Block Inherits from Sprite

image: pygame.Surface(int, int)
image.fill: tuple
screen: mainSurface
rect: image.get_rect()
speedx: int
rect.x: int
rect.y: int

update(self): None

Game

mainSurface: pygame.display()
set_mode(tuple, int, int)
blocksGroup: pygame.sprite.Group()
player_group: pygame.sprite.Group()
score: int
player: Player
player.rect.x: int
player.rect.y: int

run(self): None

main.py

__main__(): None

Player Inherits from Sprite

image: pygame.Surface(int, int)
size: int
image.fill: tuple
screen: mainSurface
rect: image.get_rect()
increment: int
calibrate_high: int
calibrate_low: int
self.data: numpy.array
fs: int
rect.x: int
rect.y: int

on_event(self, event): None
collide(self, obstacle, score): None
analyze_freq(self)
    reads from wav file
remap_interval(self, frequency): int
freq_movement(self, frequency)

record.py

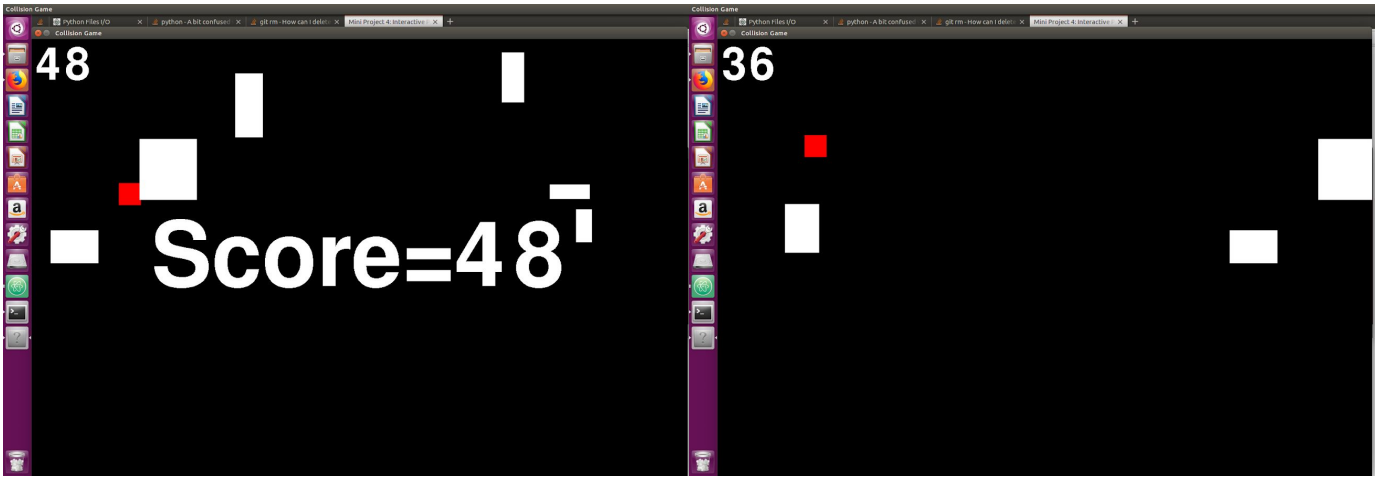
record(int, pyaudio.paInt16, int, int, int, String): None
    constantly records time intervals in separate file to be processed by main

Project Overview

Our interactive program is an obstacle avoidance video game controlled by pitch. As the player moves forwards, they must avoid blocks by moving up or down in correspondence to the pitch of their voice. Our goal was to explore interesting means of programing input, and to create a unique and fun experience for the player, as well as the people watching them play.

Results

The game works approximately as intended. A high pitched sound makes the block move up, a low pitched sound makes it move down. Running into a block ends the game, and also displays the score. Your goal is to dodge the blocks flying towards you and stay alive for as long as possible, using your voice to move up and down. It isn't beautiful, because we used basic pygame assets, but it works.



Implementation

Our program consisted of three main classes: Block, Player, and Game. Block and Player each inherited from the Sprite class, so that they could be used to process collisions. Block was used to create a group of Sprites of random sizes, moving across the screen in the negative x direction in random configurations, each at the same speed. This made it

appear as if the player were moving forwards and needed to avoid obstacles. The Player class needed to move differently, as it had to move up and down in response to the pitch detected by the computer's microphone. So that the recording interval did not determine the amount of time between frames, we had a separate script running in the background that constantly recorded short time snippets using pyAudio. Accordingly, our Player class processed the most recent recording in the analyze_freq() method using a Fast Fourier Transform to make a spectrogram and selecting the most prominent frequency. This frequency determined whether the Player moved up, down, or stayed in place in the freq_movement() class. Game initialized all aspects of the game display, including the screen, the Player, and the group of Blocks. The Run() method looped through the game cycle, rendering a screen, processing the user's input, checking for collisions to see if the game would end, and repeating.

One design decision that we made was the choice to have the separate script running in the background. Through a fair amount of trial and error, we found that 0.2 seconds was the smallest amount of time we could use for recordings to get any degree of accuracy in frequency processing. This is because the FFT is more accurate with longer time intervals. This does lead to some lag time while playing, as it takes time for the program to finish recording the next snippet to be processed, but it allowed us to have more than 5 frames per second in the rendering. Ultimately, we decided on 0.3 second recordings, as this seemed like the best compromise between lag and pitch accuracy.

Reflection

We definitely thought this would be easier than it was, and had high hopes for refining it past the point it is currently at. While it is currently functional, we wanted it to be smoother than it is. For the most part, though, the project went fairly smoothly. Work was spread out fairly evenly over the course of the time we had to finish it, and there weren't any bugs that had us stuck for longer than an hour or two. One problem we did run into was installing modules, specifically pyaudio, but because we had managed our time well, that hiccup didn't hurt us too badly. Git also caused some confusion with merge errors and deleted files (we actually just gave up on deleting old files), but we got past that.

As far as teamwork went, things worked pretty well. We split work pretty efficiently, mostly para-programming. This was probably the smoothest part of the whole process.

A coding challenge that ended up being harder than we thought was frequency analysis. Several libraries promised to be able to do that, but they either required something other than a .wav file or were just way beyond our level of comprehension (like librosa). The solution we found, using the FFT, was not ideal, because it gets less accurate for small chunks of time, but it worked well enough, and it was one of the more understandable and more functional examples we could find.

Going forward, we learned a lot about structuring of coding. This was our first time writing an extensive program with no formal structuring, and we found ourselves writing messy code as proof of concept, just getting it to work and cleaning up after. This is a good method to an extent, as it is beneficial to test code chunks as you go, but we learned some about the benefits of considering the structuring beforehand and organizing more regularly.