PvP Fighting Game with Pygame Module

Final Project Report (Algorithm & Programming)

Lyonel Judson Saputra (2802505853)

L1BC

Jude Joseph Lamug Martinez MCS

Binus International
University Jakarta

1. Abstract

This report is the documentation and discussion of a student's Algorithm and Programming class final project in the form of a PvP game. The game is a 2 player fighting game in a 1 vs 1 mode that is made by using the Pygame module. It uses python coding to get player inputs in order to allow them to play against each other in one device, by moving, using attacks, and managing each character's health and knockbacks.

2. Introduction

The purpose of this project is so that students can make their own project by using python coding in order to solve a certain problem. They are to apply what they learn in class and what they learn beyond class to solve the problem. The form of the project can be a game, an app, or a project which pushes students to learn more than what is taught inside of the class. Students are to show their mastery of the python algorithms, programming concepts, and problem solving skills by creating a python solution as a project.

3. Project Inspiration

Creating this PvP game is inspired by a pc/console game called brawlhalla, specifically the 1 v 1 mode of the game. It is a 2 player game that has multiple skills to attack the opponents and have a quite simple movement mechanic. Unlike other PvP games, brawlhalla has the main focus on pushing the opponents off the map in order to win it. Therefore, the skills and attacks in this game are also adjusted to give knockback effects other than just the damage. The movement mechanic also supports multiple jumping and dashing in order to avoid falling off the map. Experiencing these unique mechanics of the game when playing it for the first time on console with my friends makes me want to try and create it with pygame for this project. I thought that instead of making the usual PvP game, I would try and remake this game instead. The fun I had when playing this game and the unique mechanics of the game made me decide to remake this game with pygame for this project.

4. Project Specifications

This game consists of several mechanics such as movement and attack that are meant to be quite close to the game mechanics of the game Brawlhalla. This game gets the user input in order to make the character move and attack according to the player's input. It also sets the health loss of each player and the knockback received based on the attack of the other player. It keeps track of the health of each player by using a health bar. Furthermore, it checks the collision of the players with the map and also applies gravity which makes the characters fall if they do not collide with the ground or platform in the map. This game also provides 3 different attacks which have different amounts of knockback and damage. These game mechanics along with the smooth animation of each character creates an engaging 2D fighting game.

- **Framework and modules used for this project:**
    - **Pygame:** This module is used to get the player inputs, animate the characters, handle the attack logics, and also import the game environment along with the sounds.

    - **Python:** The programming language that is used to program all the logic and mechanics of the game
- **This game has several components**
    - Main game loop
    - Character class
    - Game environment
    - HUD
    - User input
- **The Characters in this game have these Attributes:**

    - **Movement**
        - Player 1 (left):
            - Left: A
            - Right: D
            - Jump: W
            - Down: S
            - Dash/Sprint: V
        - Player 2 (right):
            - Left: Left Arrow
            - Right: Right Arrow
            - Jump: Up Arrow
            - Down: Down Arrow
            - Dash/Sprint: M
    - **Attack**
        - Player 1 (left):
            - Attack 1: T
            - Attack 2: G
            - Ultimate: F
        - Player 2 (right):
            - Attack 1: I
            - Attack 2: K
            - Ultimate: L

    - **Character Animation**
        - Each action performed by each character is animated.
    - **HUD**
        - Health bar
        - Sprint bar
        - Skill bar
- **Game Environment:**

    - **Collisions**
        - Ground
        - Platform
    - **Camera**
        - Follows the middle of the two players
- **Winning Conditions**
    - The score is 3 or the difference is 2
        - Score is got if
            - Enemy health is 0
            - Goes out of frame either top, bottom, right, left
- **Challenges:**
    - Animation Handling
        - The animation part of the code is the most challenging part of this entire project. Since every single action needs to be animated, it requires a lot of effort to find the right sprite sheets and to cut each of them because there are so many actions that need to be animated. I also needed to edit some of the sprites to fit the actions that are in this project.
- **Future Improvements:**
    - More features
        - Weapons
            - In the actual brawlhalla game, there are weapons that drop randomly and can be used for the player to gain advantage. However, in this project, I was unable to create that feature in time, therefore, it can be something that can be improved on in the future.
        - Character selection
            - There are only 2 characters in this game and it is fixed by default. If this game were to be improved, a character selection feature would be one of the things that can be added to improve the overall game experience.
    - More players (online)
        - Playing with more than 2 players would also be a fun feature to add. However, since it is not possible for 4 players to play on the same pc/laptop, creating an online feature would be a great improvement.
    - More Maps

- There is only one default map in this game, and more maps could be added to add more player customizations to the game which could make them enjoy the game more.

5. Solution Discussion

This game initialises by running the main.py which also imports all the necessary modules to run the game. First of all, the code imports the Pygame module which is used to create the game and also initialises it by using pygame.init() function. It then also imports all the remaining python files of the game like character.py, hud.py, background.py, collision.py which will all be explained more later on. On the main.py, after initializing pygame, the next code sets the screen width and height, and also sets the fps of the game. Then, it loads the background image of the game which is currently the only map for this game. It then starts the main game loop, and sets the key to be pressed to end the loop.

In the game loop, firstly, it calls the update function of the character class which calls all the necessary functions to update each character's attributes throughout the loop. Then, it sets the offset values which will be passed to the background and the characters. Then, it will draw the background that is already modified by the offset and the zoom by using the screen.blit method. Finally, the game loop will call to draw the HUD and the characters itself.

The main game loop also deals with the menu tab which consists of a winning menu and a pause menu. The winning menu will automatically be triggered when one of the players wins which will be explained more in the Character class. The winning menu will be triggered right after the player position is reset which gives it a smooth transition for the players after winning and not abruptly change to the winning menu. In the winning menu, there are 2 buttons that the players can pick, the play again button and the exit button. The play again button resets the game and runs the loop again while the exit button just exits the loop and closes the game. The pause menu will be opened when someone presses the escape button. In the pause menu, there is a resume button which continues the game loop and an exit button which exits the game. These menus are in the menu.py and are imported by the main.py.

The next python file is the collision.py which is simply the file which checks all the collisions of the map. It consists of a Collision class which takes the name of each collision rect in the __init__ function and creates the rect in the rect function which gets the x position, y position, width, and height for each of the rect. Then, all of the rect is defined by giving it the name, and all the arguments for the rect function. The rects like ground bottom, platform1, etc. will be passed into the Collision class which will create and return all the values of each rect so that it can be used later on to check whether the character is in contact with any of these rects.

Then, there is the background.py file which consists of 3 functions, the zoom function, camera function, and the center_background function. The zoom function gets the background image that is passed from the main game loop where it is called, and returns the background image so that it can be drawn on the screen. The second function is the camera function which deals with the offset of the camera where it should be in the middle of the two characters when they are moving. It gets the initial midpoint coordinates of the two players and subtracts it with the live average of the two x and y positions which gives the live coordinates of the midpoint of the two characters. This mechanism causes the camera to be able to stay in the middle of these two players and move dynamically with them. It also returns the x and y offset to the main game loop to be passed on to the other elements of the game. The third function is the center_background function which gets the zoomed image and produces the centered zoomed image. This centered image will be then returned to the main game loop.

Now that the background and the collision is set up, let's discuss the longest and most important part of the entire code, the character.py file which contains the Character class. This file imports the collision.py file because it requires the checking of the collision between the player and the map. The first few lines of code just initializes the screen, and also the pygame mixer which will be used to generate the sound for the game. Then, this file starts to load all the needed elements like the audio, but also all the animation frames that are required to animate the actions of each character. There are 12 actions for each character that needs to be imported and each action has multiple frames varying from 4 frames to 14 frames. After all the animation frames are loaded, the Character class is defined and takes 5 arguments from the character1 and character2 variables that are to be defined later. In the __init__, it has a lot of instance variables which takes care of a lot of things the characters do like the movement, collision checking, attacking, animation, etc.

