Algorithm & Programming

Week 13
Final Project Documentation

Brandon Salim
Class L1CC
2602177783

15th January 2023

1. Brief Description

For this final project, I made a rhythm game in pygame. As someone who likes playing rhythm games and is into music, I immediately thought of something related to it when the project was announced. Two of the major influences of this game is osu!mania and Friday Night Funkin' (FNF), specifically their 4K (four keys) maps. However, I guessed that a couple of other people would probably have the similar idea of making osu!mania, so I decided to change it up a bit.

Before going into that, I'm going to briefly describe the title screen. Upon running the game, the user is shown a title screen with two options. They can press enter to select the song they want, or press space to view the rules. The rules show the controls, as well as a feature discussed in the next paragraph. From this screen, the user may press enter to select the song, which starts the game, showing them the title of the song, its composer and the bpm before the game actually starts.

Alright, now back to the gameplay. Another rhythm game that I've played before is Just Shapes and Beats, a bullet hell rhythm game where you have to dodge projectiles that are timed to the music. So, I thought, what if I took a bit of each? In this game, the player controls a small icon placed on the bottom of the screen, with 4 lanes. Notes fall down in one of those 4 lanes, and the player must move between lanes to dodge the notes. An extra feature that I added is the ability for the player to switch between the 1st and 4th lanes. If the player moves right from the 4th lane, the icon moves to the 1st. Likewise, if the player moves left from the 1st lane, the icon moves to the 4th. This feature is described in the rules. I mostly did this to give me more options while making the chart, since this allows me to chart sections that utilize that feature.

Speaking of charting, a bunch of patterns used in the chart are inspired from the lots and lots of osu!mania and FNF charts I've played. For the songs, I initially only charted one, but I charted a second one because there's still time. The first song is "Final Hope" by Riya. I was introduced to this song from another rhythm game, A Dance of Fire and Ice (ADOFAI), and it's one of my favorite songs in the game. A month or so ago, ADOFAI had an update to their featured custom song list, which introduced me to another one of Riya's tracks: Polygons. Since I liked both songs, I decided to listen to more tracks by Riya, and found another one that I really liked: Cleyera. I used this as the second song for the game.

Going back to Final Hope, there is a section in the middle where the player is advised to memorize a pattern, as it keeps on repeating and distractions will be thrown at the player. This is a reference to the Final Hope level in ADOFAI, and also another rhythm game I liked: Rhythm Doctor. In ADOFAI, the player is required to tap once every beat. After 12 bars (48 beats), the screen pans upwards, showing just the background and some visual distractions. The player will have to keep hitting every beat. For that reason, I made my own pattern, then asked the player to remember it.

Afterwards, I'll distract them. Initially, I wanted to make it the same as ADOFAI, and completely hid the notes.

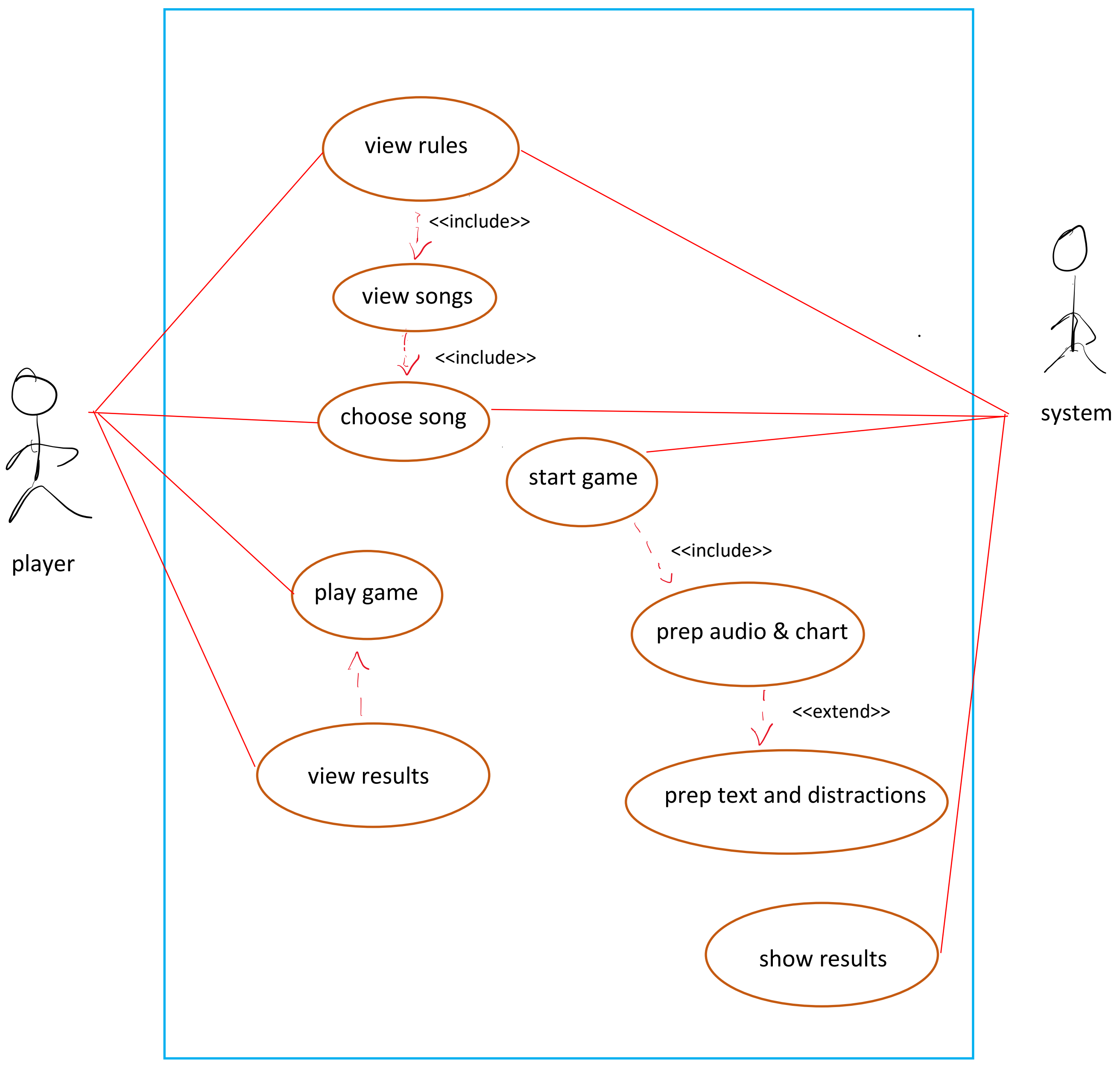
However, I found that this is too difficult for a certain reason. In ADOFAI, the player is required to hit on every beat. Although this section uses the hardcore subgenre of electronic, which uses distortion and saturation (thus making it harder to identify beats), most people with a sense of rhythm should still be able to hit every beat quite easily. However, since my game revolves around dodging the notes, the player can't hit on every beat. They have to hit right before the beat to switch to the safe lane. The combination of a fast bpm (220), the inability to see and the requirement to hit on offbeats are too nonintuitive.

For that reason, I scrapped the idea of completely hiding the notes, and took some reference from Rhythm Doctor, which also utilizes distractions. Kenneth (Jayadi Yu, from class B) also gave me some inspiration from a Guitar Hero (yet another rhythm game) map he saw online. In the video, an image of a hand is dropped down to cover roughly half the screen. This gives the player less time to react to incoming notes, but it shouldn't be a problem if they have memorized the pattern. I decided that it wasn't enough. After two more repetitions with the hand covering half the screen, I hid the player's icon by putting a rectangle with the same color as the background over it. For the last repetition, I changed the color of notes. These aren't fully invisible, but they blend in with the background more, so it's harder to see.

