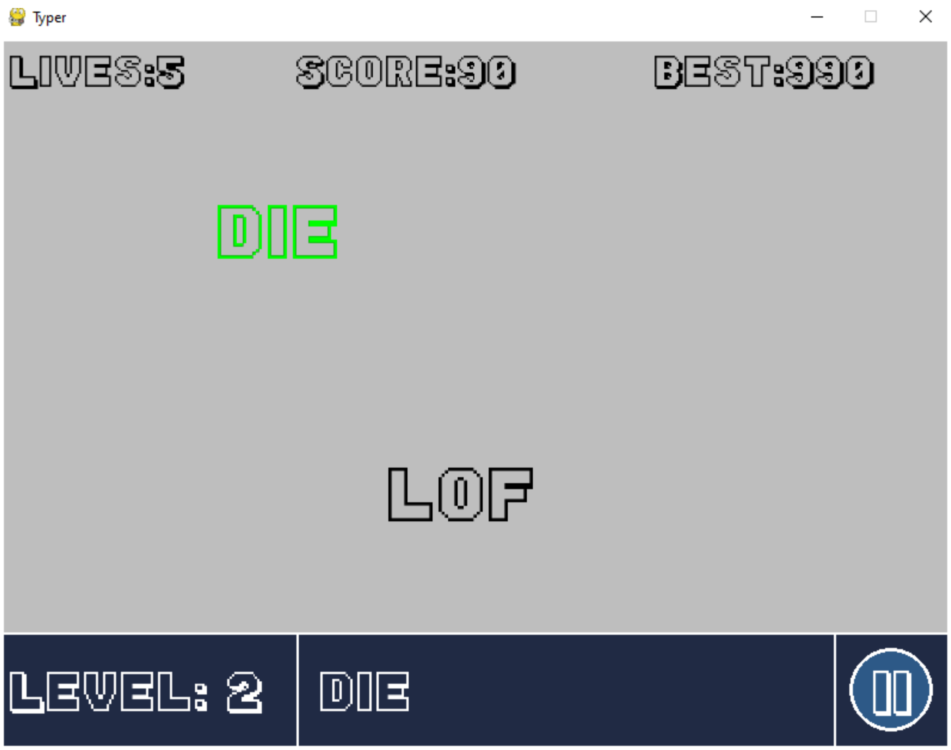
LIVES:5 SCORE:26 BEST:990

LEVEL: 2 YOUR ANSWER: "FETE"


LIVES:5
SCORE:26
BEST:990
LF
MENU
PLAY
QUIT
RESTART
LETTER LENGTH:
2 3 4 5 6 7 8
ENGLISH
DUTCH
FRENCH
FETE
LEVEL: 2
YOUR ANSWER: "FETE"

Arcade(NLTK):


Typer
LIVES:5
SCORE:0
BEST:990
MENU
PLAY
X QUIT
LETTER LENGTH:
2 3 4 5 6 7 8
LEVEL: 1



Conclusion

This game was supposed to be simple. But it turned out to be the most grueling and difficult learning process i have ever endured.