The first function of the Character class is a get function which is the get_rect_hitbox which returns the hitbox rect of the character. It along with the second function, which returns the range rect, will be used to detect collisions and detect the range of the character’s attacks. After that, there is the take_damage function which subtracts the current character health by the damage it took. It also calls the check_health function which is the next function in the class. The check_health function just checks whether the character has health or not and decides what to do in either situation. If the character's health is 0, then it will start the reset process, and set the animation logic to show the death animation along with playing the death sound effect. On the contrary, when the character’s health is above 0, then it will have the original character attributes and the player can continue playing like normal.

After that, there is the check_collision function which checks whether the character’s hitbox rect is colliding with the map such as ground, platforms, and

borders which is obtained from the collision file discussed before. First of all, this function checks the borders which are the top, bottom, left, and right borders of the screen, and if the character makes contact with it, their health will be set to 0. Since there are 4 borders, it uses the for loop which checks all three of them and if either of the three collides with the character hitbox rect. If any of them collides, then the character will automatically be reset and the other character will get a point. Then, it checks the ground and platforms which would stop the character from falling if colliding with it since there is a gravity mechanism in this game. Upon colliding with the ground or the platforms, the character will stop falling.

The next function is the check_movement function whose purpose is to decide which action's animation is prioritised, for example the attack animation has to override the walking animation for it to make sense when playing the game. All of them use initial variables that are triggered by each input or action the character performs and call another function that sets the current_action of the character which is directly linked to the animation. It also checks for the direction of the character to set the animation for the attack logic.

The movement function handles the user input for moving the characters. It has two controls, the left one, and the right one. The left one consists of wasd to move up, down, right, and left while the right one uses the arrow keys. The jumping mechanism makes use of the gravity mechanics in this game and adds the character's y position which makes it jump and the gravity mechanic pulls it back down. The left and right movement is just adding to the character's x position. The dash logic makes the jump and walk function much faster, and there is a limited time to use it. The movement function also calls the check_movement function that is mentioned before in order to set the animation of each movement. Lastly, it also calls on the collision function that is also discussed before to check the collision of the character and the collision elements.

The reset position function resets the character's position and health after they fall off the map or their health is 0. This function does not teleport the character to the original position, however, it gradually transports them by dividing the x and y distance from the character's current position and the target position by 30, and then incrementing the x and y position of the character by the value that is obtained. Then, it will stop once it gets close to the respawn position and reset all the player attributes so that the player can continue to play.

The next one is the check_hits function which checks whether a character is in range of the other and it does it by using the colliderect between the character's hitbox and the opponent's range rect. It then returns whether or not the character is in range. The next function is the player_direction function which just gets the player direction according to the movement function that was discussed before and uses the player direction to modify the size and position of the range rect. It modifies it for two

situations, which are melee and range, and in both of these situations, the range rect will be modified differently.

After that, there is the knockback function which sets the knockback amount and knockback direction when a character is hit by the other. It only sets the knockback direction if there is no previous knockback direction in order to prevent the knockback from changing directions. Then, it also sets the multipliers according to the attack that the player uses, and also the health loss of the character. If the health lost is more, then the knockback amount will also be more. After that, it checks the knockback direction that is set and changes the x or y position of the character accordingly. Then, it decelerates the knockback by dividing the knockback amount by a certain amount continuously until it stops. Once the knockback stops, all the movement and attack character attributes are returned to normal.

The attack function handles the attack based on the input of the player. It then sets the damage that will be dealt to the other character, and also the knockback amount after checking if the enemy is in range and if the counter is full already. The first attack is a basic attack that can be used whenever, but has low knockback and damage, and is also melee. The second attack has a higher damage and knockback, and is a ranged attack, which requires 3 times hit with the basic attack. This is done by setting the range to ‘range’ so that the range rect will be edited to support a ranged attack. The ultimate does the most knockback, but requires 8 times hit with the basic attack and is a melee attack. The attack function also sets the instance variables that are used in the check_movement function according to the attack that the character does. It then animates the current attack. It also plays the sound effect when pressing any attack.

The next function is the score function which adds the score based on the instance variable that is set by the other functions and determines the winning condition for each character. It just uses 2 if statements and some ‘and’ and ‘or’. The set_action statement sets the action of the character which will be used for the animation.

The update_animation function determines the current frame index from the animation frames to draw by the draw function and adds the frame index by a certain speed which determines how fast the animation is or how fast the frames change. It is the code that handles the logic for the animation of the character’s actions. The speed of the animation change for each character’s actions is also set within this function. It also sets certain actions like walk and idle on a loop by setting the frame index to 0 again after reaching the final frame index of that action. This function also handles the flipping of the image when the direction is left and right.

The draw function gets the current frame index from the update_animation function and draws the image corresponding to it. This function also further modify the offsets so that the animation images will fit to the character rects so that the

gameplay will be better and to make sure that the animation is in sync with the gameplay. Finally, there is the reset function which is called when one of the character's wins and they want to play again, and also the update function which calls every necessary function from the Character class. Overall, the Character class handles everything the character does and keeps track of all the character attributes.

The character1 and character2 is then defined to pass all the arguments that are inside of the Character class. It also uses a dictionary to pass the animation argument for the Character class. It also helps to decide which animation is referred to in the update animation and set action function.

The hud.py file handles all the HUD elements of the game such as the health bar, skill bar, etc. These are linked to the Character class because the HUD elements get their values from the character attributes. The hud.py file contains the draw_hud function which is responsible for getting the character attributes and setting each hud element like the health bar. The first few lines on this function sets the fonts and the bar colors for each hud element for the first character. It also sets the size for the health bar like the height and width of it. Then, it sets the max arc, thickness, and radius for the skill arc. After everything is set, the function then draws the rect of the health bar by using all the set items and also giving the x and y position of the health bar. The health bar itself has 2 bars, one is the background bar which is slightly bigger and has a constant size, while the other one is the bar that shows the actual amount of health the character has. The health bar also writes the number of the character's current health.

Besides the health bar, the draw_hud function also draws the arc for the second attack and the ultimate. It also has 2 circles, one for the background and one for the actual skill bar. The main skill arc divides the attack 2 counter by 3 because it takes 3 hits to be full and divides the ultimate counter by 8 because it takes 8 hits to be full. So, it fills ⅓ of the arc for the second attack and fills ⅛ of the arc for the ultimate. These skill arcs show when the special attacks are ready. The HUD function also shows the score of each player which it gets the data from the Character class. It also shows the sprint bar which tells the player how much sprint they have left. The draw_hud function also removes all the mentioned HUD items and shows a winning text when one of the players wins, which goes hand in hand with the code I discussed in the main game loop.