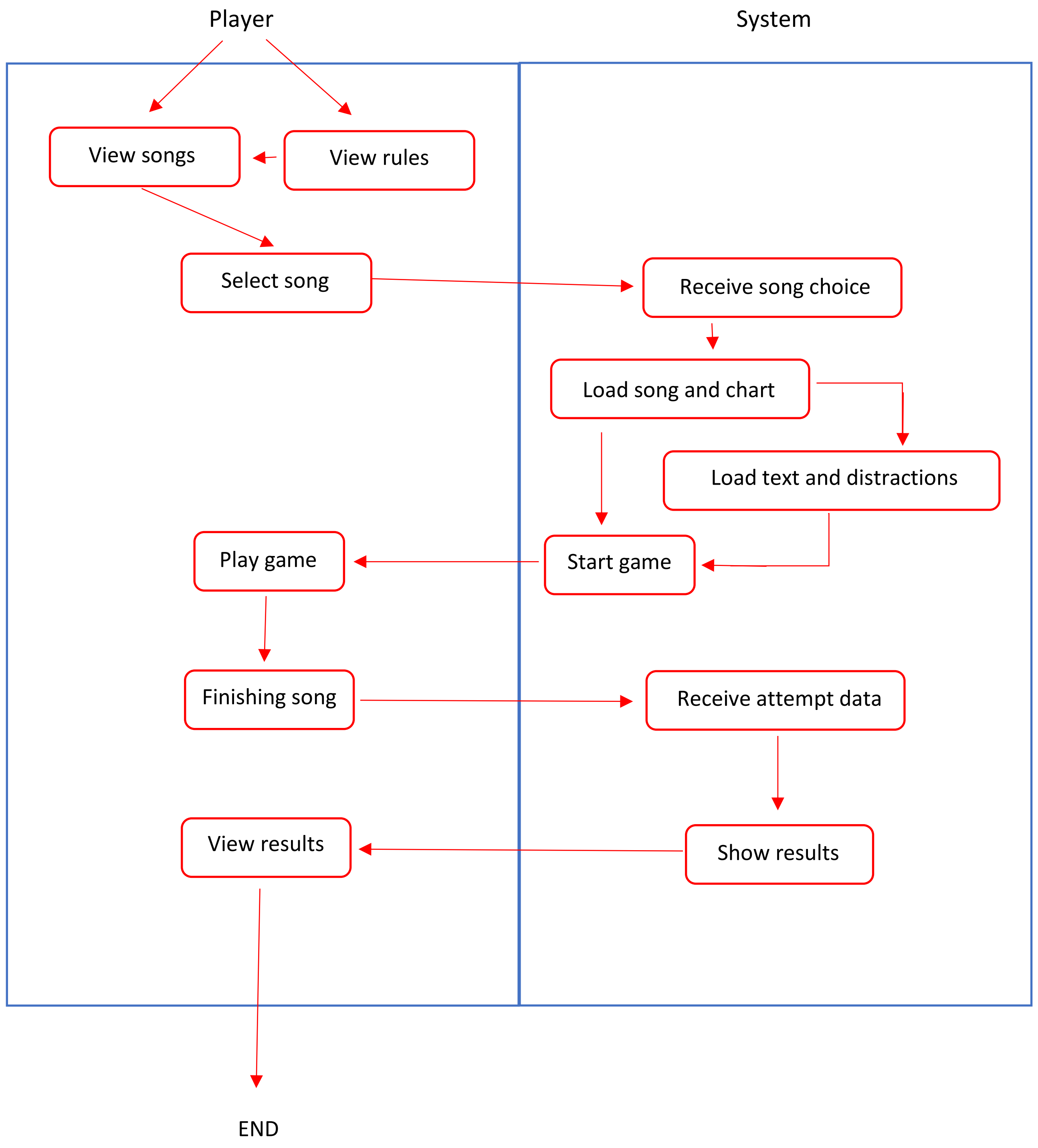
For Cleyera, I didn't put any distractions on purpose to create contrast between the two charts. Final Hope doesn't have the most difficult charting. It's not too dense, and the scroll speed is mediocre, giving the player quite some time to dodge. However, it does have distractions that could mess up the player in the middle. Cleyera is different. The second half of the track is packed with semiquaver notes (1/4th of a beat). It has faster scroll speed and smaller windows to dodge. In comparison to Final Hope, this chart doesn't distract the user, but it's a threat because of its sheer difficulty.

Once the player gets to the end of the song, they will be shown a results screen. This screen shows the total amount of notes in the song they just played, along with the number of notes they failed to dodge and the percentage of notes that they dodged.

2. Use-case Diagram

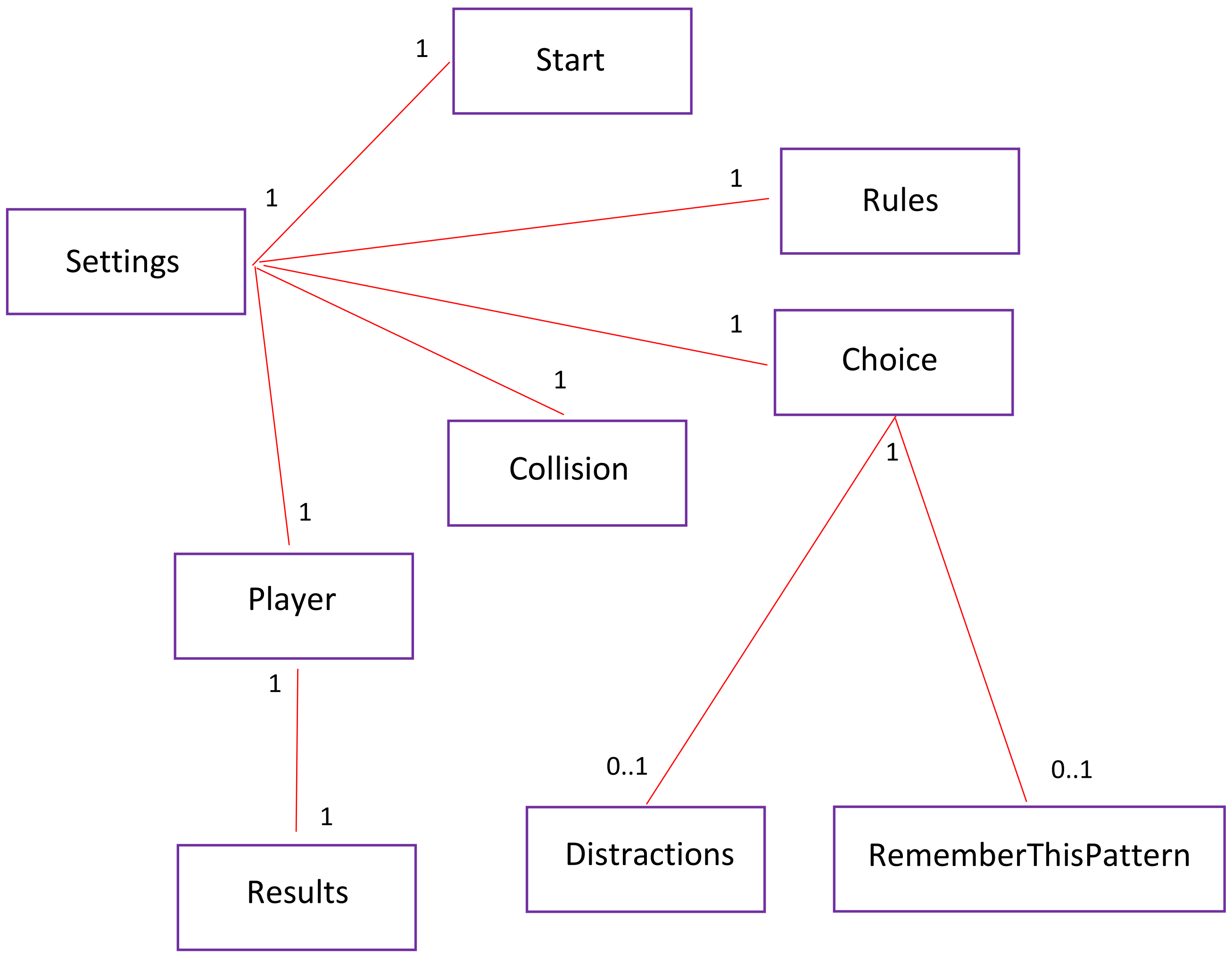


3. Activity Diagram



4. Class Diagram

All the classes: Settings, Player, Collision, RememberThisPattern, Start, Rules, Results, Distractions, Choice