6. Evidence

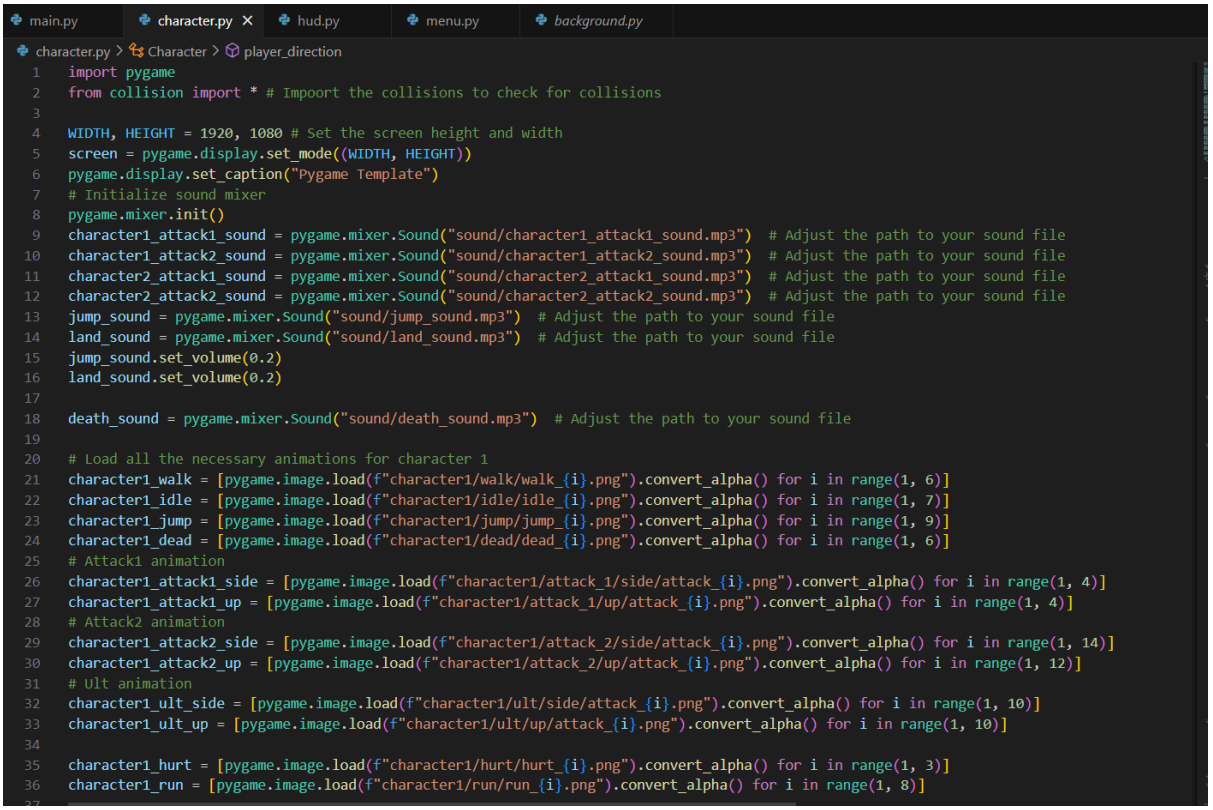
Image 1: main.py: Importing Pygame and running the main loop

main.py | character.py | hud.py | menu.py | background.py

main.py > ...

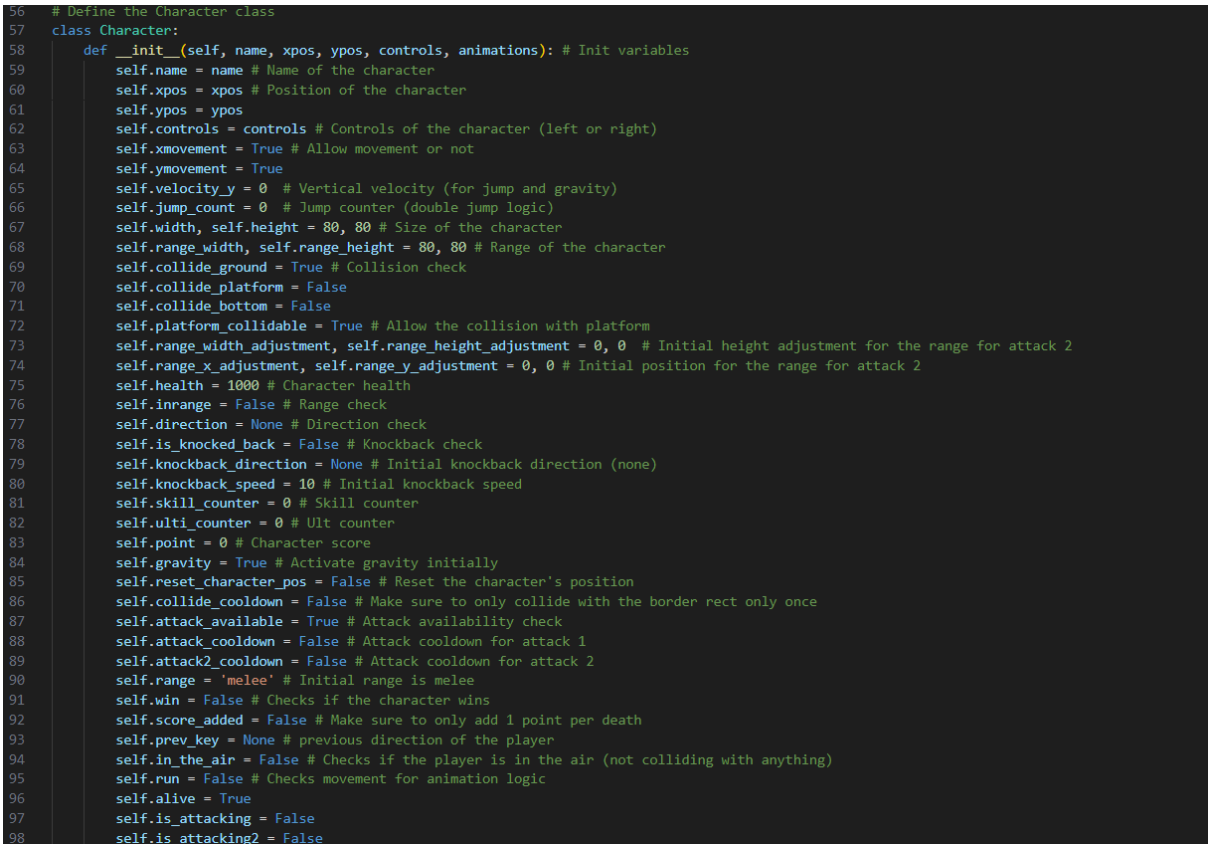
```python
import pygame
import sys
from character import *
from hud import *
from background import *
from collision import *
from menu import show_menu, pause_menu

# Initialize Pygame
pygame.init()

# Screen dimensions
WIDTH, HEIGHT = 1920, 1080
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Pygame Template")

# Frame rate
FPS = 120
clock = pygame.time.Clock()

# Load background image
background_image = pygame.image.load("map.webp").convert_alpha()

# Main game loop
if __name__ == '__main__':
    running = True
    while running:
        # Reset game
        character1.reset()
        character2.reset()

        while True:  # Inner game loop
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False
                    break
                elif event.type == pygame.KEYDOWN:
```

Image 2: character.py: Importing and Loading all the animation and audio



```python
import pygame
from collision import * # Impoort the collisions to check for collisions

WIDTH, HEIGHT = 1920, 1080 # Set the screen height and width
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Pygame Template")
# Initialize sound mixer
pygame.mixer.init()
character1_attack1_sound = pygame.mixer.Sound("sound/character1_attack1_sound.mp3")  # Adjust the path to your sound file
character1_attack2_sound = pygame.mixer.Sound("sound/character1_attack2_sound.mp3")  # Adjust the path to your sound file
character2_attack1_sound = pygame.mixer.Sound("sound/character2_attack1_sound.mp3")  # Adjust the path to your sound file
character2_attack2_sound = pygame.mixer.Sound("sound/character2_attack2_sound.mp3")  # Adjust the path to your sound file
jump_sound = pygame.mixer.Sound("sound/jump_sound.mp3")  # Adjust the path to your sound file
land_sound = pygame.mixer.Sound("sound/land_sound.mp3")  # Adjust the path to your sound file
jump_sound.set_volume(0.2)
land_sound.set_volume(0.2)

death_sound = pygame.mixer.Sound("sound/death_sound.mp3")  # Adjust the path to your sound file

# Load all the necessary animations for character 1
character1_walk = [pygame.image.load(f"character1/walk/walk_{i}.png").convert_alpha() for i in range(1, 6)]
character1_idle = [pygame.image.load(f"character1/idle/idle_{i}.png").convert_alpha() for i in range(1, 7)]
character1_jump = [pygame.image.load(f"character1/jump/jump_{i}.png").convert_alpha() for i in range(1, 9)]
character1_dead = [pygame.image.load(f"character1/dead/dead_{i}.png").convert_alpha() for i in range(1, 6)]
# Attack1 animation
character1_attack1_side = [pygame.image.load(f"character1/attack_1/side/attack_{i}.png").convert_alpha() for i in range(1, 4)]
character1_attack1_up = [pygame.image.load(f"character1/attack_1/up/attack_{i}.png").convert_alpha() for i in range(1, 4)]
# Attack2 animation
character1_attack2_side = [pygame.image.load(f"character1/attack_2/side/attack_{i}.png").convert_alpha() for i in range(1, 14)]
character1_attack2_up = [pygame.image.load(f"character1/attack_2/up/attack_{i}.png").convert_alpha() for i in range(1, 12)]
# Ult animation
character1_ult_side = [pygame.image.load(f"character1/ult/side/attack_{i}.png").convert_alpha() for i in range(1, 10)]
character1_ult_up = [pygame.image.load(f"character1/ult/up/attack_{i}.png").convert_alpha() for i in range(1, 10)]

character1_hurt = [pygame.image.load(f"character1/hurt/hurt_{i}.png").convert_alpha() for i in range(1, 3)]
character1_run = [pygame.image.load(f"character1/run/run_{i}.png").convert_alpha() for i in range(1, 8)]
```

Image 3: character.py: Character class



```python
# Define the Character class
class Character:
    def __init__(self, name, xpos, ypos, controls, animations): # Init variables
        self.name = name # Name of the character
        self.xpos = xpos # Position of the character
        self.ypos = ypos
        self.controls = controls # Controls of the character (left or right)
        self.xmovement = True # Allow movement or not
        self.ymovement = True
        self.velocity_y = 0  # Vertical velocity (for jump and gravity)
        self.jump_count = 0  # Jump counter (double jump logic)
        self.width, self.height = 80, 80 # Size of the character
        self.range_width, self.range_height = 80, 80 # Range of the character
        self.collide_ground = True # Collision check
        self.collide_platform = False
        self.collide_bottom = False
        self.platform_collidable = True # Allow the collision with platform
        self.range_width_adjustment, self.range_height_adjustment = 0, 0  # Initial height adjustment for the range for attack 2
        self.range_x_adjustment, self.range_y_adjustment = 0, 0 # Initial position for the range for attack 2
        self.health = 1000 # Character health
        self.inrange = False # Range check
        self.direction = None # Direction check
        self.is_knocked_back = False # Knockback check
        self.knockback_direction = None # Initial knockback direction (none)
        self.knockback_speed = 10 # Initial knockback speed
        self.skill_counter = 0 # Skill counter
        self.ulti_counter = 0 # Ult counter
        self.point = 0 # Character score
        self.gravity = True # Activate gravity initially
        self.reset_character_pos = False # Reset the character's position
        self.collide_cooldown = False # Make sure to only collide with the border rect only once
        self.attack_available = True # Attack availability check
        self.attack_cooldown = False # Attack cooldown for attack 1
        self.attack2_cooldown = False # Attack cooldown for attack 2
        self.range = 'melee' # Initial range is melee
        self.win = False # Checks if the character wins
        self.score_added = False # Make sure to only add 1 point per death
        self.prev_key = None # previous direction of the player
        self.in_the_air = False # Checks if the player is in the air (not colliding with anything)
        self.run = False # Checks movement for animation logic
        self.alive = True
        self.is_attacking = False
        self.is_attacking2 = False
```

Image 5: character.py: Character rect and health

```python
    # Create the character's hitbox rect
    def get_rect_hitbox(self):
        return pygame.Rect(self.xpos + 30, self.ypos , self.width - 40, self.height)

    # Create the character's range rect
    def get_rect_range(self):
        return pygame.Rect(self.xpos + self.range_x_adjustment + 30, self.ypos - self.range_y_adjustment, self.range_width - self.range_width_adjustment - 40, self.range_height - self.range_height_adjustment
    # Sets the health of the character based on the damage taken
    def take_damage(self, damage):
        self.health = max(0, self.health - damage)  # Health can't go below 0
        self.check_health()
    # Checks the health of the character
    def check_health(self):
        if self.health == 0: # Checks if the health is 0
            self.alive = False # Adjusts the character attributes when losing a round
            self.set_action('dead')
            self.reset_character_pos = True
            self.ulti_counter = 0
            self.xmovement = False
            self.ymovement = False
            character1.attack_available, character2.attack_available = False, False
            self.in_the_air = False
            death_sound.play()

        elif self.health > 0: # Restore the character attributes when the health is more than 0
            self.alive = True
            self.set_action('idle')
            self.reset_character_pos = False
            self.xmovement = True
            self.ymovement = True
            character1.attack_available, character2.attack_available = True, True
    # Checks the character hitbox rect collision with the map
    def check_collision(self, ground, platforms, borders):
        for border in borders:
            if self.get_rect_hitbox().colliderect(border) and self.collide_cooldown == False:
                self.reset_character_pos = True
                self.collide_cooldown = True
                self.health = 0
                self.check_health()

        if self.get_rect_hitbox().colliderect(ground):
            self.ypos = ground.top - self.height
            self.velocity_y = 0
```

The git repository at "c:\Users\ASUS" has too many a
changes, only a subset of Git features will be enabled
Source: Git
OK

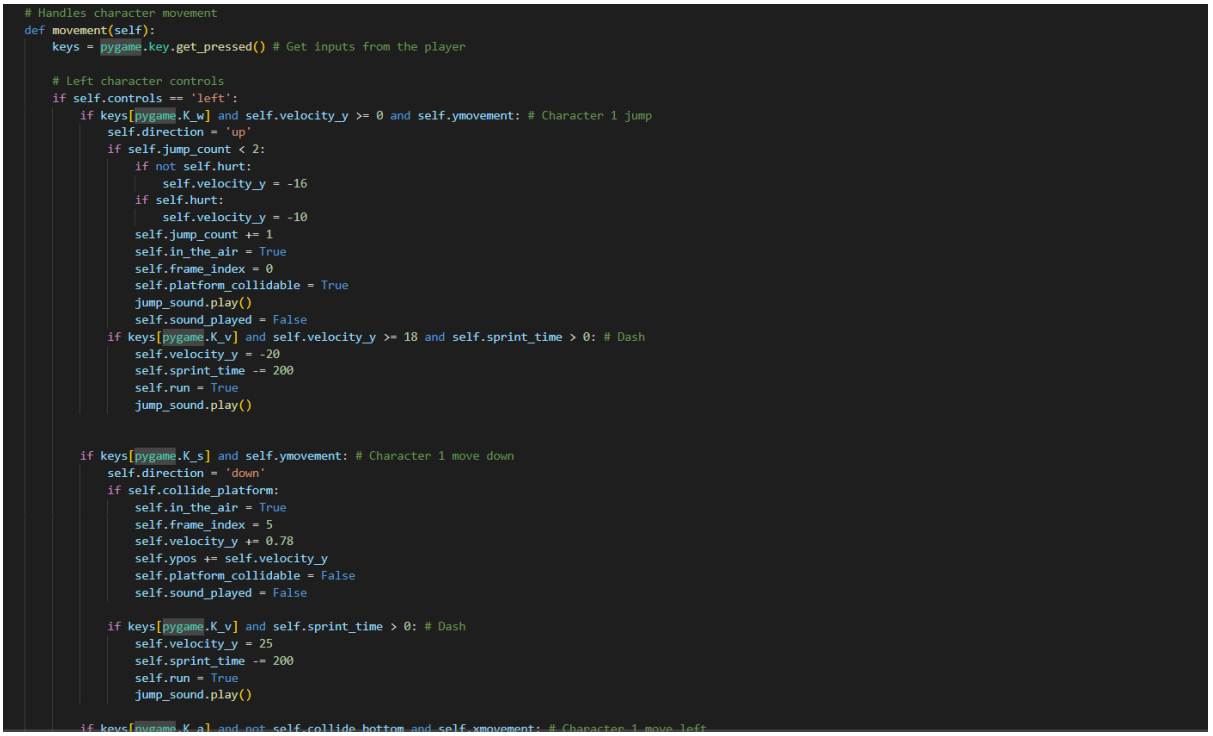
Image 6: character.py: Character collision

```python
    # Checks the character hitbox rect collision with the map
    def check_collision(self, ground, platforms, borders):
        for border in borders:
            if self.get_rect_hitbox().colliderect(border) and self.collide_cooldown == False:
                self.reset_character_pos = True
                self.collide_cooldown = True
                self.health = 0
                self.check_health()

        if self.get_rect_hitbox().colliderect(ground):
            self.ypos = ground.top - self.height
            self.velocity_y = 0
            self.jump_count = 0
            self.collide_ground = True
            self.collide_platform = False
            self.collide_bottom = False
            self.platform_collidable = True
            self.in_the_air = False
            if self.sound_played == False:
                land_sound.play()
                self.sound_played = True
        elif self.get_rect_hitbox().colliderect(bottom):
            self.collide_bottom = True
            self.collide_ground = False
            self.collide_platform = False
            self.in_the_air = False
            if self.sound_played == False:
                land_sound.play()
                self.sound_played = True
        else:
            for platform in platforms:
                if self.get_rect_hitbox().colliderect(platform):
                    self.collide_platform = True
                    self.collide_ground = False
                    self.collide_bottom = False
                    if self.velocity_y > 0 and self.platform_collidable:
                        self.ypos = platform.top - self.height
                        self.velocity_y = 0
                        self.jump_count = 0
                        self.platform_collidable = True
                        self.in_the_air = False
                        if self.sound_played == False:
                            land_sound.play()
                            self.sound_played = True
```