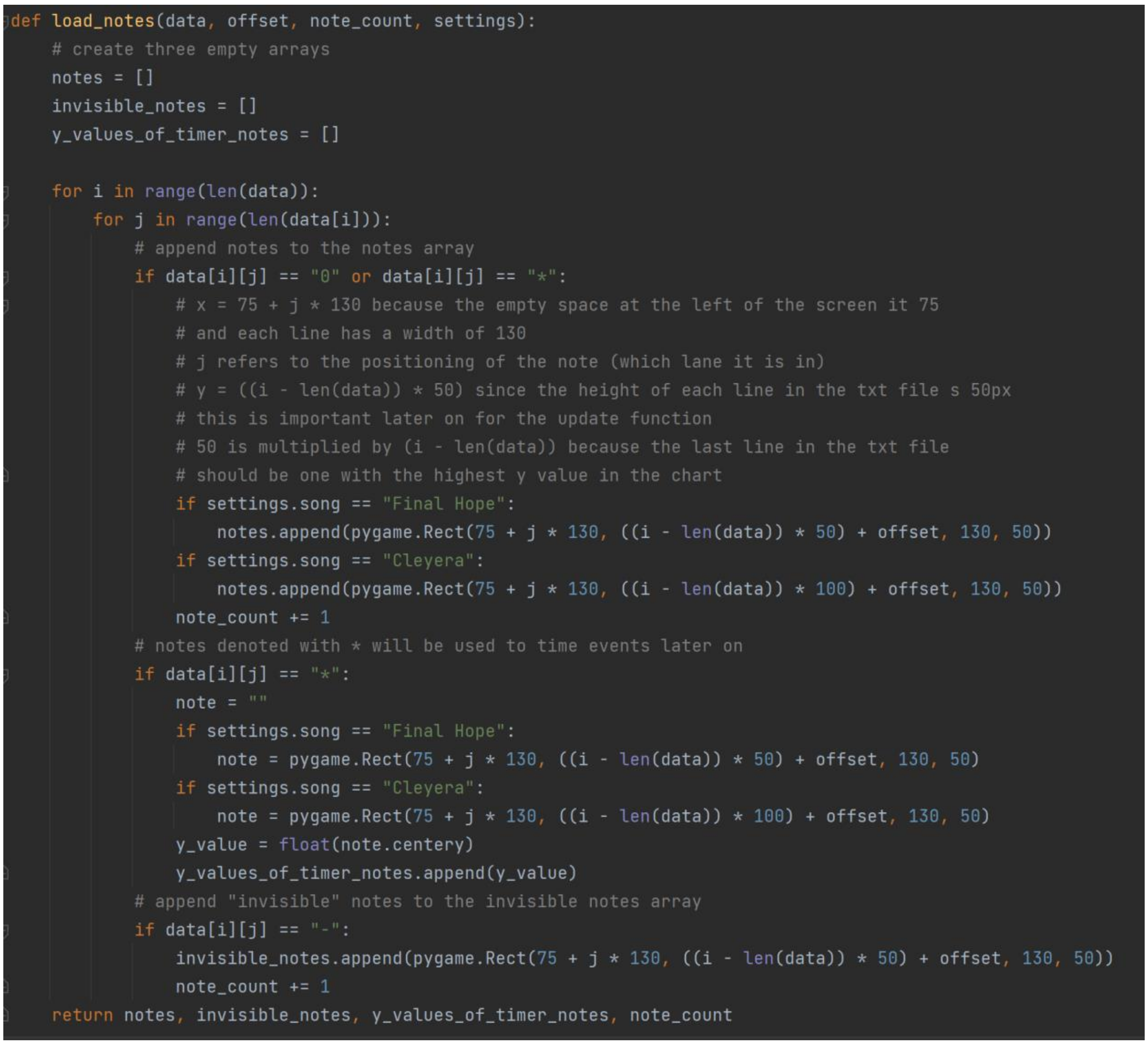
Just to quickly explain, when the game is run, one Settings class, Start class, Rules class, Choice class and Collision class are immediately defined. The results class is defined only after the player has finished playing the song. The Distractions and RememberThePattern class only get defined if the player chooses to play "Final Hope". Otherwise, it isn't.

5. Modules

The only module I used here is pygame. It is a set of python modules to facilitate the making of video games.

6. Essential algorithms

This section is going to be a long one. I'm going to start with the algorithm that brings the chart to pygame. To chart the song, I used a text file. In the text file, I used "0" to symbolize notes, "*" to symbolize notes that help me time events, and "-" to symbolize invisible notes for Final Hope. Before I explain it, I'll show a screenshot of the code.



```python
def load_notes(data, offset, note_count, settings):
    # create three empty arrays
    notes = []
    invisible_notes = []
    y_values_of_timer_notes = []

    for i in range(len(data)):
        for j in range(len(data[i])):
            # append notes to the notes array
            if data[i][j] == "0" or data[i][j] == "*":
                # x = 75 + j * 130 because the empty space at the left of the screen it 75
                # and each line has a width of 130
                # j refers to the positioning of the note (which lane it is in)
                # y = ((i - len(data)) * 50) since the height of each line in the txt file s 50px
                # this is important later on for the update function
                # 50 is multiplied by (i - len(data)) because the last line in the txt file
                # should be one with the highest y value in the chart
                if settings.song == "Final Hope":
                    notes.append(pygame.Rect(75 + j * 130, ((i - len(data)) * 50) + offset, 130, 50))
                if settings.song == "Cleyera":
                    notes.append(pygame.Rect(75 + j * 130, ((i - len(data)) * 100) + offset, 130, 50))
                note_count += 1
            # notes denoted with * will be used to time events later on
            if data[i][j] == "*":
                note = ""
                if settings.song == "Final Hope":
                    note = pygame.Rect(75 + j * 130, ((i - len(data)) * 50) + offset, 130, 50)
                if settings.song == "Cleyera":
                    note = pygame.Rect(75 + j * 130, ((i - len(data)) * 100) + offset, 130, 50)
                y_value = float(note.centery)
                y_values_of_timer_notes.append(y_value)
            # append "invisible" notes to the invisible notes array
            if data[i][j] == "-":
                invisible_notes.append(pygame.Rect(75 + j * 130, ((i - len(data)) * 50) + offset, 130, 50))
                note_count += 1
    return notes, invisible_notes, y_values_of_timer_notes, note_count
```

Technically the comments already explain it, but I'll do it again. The data array is an array of lines in the text file and each of which is an array for the characters in that line. The 1st for loop loops through each line, and the 2nd for loop loops through the characters in each line. This is important to figure out the lane placement of the notes. There is 75px of empty space on the left, and each lane has a width for 130px. This is why the x position is 75 + index * 130. Let's just take the Final Hope chart as the example. The y position is a bit trickier: ((index – len(data)) * 50) + offset. The height of each line is 50px. This number is important because for the next essential algorithm. That aside, in the txt file, the notes that are supposed to show up first are at the last line. That is because during charting, I obviously had to follow along from the start of the song, not backwards. For that reason, I had to subtract the number of lines in the entire text file from the index to appropriately place each note.

However, there is one more thing: the + offset part. All rhythm games have this problem, and I put it there since nearly every device / output device will end up having different offsets. For my laptop, assuming I'm not using my wireless earbuds, this offset is 900. I put a self.offset variable in the settings class so that the player can change this value when the notes do not sync on their device. This is done the same way for Cleyera, but instead with a value of 100px per line. Once again, there is a reason for this that will be explained soon.