Image 7: character.py: Character current movement

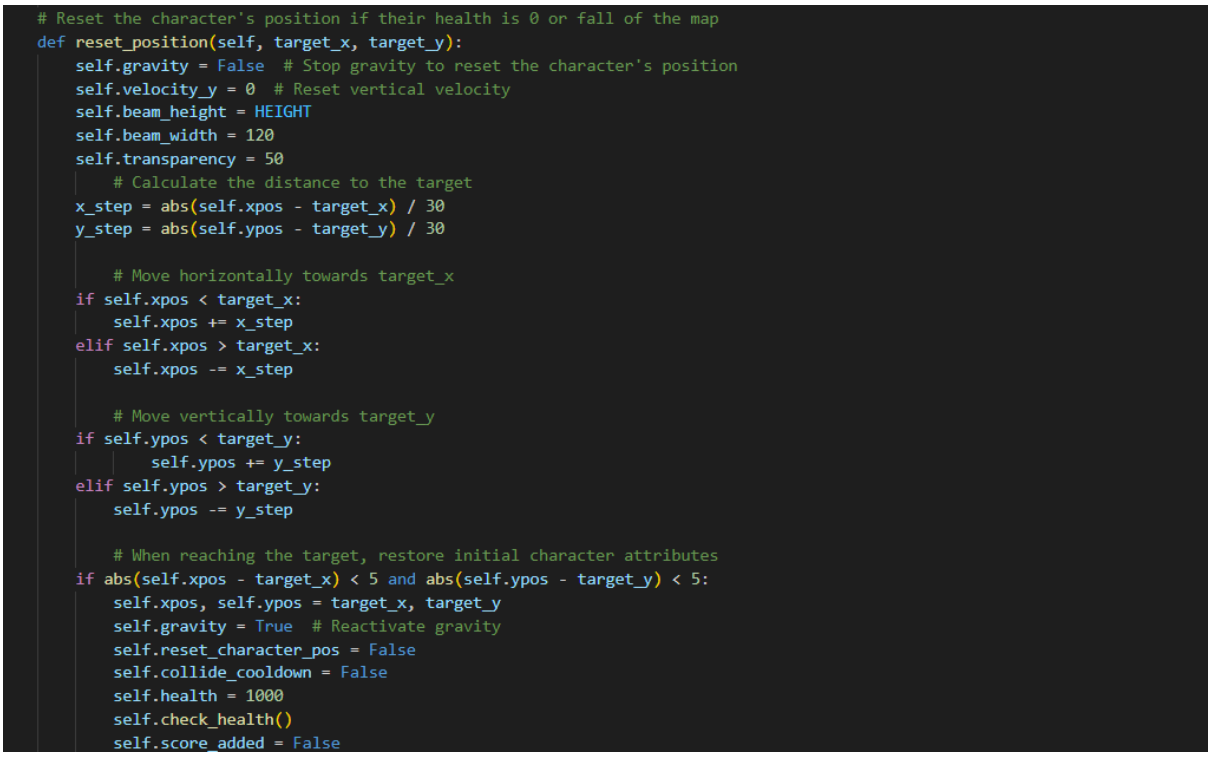
```python
                        break
    # For animation logic, checks which animation is prioritized
    def check_movement(self):
        if self.alive:
            if self.is_attacking:
                if self.direction == 'up':
                    self.set_action('attack1_up')
                elif self.direction == 'left' or 'right':
                    self.set_action('attack1_side')
            if self.is_attacking2:
                if self.direction == 'up':
                    self.set_action('attack2_up')
                elif self.direction == 'left' or 'right':
                    self.set_action('attack2_side')
            if self.is_ult:
                if self.direction == 'up':
                    self.set_action('ult_up')
                elif self.direction == 'left' or 'right':
                    self.set_action('ult_side')
            if self.hurt:
                self.set_action('hurt')
            if self.in_the_air and not self.is_attacking and not self.is_attacking2 and not self.is_ult and not self.hurt:
                self.set_action('jump')
            if self.run and not self.in_the_air and not self.is_attacking and not self.is_attacking2 and not self.is_ult and not self.hurt:
                self.set_action('run')
            if self.walking and not self.in_the_air and not self.is_attacking and not self.is_attacking2 and not self.is_ult and not self.hurt and not self.run:
                self.set_action('walk')
            if not self.walking and not self.in_the_air and not self.is_attacking and not self.is_attacking2 and not self.is_ult and not self.hurt and not self.run:
                self.set_action('idle')
```

Image 8: character.py: Character movement



```python
# Handles character movement
def movement(self):
    keys = pygame.key.get_pressed() # Get inputs from the player

    # Left character controls
    if self.controls == 'left':
        if keys[pygame.K_w] and self.velocity_y >= 0 and self.ymovement: # Character 1 jump
            self.direction = 'up'
            if self.jump_count < 2:
                if not self.hurt:
                    self.velocity_y = -16
                if self.hurt:
                    self.velocity_y = -10
                self.jump_count += 1
                self.in_the_air = True
                self.frame_index = 0
                self.platform_collidable = True
                jump_sound.play()
                self.sound_played = False
            if keys[pygame.K_v] and self.velocity_y >= 18 and self.sprint_time > 0: # Dash
                self.velocity_y = -20
                self.sprint_time -= 200
                self.run = True
                jump_sound.play()


        if keys[pygame.K_s] and self.ymovement: # Character 1 move down
            self.direction = 'down'
            if self.collide_platform:
                self.in_the_air = True
                self.frame_index = 5
                self.velocity_y += 0.78
                self.ypos += self.velocity_y
                self.platform_collidable = False
                self.sound_played = False

            if keys[pygame.K_v] and self.sprint_time > 0: # Dash
                self.velocity_y = 25
                self.sprint_time -= 200
                self.run = True
                jump_sound.play()

        if keys[pygame.K_a] and not self.collide_bottom and self.xmovement: # Character 1 move left
```

Image 9: character.py: Character reset position



```python
# Reset the character's position if their health is 0 or fall of the map
def reset_position(self, target_x, target_y):
    self.gravity = False  # Stop gravity to reset the character's position
    self.velocity_y = 0  # Reset vertical velocity
    self.beam_height = HEIGHT
    self.beam_width = 120
    self.transparency = 50
        # Calculate the distance to the target
    x_step = abs(self.xpos - target_x) / 30
    y_step = abs(self.ypos - target_y) / 30

        # Move horizontally towards target_x
    if self.xpos < target_x:
        self.xpos += x_step
    elif self.xpos > target_x:
        self.xpos -= x_step

        # Move vertically towards target_y
    if self.ypos < target_y:
            self.ypos += y_step
    elif self.ypos > target_y:
        self.ypos -= y_step

        # When reaching the target, restore initial character attributes
    if abs(self.xpos - target_x) < 5 and abs(self.ypos - target_y) < 5:
        self.xpos, self.ypos = target_x, target_y
        self.gravity = True  # Reactivate gravity
        self.reset_character_pos = False
        self.collide_cooldown = False
        self.health = 1000
        self.check_health()
        self.score_added = False
```

Image 10: character.py: Character check hits and direction

```
# Reset the character's position if their health is 0 or fall of the map
def reset_position(self, target_x, target_y):
    self.gravity = False  # Stop gravity to reset the character's position
    self.velocity_y = 0  # Reset vertical velocity
    self.beam_height = HEIGHT
    self.beam_width = 120
    self.transparency = 50
        # Calculate the distance to the target
    x_step = abs(self.xpos - target_x) / 30
    y_step = abs(self.ypos - target_y) / 30

        # Move horizontally towards target_x
    if self.xpos < target_x:
        self.xpos += x_step
    elif self.xpos > target_x:
        self.xpos -= x_step

        # Move vertically towards target_y
    if self.ypos < target_y:
            self.ypos += y_step
    elif self.ypos > target_y:
        self.ypos -= y_step

        # When reaching the target, restore initial character attributes
    if abs(self.xpos - target_x) < 5 and abs(self.ypos - target_y) < 5:
        self.xpos, self.ypos = target_x, target_y
        self.gravity = True  # Reactivate gravity
        self.reset_character_pos = False
        self.collide_cooldown = False
        self.health = 1000
        self.check_health()
        self.score_added = False
```

Image 11: character.py: Character knockback

```
# Reset the character's position if their health is 0 or fall of the map
def reset_position(self, target_x, target_y):
    self.gravity = False  # Stop gravity to reset the character's position
    self.velocity_y = 0  # Reset vertical velocity
    self.beam_height = HEIGHT
    self.beam_width = 120
    self.transparency = 50
        # Calculate the distance to the target
    x_step = abs(self.xpos - target_x) / 30
    y_step = abs(self.ypos - target_y) / 30

        # Move horizontally towards target_x
    if self.xpos < target_x:
        self.xpos += x_step
    elif self.xpos > target_x:
        self.xpos -= x_step

        # Move vertically towards target_y
    if self.ypos < target_y:
            self.ypos += y_step
    elif self.ypos > target_y:
        self.ypos -= y_step

        # When reaching the target, restore initial character attributes
    if abs(self.xpos - target_x) < 5 and abs(self.ypos - target_y) < 5:
        self.xpos, self.ypos = target_x, target_y
        self.gravity = True  # Reactivate gravity
        self.reset_character_pos = False
        self.collide_cooldown = False
        self.health = 1000
        self.check_health()
        self.score_added = False
```

Image 12: character.py: Character attack

```python
# Handles the character's attack
def attack(self, other):

    keys = pygame.key.get_pressed() # Get the user input
    multiplier = 1 # multiplier of knockback
    # Left character attack controls
    if self.controls == 'left':
        if keys[pygame.K_t] and self.attack_available and not self.attack_cooldown and not self.attack2_cooldown:
            self.attack_cooldown = True
            if other.inrange:  # If the other character is within range
                other.take_damage(50)
                other.inrange = False
                other.is_knocked_back = True
                self.skill_counter += 1
                self.ulti_counter += 1
                self.skill_counter = min(3, self.skill_counter)
                self.ulti_counter = min(8, self.ulti_counter)
                self.attack_available = False
                self.attack_cooldown = True
                self.is_attacking = True
                character1_attack1_sound.play()
        if self.attack_available:
            if self.frame_index > 2:
                self.is_attacking = False
        if not keys[pygame.K_t]:
            self.attack_cooldown = False

        if keys[pygame.K_g] and self.attack_available and not self.attack2_cooldown:
            if self.skill_counter == 3:
                self.attacking_range = True
                self.skill_counter = 0
                self.xmovement = False
                self.ymovement = False
                character1_attack2_sound.play()

        if self.attacking_range == True:
            self.range = 'range'
            self.is_attacking2 = True
            self.attack2_cooldown = True
            if self.frame_index > 6:
                if other.inrange:
                    other.take_damage(70)
                    other.inrange = False
                    multiplier = 5
```

Image 13: character.py: Character score and animation

```python
def score(self, other): # Sets the winning condition
    if self.alive == False and not self.score_added: # Adds the score to the other character if not alive
        other.point += 1
        self.score_added = True
    if (self.point - other.point == 2 and self.point > other.point) or (self.point >= 3 and other.point != self.point): # sets who wins the game, either 2 point difference or
        self.win = True
# Sets the action for the animation
def set_action(self, action):
    if self.current_action != action:
        self.frame_index = 0  # Reset to the first frame of the new action
    self.current_action = action
# Update the animation of the character
def update_animation(self, character):
    # Speed of the frames logic for each action
    if self.current_action == 'idle' or 'walk' or 'dead' or 'hurt':
        self.animation_speed = 0.01
    if self.current_action == 'run':
        self.animation_speed = 0.2
    if self.current_action == 'jump':
        if self.frame_index < 4:
            self.animation_speed = 0.7
        else:
            self.animation_speed = 0.05
    if character1.current_action == 'attack1_side' or 'attack1_up':
        self.animation_speed = 0.6
    if character1.current_action == 'attack2_side' or 'attack2_up':
        if character1.frame_index < 7:
            self.animation_speed = 0.6
        elif character1.frame_index >= 7:
            self.animation_speed = 0.1

    if character1.current_action == 'ult_side':
        if self.frame_index < 6:
            self.animation_speed = 0.6
        elif self.frame_index >= 6:
            self.animation_speed = 0.02
    if character1.current_action == 'ult_up':
        if self.frame_index < 7:
            self.animation_speed = 0.6
        elif self.frame_index >= 7:
            self.animation_speed = 0.05
    if character2.current_action == 'attack1_side' or 'attack1_up':
```

Image 14: character.py: Character draw

```
def score(self, other): # Sets the winning condition
    if self.alive == False and not self.score_added: # Adds the score to the other character if not alive
        other.point += 1
        self.score_added = True
    if (self.point - other.point == 2 and self.point > other.point) or (self.point >= 3 and other.point != self.point): # sets who wins the game, either 2 point difference or
        self.win = True
# Sets the action for the animation
def set_action(self, action):
    if self.current_action != action:
        self.frame_index = 0  # Reset to the first frame of the new action
    self.current_action = action
# Update the animation of the character
def update_animation(self, character):
    # Speed of the frames logic for each action
    if self.current_action == 'idle' or 'walk' or 'dead' or 'hurt':
        self.animation_speed = 0.01
    if self.current_action == 'run':
        self.animation_speed = 0.2
    if self.current_action == 'jump':
        if self.frame_index < 4:
            self.animation_speed = 0.7
        else:
            self.animation_speed = 0.05
    if character1.current_action == 'attack1_side' or 'attack1_up':
        self.animation_speed = 0.6
    if character1.current_action == 'attack2_side' or 'attack2_up':
        if character1.frame_index < 7:
            self.animation_speed = 0.6
        elif character1.frame_index >= 7:
            self.animation_speed = 0.1

    if character1.current_action == 'ult_side':
        if self.frame_index < 6:
            self.animation_speed = 0.6
        elif self.frame_index >= 6:
            self.animation_speed = 0.02
    if character1.current_action == 'ult_up':
        if self.frame_index < 7:
            self.animation_speed = 0.6
        elif self.frame_index >= 7:
            self.animation_speed = 0.05
    if character2.current_action == 'attack1_side' or 'attack1_up':
```

Image 15: character.py: Character reset and update

```
def score(self, other): # Sets the winning condition
    if self.alive == False and not self.score_added: # Adds the score to the other character if not alive
        other.point += 1
        self.score_added = True
    if (self.point - other.point == 2 and self.point > other.point) or (self.point >= 3 and other.point != self.point): # sets who wins the game, either 2 point difference or
        self.win = True
# Sets the action for the animation
def set_action(self, action):
    if self.current_action != action:
        self.frame_index = 0  # Reset to the first frame of the new action
    self.current_action = action
# Update the animation of the character
def update_animation(self, character):
    # Speed of the frames logic for each action
    if self.current_action == 'idle' or 'walk' or 'dead' or 'hurt':
        self.animation_speed = 0.01
    if self.current_action == 'run':
        self.animation_speed = 0.2
    if self.current_action == 'jump':
        if self.frame_index < 4:
            self.animation_speed = 0.7
        else:
            self.animation_speed = 0.05
    if character1.current_action == 'attack1_side' or 'attack1_up':
        self.animation_speed = 0.6
    if character1.current_action == 'attack2_side' or 'attack2_up':
        if character1.frame_index < 7:
            self.animation_speed = 0.6
        elif character1.frame_index >= 7:
            self.animation_speed = 0.1

    if character1.current_action == 'ult_side':
        if self.frame_index < 6:
            self.animation_speed = 0.6
        elif self.frame_index >= 6:
            self.animation_speed = 0.02
    if character1.current_action == 'ult_up':
        if self.frame_index < 7:
            self.animation_speed = 0.6
        elif self.frame_index >= 7:
            self.animation_speed = 0.05
    if character2.current_action == 'attack1_side' or 'attack1_up':
```