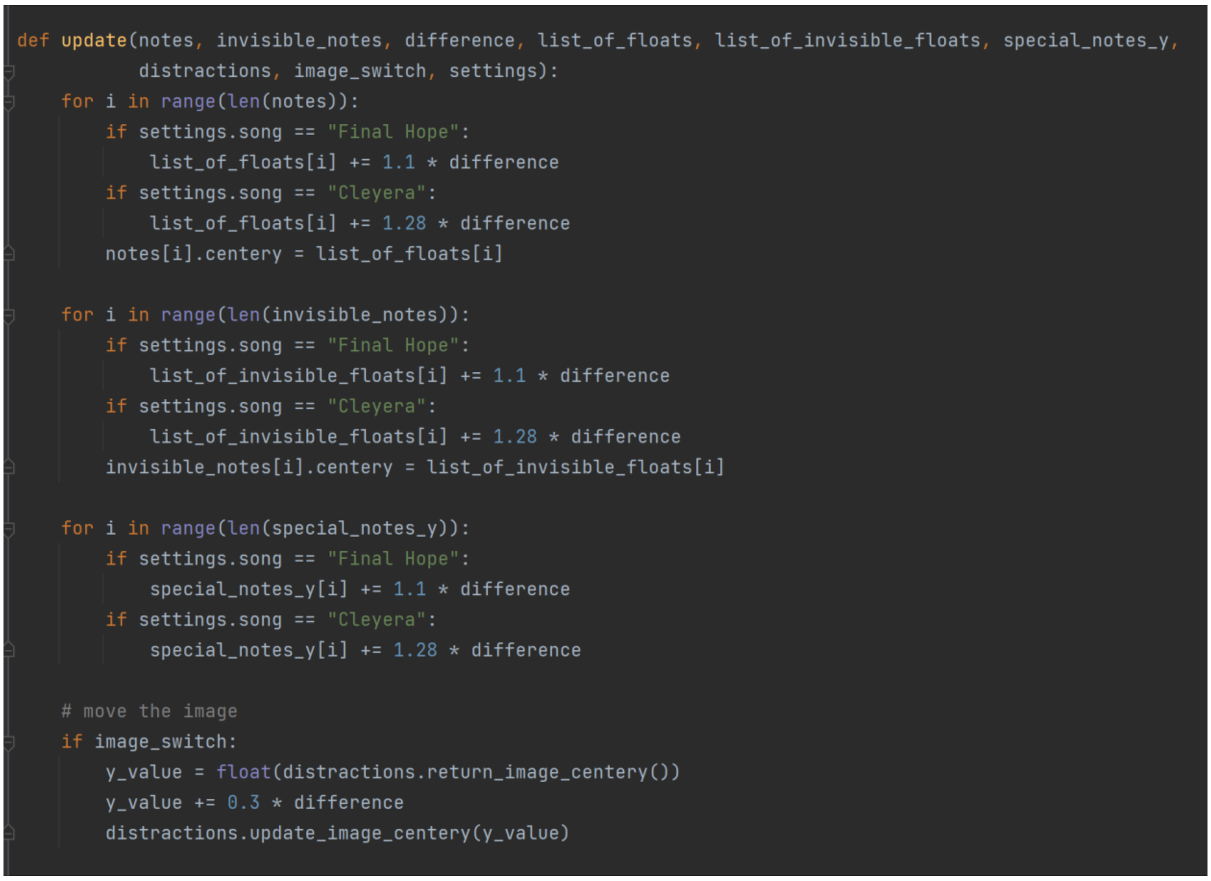
The next part of the screenshot addresses the notes symbolized by "*". In Final Hope, these are used to show text, distractions and end the game. In Cleyera, this is used only to end the game. I put the centery values of these special notes in their own array, so that it will be easier to check later on. You may notice that the centery values are stored as floats. This is also done for the array of notes and invisible notes, but it's quite self-explanatory and is not shown here. All it does is update the values as floats and then replace the centery values with those floats, similar to the alien invasion forum we had a while ago.

The last part just repeats the process for the invisible notes. These are separated since they will be drawn with a different color. Simple enough. Also, I reused this function to also return the note count for the results instead of counting all of it myself and putting in a static number on the results screen.

Okay next algorithm. This is the part that stumped me the most because of my approach. It's regarding how the notes fall. I will use Final Hope to discuss this, as once I figured it out, Cleyera didn't have any problems. Final Hope has a bpm of 220. This means that there are 220 beats within 60 seconds. As a musician, I focused on these beats and their length, only to realize that it forms a recurring decimal. Each beat has a length of 3/11s. Final Hope has quavers (notes with a length of half a beat) and quaver triplets (notes with a length of 1/3 of a beat). Because of this, I made it so that 6 lines in the text file is equal to 1 beat. What does this mean? This means that each line in the text file should have the length of 1/22s, or 0.454545s. At this point, I set the height of each line to be 60px. I couldn't figure out a way to make it work. Each note has to move by 60px every 1/22s. I tried to implement the same logic from the alien invasion project, changing the speed again and again to see if one of them would work. However, no matter the speed, nothing was ever perfectly in sync. I eventually realized why.

I've been using pygame.time.Clock to set up frame rate. At that time, I thought that if I set it to, let's say 500, then the frame rate will be 500 all the time. However, after printing out the frame rate, I realized that was wrong. It keeps it around 500, but it changes all the time and is inconsistent. This is fine for the alien invasion forum we had, but it doesn't work for a song with perfectly consistent bpm. For a couple days, I thought about ways to make sure that each line in the txt file lasts for 1/22s, or for each beat to last for 3/11s, but the recurring decimals screwed it up no matter what, and the inconsistency of time between each loop screws it up even more.

After a few days, I finally figured out the solution. Here's the code.



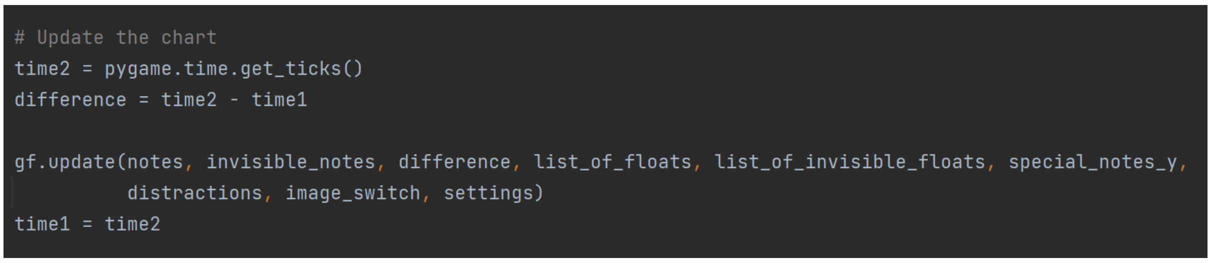
```python
def update(notes, invisible_notes, difference, list_of_floats, list_of_invisible_floats, special_notes_y,
           distractions, image_switch, settings):
    for i in range(len(notes)):
        if settings.song == "Final Hope":
            list_of_floats[i] += 1.1 * difference
        if settings.song == "Cleyera":
            list_of_floats[i] += 1.28 * difference
        notes[i].centery = list_of_floats[i]

    for i in range(len(invisible_notes)):
        if settings.song == "Final Hope":
            list_of_invisible_floats[i] += 1.1 * difference
        if settings.song == "Cleyera":
            list_of_invisible_floats[i] += 1.28 * difference
        invisible_notes[i].centery = list_of_invisible_floats[i]

    for i in range(len(special_notes_y)):
        if settings.song == "Final Hope":
            special_notes_y[i] += 1.1 * difference
        if settings.song == "Cleyera":
            special_notes_y[i] += 1.28 * difference

    # move the image
    if image_switch:
        y_value = float(distractions.return_image_centery())
        y_value += 0.3 * difference
        distractions.update_image_centery(y_value)
```

There should be comments on this part of the code, but I temporarily removed it for the screenshot so it fits on the screen. Here's the main point of this solution. Instead of looking at the length of each beat or line in seconds, I reversed it. Instead, I looked at the beats or lines per second. Let me quickly explain the thought process.

Once again, this song has a bpm of 220 and 6 lines per beat, so each line lasts 1/22s. I changed the height of each line to 50, meaning that everything should move by 50px every 1/22s. This means that everything should move by 1 pixel every 1/1100 of a second. Finally, this means that everything should move by 1.1px every millisecond. With this, I formed a general formula to calculate the amount of px that everything should move per millisecond:

$$\frac{1}{\left(\frac{60 * 1000}{bpm * beats\ per\ line * pixels\ per\ line}\right)}$$

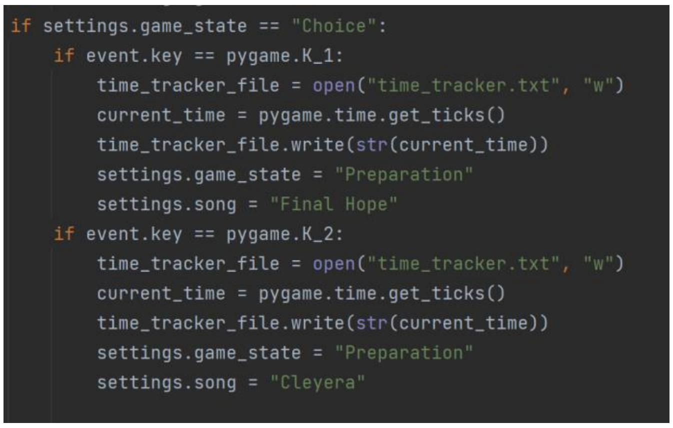
This is how I got the 1.28 that you see in the screenshot as well. Cleyera has 192bpm with 4 beats per line and 100px per line. Put those numbers in the formula, and you get 1.28px/ms. Now, what about the "difference' variable? As you may have guessed, that is the amount of milliseconds. Since pygame can't run the loop exactly the same every single time, the amount of milliseconds between any two loops are most likely different. So I added two variables for time and used this:



```
# Update the chart
time2 = pygame.time.get_ticks()
difference = time2 - time1

gf.update(notes, invisible_notes, difference, list_of_floats, list_of_invisible_floats, special_notes_y,
          distractions, image_switch, settings)
time1 = time2
```

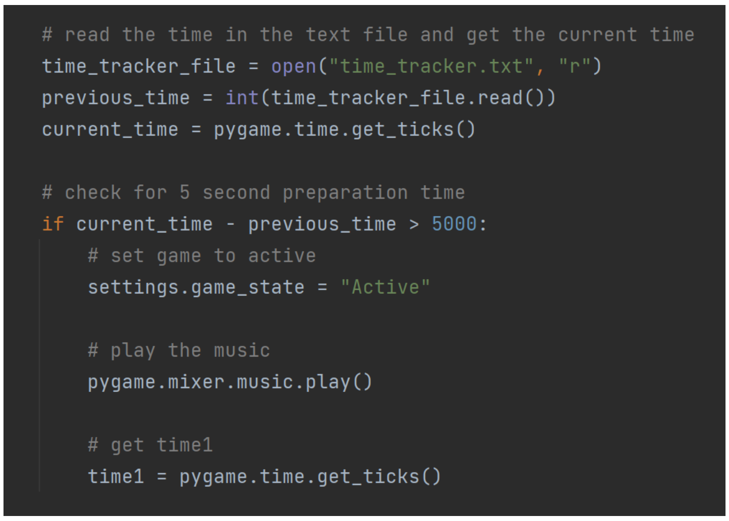
Time1 contains the value of time when the previous loop was carried out. Time2 then gets the value of time for this loop. Then, the difference between the two are calculated and each note is updated by 1.1 or 1.28 multiplied by the difference. Finally, the value of time2 is put into time1 for the next loop. However, there is one last thing that I had to consider that's related to time.

After the player chooses the song, there is 5 seconds of preparation time for them to get ready, as well as for me to briefly show the song title, composer and bpm. I had to measure these 5 seconds somehow, and the first solution I thought about was by using a file. Here's the start of it:



```
if settings.game_state == "Choice":
    if event.key == pygame.K_1:
        time_tracker_file = open("time_tracker.txt", "w")
        current_time = pygame.time.get_ticks()
        time_tracker_file.write(str(current_time))
        settings.game_state = "Preparation"
        settings.song = "Final Hope"
    if event.key == pygame.K_2:
        time_tracker_file = open("time_tracker.txt", "w")
        current_time = pygame.time.get_ticks()
        time_tracker_file.write(str(current_time))
        settings.game_state = "Preparation"
        settings.song = "Cleyera"
```

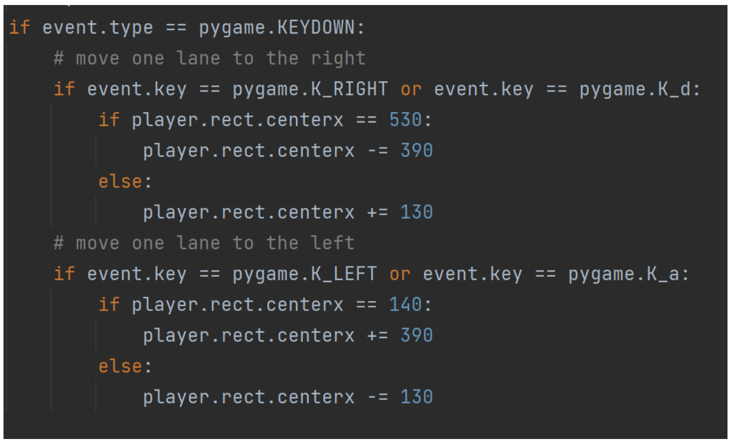
This is a part of the function that checks for key events when the song isn't playing. As seen in the screenshot, I open a file called "time_tracker.txt" in writing mode, then write down the time when the user picks the song before setting the game to "Preparation", which lasts for five seconds. Here is what I put to check that:



```
# read the time in the text file and get the current time
time_tracker_file = open("time_tracker.txt", "r")
previous_time = int(time_tracker_file.read())
current_time = pygame.time.get_ticks()

# check for 5 second preparation time
if current_time - previous_time > 5000:
    # set game to active
    settings.game_state = "Active"

    # play the music
    pygame.mixer.music.play()

    # get time1
    time1 = pygame.time.get_ticks()
```

I open the txt file again in read mode. I then store the value in the txt file into the previous_time variable and check for the current time. Afterwards, I check for their difference. If it's over 5 seconds (I can't put == 5000 since the frames are inconsistent and it is quite likely that it won't ever be exactly 5000), then I set the game to active and start the main game, play the music, and define time1 to start off the note updates.

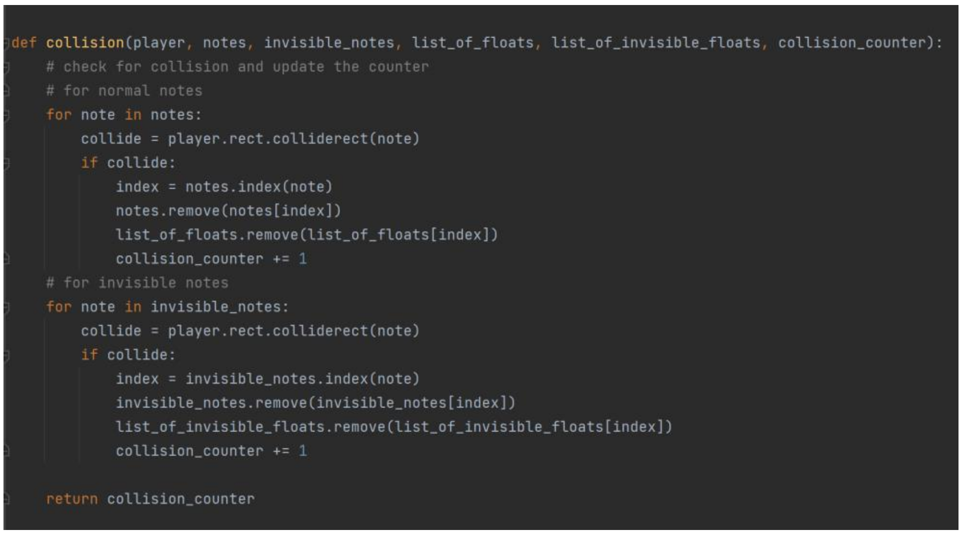
The next 2 pieces I want to show aren't exactly complex or anything, but are definitely essential to the game. Firstly, this is the code that allows the player to move:



```python
if event.type == pygame.KEYDOWN:
    # move one lane to the right
    if event.key == pygame.K_RIGHT or event.key == pygame.K_d:
        if player.rect.centerx == 530:
            player.rect.centerx -= 390
        else:
            player.rect.centerx += 130
    # move one lane to the left
    if event.key == pygame.K_LEFT or event.key == pygame.K_a:
        if player.rect.centerx == 140:
            player.rect.centerx += 390
        else:
            player.rect.centerx -= 130
```

Pressing the right arrow or "d" sends the player one lane to the right, but when the player is at the fourth lane (which means the value of centerx is 530), they get sent to the first lane (-390 is the same as going three lanes left). Pressing the left arrow or "a" sends the player one lane to the left, but when the player is at the first lane (which means the value of centerx is 140), they get send to the first lane (+390 is the same as going three lanes right). I initially allowed players to use two different set of inputs for comfort, as some people are used to WASD, but I ended up using this to my advantage for the second song. Having two sets of inputs allowed me to chart a couple sections that require fast movement in one direction, basically forcing the player to use both sets of inputs.