Image 16: character.py: Character 1 and 2

```python
# Initialize character 1
character1 = Character('character1', 1.2 * WIDTH // 3, HEIGHT // 2, 'left', animations={
        'walk': character1_walk, # List of frames for walking
        'idle': character1_idle,  # List of frames for idle
        'jump': character1_jump,  # List of frames for jumping
        'dead': character1_dead,  # List of frames for dead
        'attack1_side': character1_attack1_side,  # List of frames for attack1
        'attack1_up': character1_attack1_up,

        'attack2_side': character1_attack2_side,  # List of frames for attack2
        'attack2_up': character1_attack2_up,

        'ult_side': character1_ult_side,  # List of frames for ult
        'ult_up': character1_ult_up,

        'hurt': character1_hurt, # List of frames for hurt
        'run': character1_run # List of frames for run


    })
# Initialize character 2
character2 = Character('character2', 1.65 * WIDTH // 3, HEIGHT // 2, 'right', animations={
        'walk': character2_walk, # List of framse for wallking
        'idle': character2_idle,  # List of frames for idle
        'jump': character2_jump,  # List of frames for jumping
        'dead': character2_dead,  # List of frames for dead
        'attack1_side': character2_attack1_side,  # List of frames for attack1
        'attack1_up': character2_attack1_up,

        'attack2_side': character2_attack2_side,  # List of frames for attack2
        'attack2_up': character2_attack2_up,

        'ult_side': character2_ult_side,  # List of frames for ult
        'ult_up': character2_ult_up,

        'hurt': character2_hurt, # List of frames for hurt
        'run': character2_run # List of frames for run

    })
```

Image 17: hud.py: Drawing the HUD display for the characters

main.py | character.py | hud.py | menu.py | background.py

hud.py > draw_hud

```
def draw_hud(offset_x, offset_y):
        score_font = pygame.font.Font(None, 48)
        hud_font = pygame.font.Font(None, 36)

        # Define colors for HUD
        health_bar_color = (255, 0, 0)
        skill_bar_color = (0, 255, 0)
        background_bar_color = (50, 50, 50)
        # Bar size
        health_bar_width = 300
        bar_height = 40
        max_skill_arc = 360  # Max degrees for the skill

        # Character 1's attack2 circle properties
        skill_circle_radius_1 = 30  # Radius for Character 1's attack2 counter
        skill_circle_thickness_1 = 10  # Thickness for Character 1's attack2 circle
        skill_circle_color_1 = (211, 211, 211)


        ulti_circle_radius_1 = 50  # Radius for Character 1's skill counter
        ulti_circle_thickness_1 = 20  # Thickness for Character 1's skill circle

        # Character 2's attack2 skill circle properties
        skill_circle_radius_2 = 30  # Radius for Character 2's skill counter
        skill_circle_thickness_2 = 10  # Thickness for Character 2's skill circle
        skill_circle_color_2 = (211, 211, 211)


        ulti_circle_radius_2 = 50  # Radius for Character 1's skill counter
        ulti_circle_thickness_2 = 20  # Thickness for Character 1's skill circle

        # Draw Character 1's HUD
        # Score for character 1

        screen.blit(score_font.render(f"Score: {character1.point}", True, (0, 0, 255)), (WIDTH // 3 - 100, 120))
```

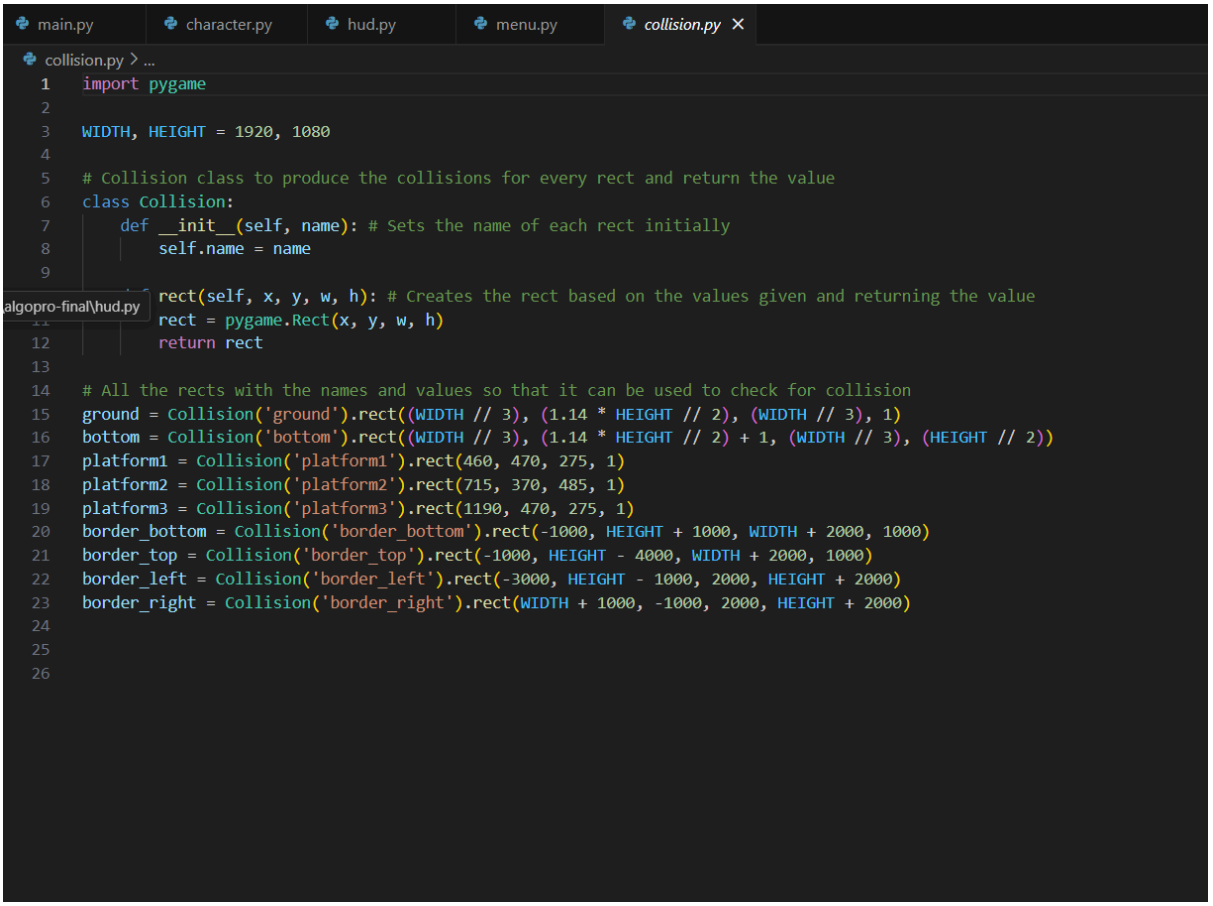
Image 18: menu.py: Showing the winning menu with play again and exit button