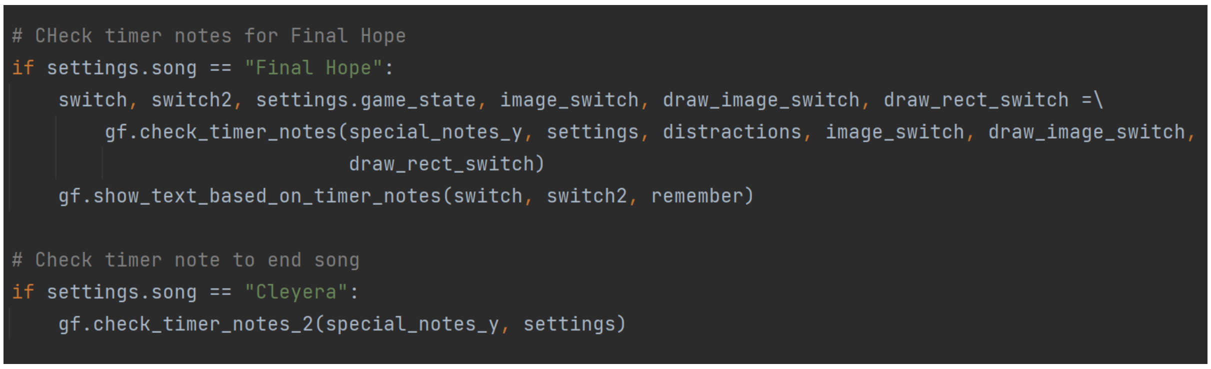
Secondly, this is the code that checks for collision:



```python
def collision(player, notes, invisible_notes, list_of_floats, list_of_invisible_floats, collision_counter):
    # check for collision and update the counter
    # for normal notes
    for note in notes:
        collide = player.rect.colliderect(note)
        if collide:
            index = notes.index(note)
            notes.remove(notes[index])
            list_of_floats.remove(list_of_floats[index])
            collision_counter += 1
    # for invisible notes
    for note in invisible_notes:
        collide = player.rect.colliderect(note)
        if collide:
            index = invisible_notes.index(note)
            invisible_notes.remove(invisible_notes[index])
            list_of_invisible_floats.remove(list_of_invisible_floats[index])
            collision_counter += 1

    return collision_counter
```

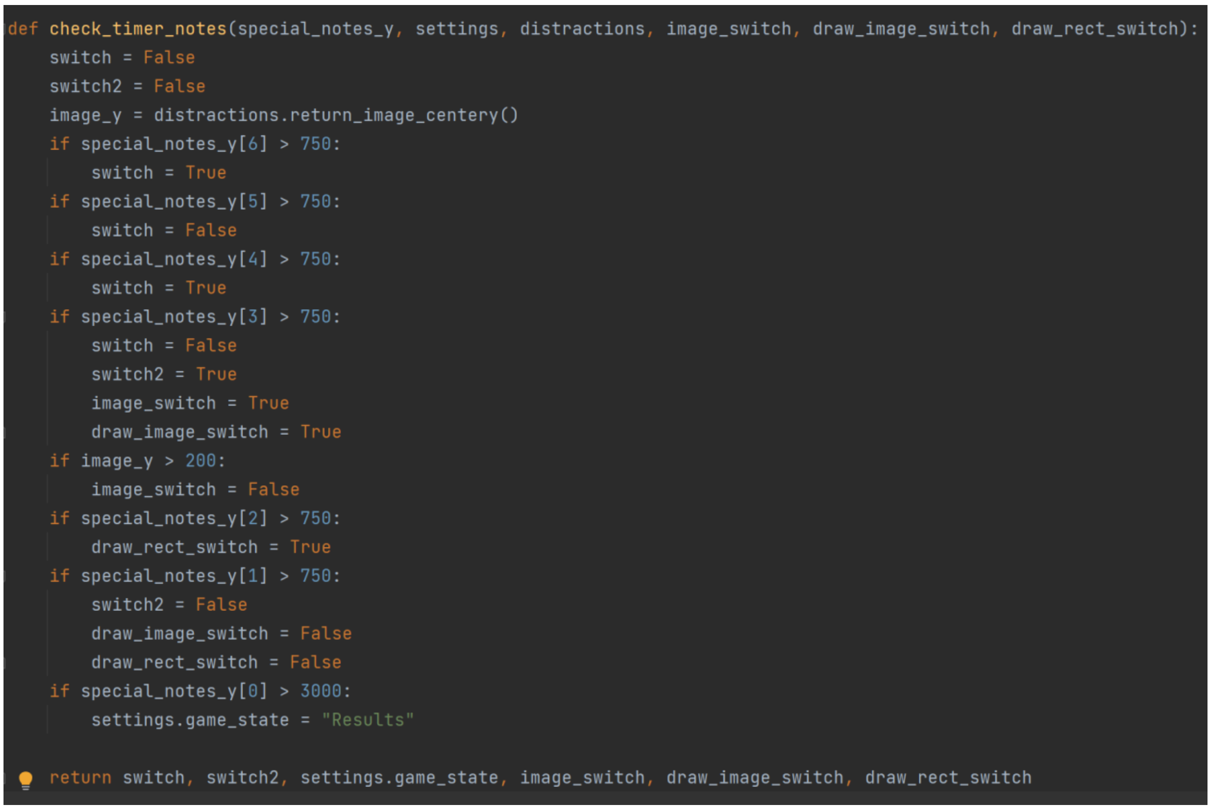
I used the colliderect function from to check for collision between the player and any of the notes. When a collision happens, I get the index of the note and remove the note from the notes array and the list_of_floats array. The reason for this is simple. If I don't remove these notes, then it will detect a collision for every loop while the player is still inside the note's rect. To avoid that, touching the notes immediately deletes them and adds 1 to the collision counter. This collision counter will obviously be displayed, but also used to display the results later on.

The last piece of code that I want to show in this section is this:



```python
# CHeck timer notes for Final Hope
if settings.song == "Final Hope":
    switch, switch2, settings.game_state, image_switch, draw_image_switch, draw_rect_switch =\
        gf.check_timer_notes(special_notes_y, settings, distractions, image_switch, draw_image_switch,
                             draw_rect_switch)
    gf.show_text_based_on_timer_notes(switch, switch2, remember)

# Check timer note to end song
if settings.song == "Cleyera":
    gf.check_timer_notes_2(special_notes_y, settings)
```

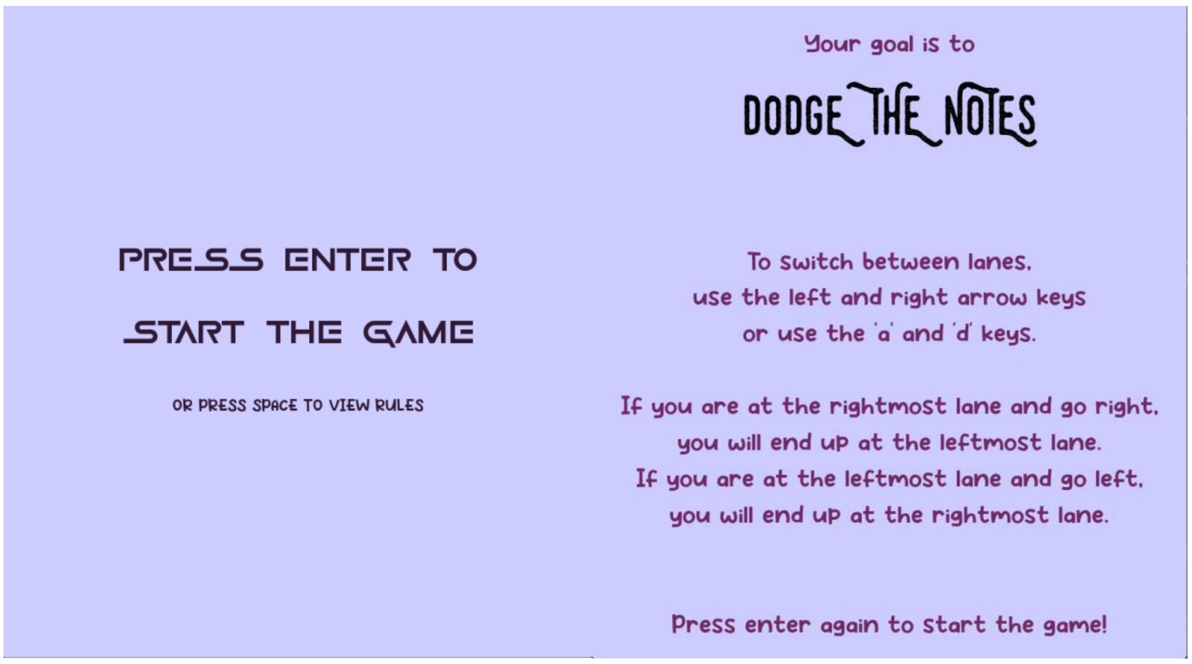
The function for Cleyera seems pretty tame, as once again, there are no distractions. The only timer note in the entire chart is the last note, which is why the comment says it's used to end the song. However, the one for final hope looks quite intense with 5 switches. Here's the function being called:



```python
def check_timer_notes(special_notes_y, settings, distractions, image_switch, draw_image_switch, draw_rect_switch):
    switch = False
    switch2 = False
    image_y = distractions.return_image_centery()
    if special_notes_y[6] > 750:
        switch = True
    if special_notes_y[5] > 750:
        switch = False
    if special_notes_y[4] > 750:
        switch = True
    if special_notes_y[3] > 750:
        switch = False
        switch2 = True
        image_switch = True
        draw_image_switch = True
    if image_y > 200:
        image_switch = False
    if special_notes_y[2] > 750:
        draw_rect_switch = True
    if special_notes_y[1] > 750:
        switch2 = False
        draw_image_switch = False
        draw_rect_switch = False
    if special_notes_y[0] > 3000:
        settings.game_state = "Results"

    return switch, switch2, settings.game_state, image_switch, draw_image_switch, draw_rect_switch
```

There should be comments everywhere, but I temporarily got rid of them so that the function can physically fit on the screen. I'll quickly explain what each of these does. The first if statement about special_notes_y[6] checks for the first hardcore (just to refresh, hardcore is a subgenre of electronic music) section of the song (which is where the pattern that needs to be memorized is first shown). This changed switch into True, whish shows "Remember this pattern!". The second if statement (special_notes_y[5]) checks for the end of that first hardcore section, which turns the switch back off to no longer show the text. The third if statement (special_notes_y[4]) checks for the main hardcore section of the song, turning the switch back on to give a second chance for the player to remember the pattern. The fourth if statement (special_notes_y[3]) does a couple of things. It turns the switch off, so the text no longer shows. It then turns switch2 on, which shows the text "Keep the pattern, no matter what!". It then turns the image switch and the draw image switch on. Here is why there are two switches. The image switch makes it so that the image of the hand goes down from above the top of the screen. This is done in the update function shown a few pages ago if you were paying attention. When the image is already at a certain value of y, it stops moving. However, it should still appear on the screen until the end of the hardcore section, so the fifth if statement turns only the image switch off. The sixth if statement (special_notes_y[2]) turns the draw rect switch on. This switch toggles a small rectangle the size of the player's icon and the same color as the background. When it is on, the rectangle is drawn above the player, causing the player's icon to be invisible. The seventh if statement (special_notes_y[1]) checks for the end of the main hardcore section. It turns switch2 off to no longer show the text, draw image switch off to no longer show the hand, and draw rect switch off to no longer hide the player. The final if statement, like the one for Cleyera, ends the game when the last note has passed, showing the result.

7. Screenshots


PRESS ENTER TO

START THE GAME

OR PRESS SPACE TO VIEW RULES

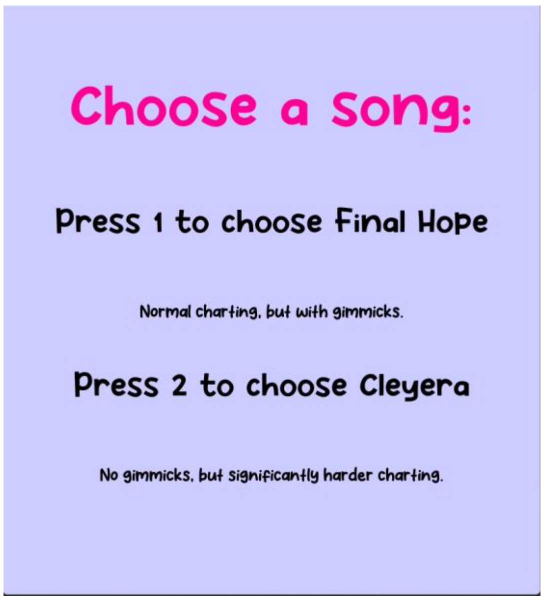
Your goal is to

DODGE THE NOTES

To switch between lanes,
use the left and right arrow keys
or use the 'a' and 'd' keys.

If you are at the rightmost lane and go right,
you will end up at the leftmost lane.
If you are at the leftmost lane and go left,
you will end up at the rightmost lane.

Press enter again to start the game!



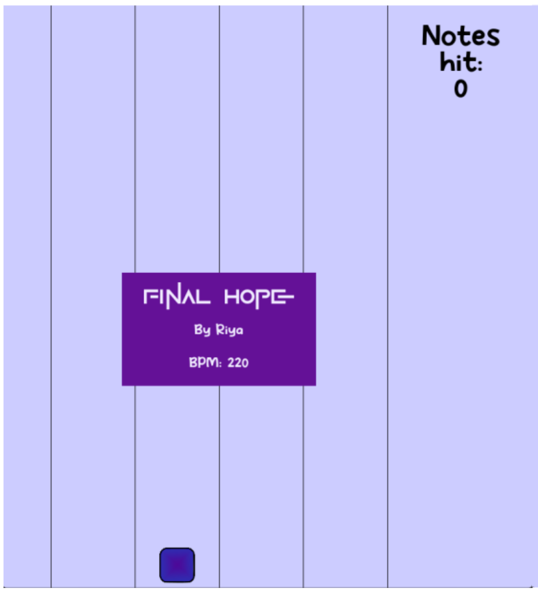
Choose a song:

Press 1 to choose Final Hope

Normal charting, but with gimmicks.

Press 2 to choose Cleyera

No gimmicks, but significantly harder charting.

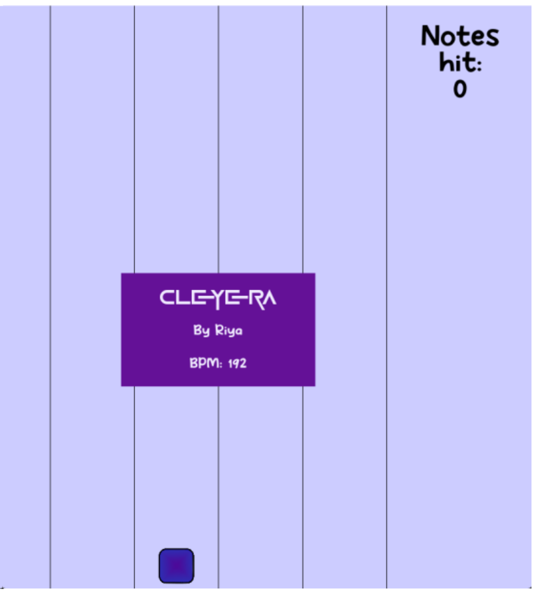


Notes hit: 0

FINAL HOPE

By Riya

BPM: 220

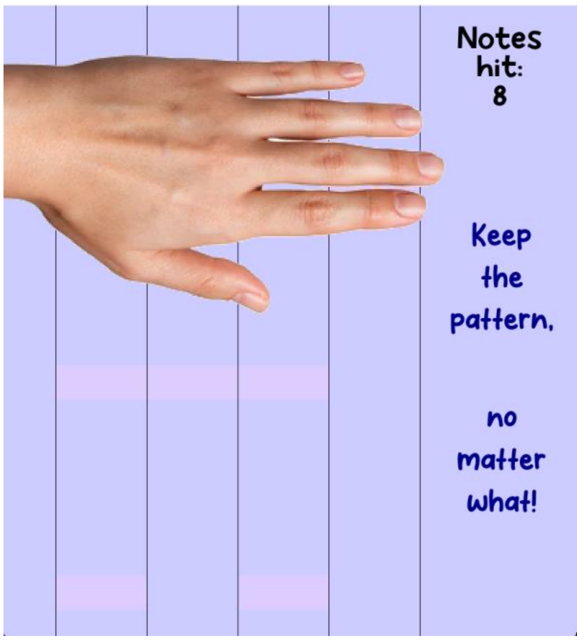


Notes hit: 0

CLEYERA

By Riya

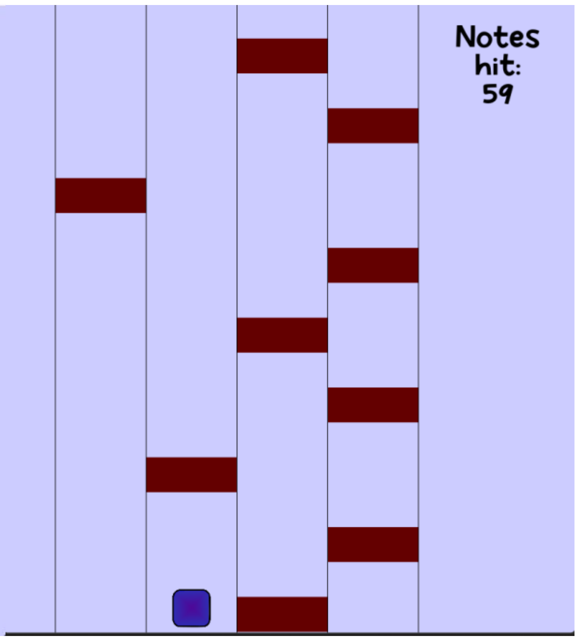
BPM: 192



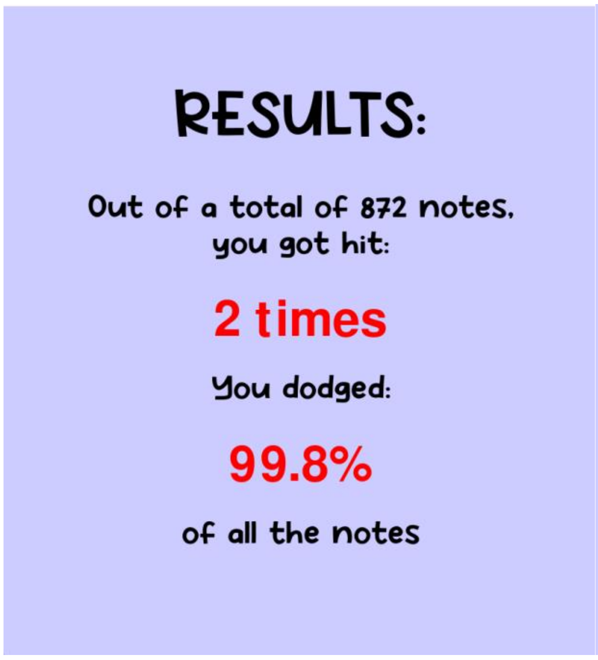
Notes hit: 8

Keep the pattern,

no matter what!



Notes hit: 59


RESULTS:

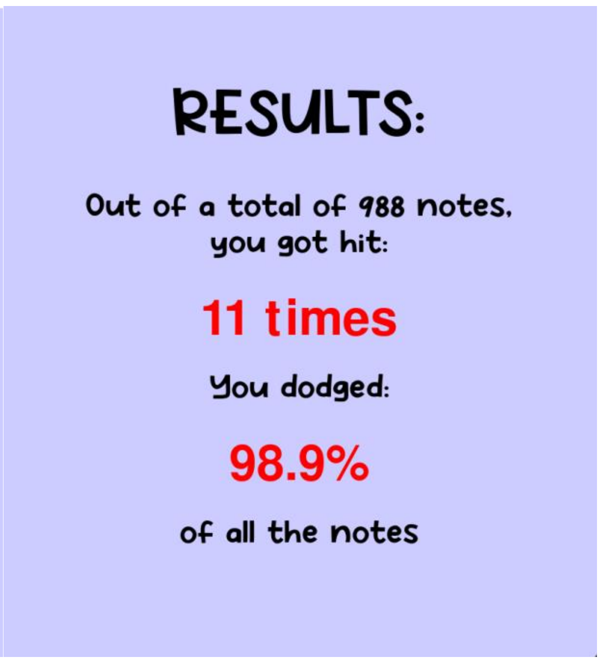
Out of a total of 872 notes, you got hit:

2 times

You dodged:

99.8%

of all the notes


RESULTS:

Out of a total of 988 notes, you got hit:

11 times

You dodged:

98.9%

of all the notes

8. Lesson learned / Reflection

Personally, this is by far my favorite project in this semester. In contrary to the other projects, I feel that I've learnt a lot more from this one. From a technical perspective, I learnt quite a lot about pygame, and also experienced the process of creating a game like this from scratch. From creating the window, to drawing the lines, then the player, then implementing movement, then charting the song and putting it in, then making the notes move, then implementing collision, adding the distractions and text, creating other screens to make it feel more polished, etc. It does feel quite surreal, as at the start, I had absolutely no idea how to pull it off. I just had the idea. One important lesson I learnt is that by taking it step by step, solving one problem after another, and adding new things one at a time, something that first seemed unfathomably difficult becomes a lot more intuitive.

Thinking back, this is also the only project where I never really felt pressured. I was extremely stressed about Pancasila and HCI, partially because of my teammates, but also because those just didn't feel enjoyable to do at all. However, the free topic choice on this one allowed me to pick music, which I'm very passionate about. Working with beats, rhythm game charts and music felt infinitely more fun than dealing with css, javascript, or the five precepts of Pancasila. I've known this for quite some time, but I've never really done big projects like this, so it's also a lesson for me to learn, now that I know just how much changes when I enjoy the project. That's why I made the second song too, simply because it was enjoyable.

One last important thing that I've learnt is that how much more satisfying it is to do something by myself. Since I finished the first song before the break ended, I went back to campus with a finished game. A bunch of people were discussing about this project, so I went ahead and took a look. Most of the others' projects weren't done, but when I saw the code, there were things I didn't really understand. An example of this is the usage of sprite sheets. For some time, a part of me felt slightly disappointed, thinking that based on what they've made so far, their code seems to be more complicated than mine. However, as the day progressed and this class started, I started hearing very similar sentences from many people. It is the fact that many people followed tutorials or copied a bunch of code, then modified it.

I guess that did help me become a lot prouder with this game, as I know for a fact that I thought of the idea myself, and even though I searched google for some syntaxes (for example, how to play music), I ultimately made the entire code alone. I went through each problem and tried solving them, some that I figured out quickly and some that stumped me for days. I charted the songs alone, put them to the .txt file alone, typed out every function, experimented with a bunch of stuff, tried again and again to fix stuff that just wouldn't want to work, and now, I end up with a functioning game. Perhaps that's why people say it's all about the journey. Regardless of this game's complexity, I've come to terms that a game that I made and completely understand in and out is something to be proud of, too.

~End~