main.py | character.py | hud.py | menu.py | background.py

menu.py > show_menu

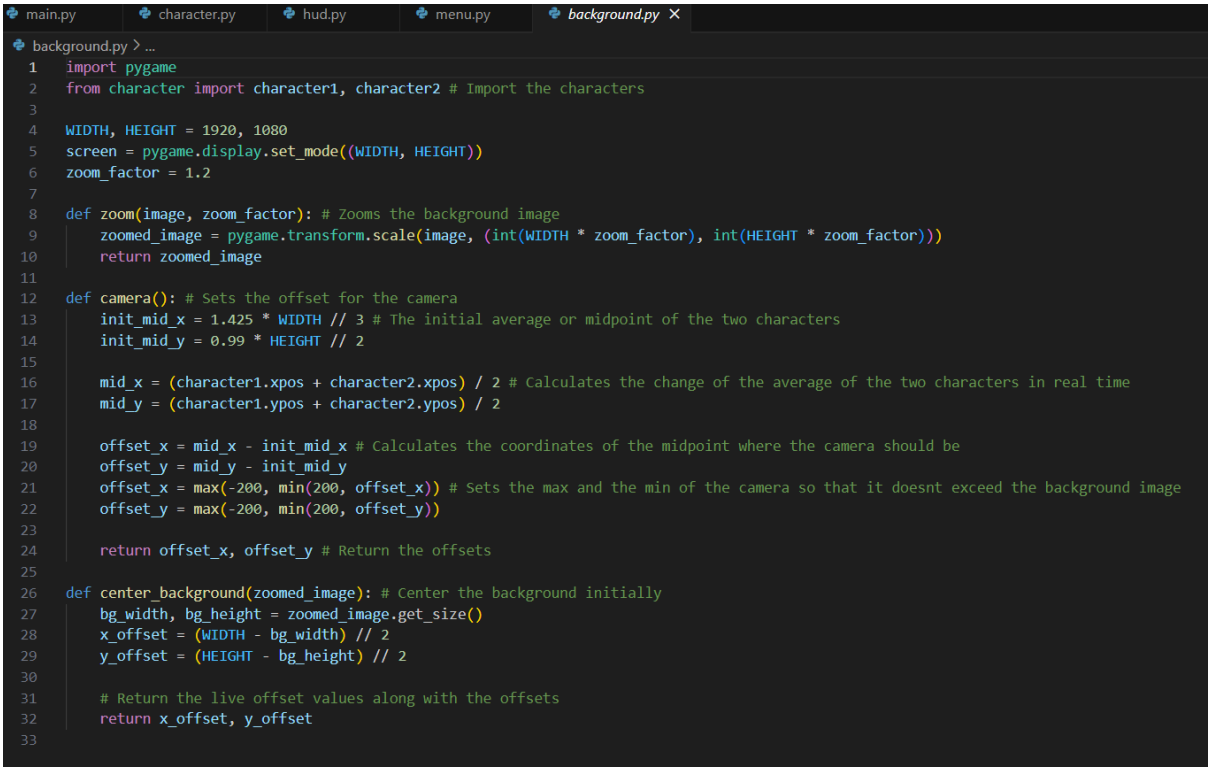
```
# Function to blur the background
def blur_surface(surface, amount):
    blur = pygame.transform.smoothscale(surface, (surface.get_width() // amount, surface.get_height() // amount))
    return pygame.transform.smoothscale(blur, surface.get_size())

# Menu function to the winner and points
def show_menu(screen, background_image, player1_points, player2_points, winner, WIDTH, HEIGHT):
    blurred_background = blur_surface(background_image, 10)
    winner_font = pygame.font.Font(None, 130)
    points_font = pygame.font.Font(None, 120)

    # Determine winner text color
    if winner == 1:
        winner_text = winner_font.render("Player 1 Wins!", True, (255, 0, 0))  # Red
    else:
        winner_text = winner_font.render("Player 2 Wins!", True, (0, 0, 255))  # Blue

    # Points text
    player1_points_text = points_font.render(f"{player1_points}", True, (255, 0, 0))  # Red
    player2_points_text = points_font.render(f"{player2_points}", True, (0, 0, 255))  # Blue

    # Button size
    button_width, button_height = 300, 80
    play_again_rect = pygame.Rect((WIDTH // 2 - button_width // 2, HEIGHT // 2), (button_width, button_height))
    exit_rect = pygame.Rect((WIDTH // 2 - button_width // 2, HEIGHT // 2 + 120), (button_width, button_height))

    while True:
        screen.blit(blurred_background, (0, 0)) # Draw the blurred background

        # Draw the winning texts
        screen.blit(player1_points_text, (WIDTH // 2 - winner_text.get_width() - player1_points_text.get_width() + 50, HEIGHT // 2 - 200))
        screen.blit(winner_text, (WIDTH // 2 - winner_text.get_width() // 2, HEIGHT // 2 - 200))  # winner text
        screen.blit(player2_points_text, (WIDTH // 2 + winner_text.get_width() - 50, HEIGHT // 2 - 200))  # Player 2 score

        # Draw buttons
        pygame.draw.rect(screen, (0, 128, 0), play_again_rect)  # Green play again button
        pygame.draw.rect(screen, (128, 0, 0), exit_rect)  # Red exit button
```

Image 19: collision.py: Defining all the collisions in the map



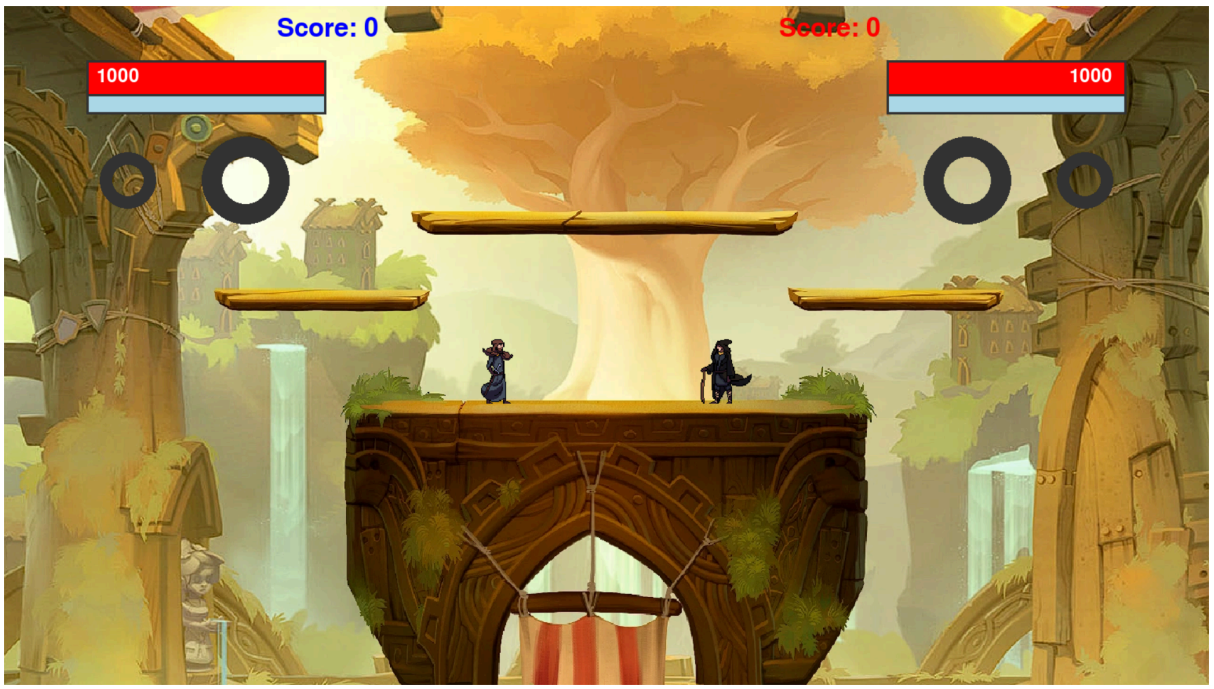
```python
import pygame

WIDTH, HEIGHT = 1920, 1080

# Collision class to produce the collisions for every rect and return the value
class Collision:
    def __init__(self, name): # Sets the name of each rect initially
        self.name = name

    def rect(self, x, y, w, h): # Creates the rect based on the values given and returning the value
        rect = pygame.Rect(x, y, w, h)
        return rect

# All the rects with the names and values so that it can be used to check for collision
ground = Collision('ground').rect((WIDTH // 3), (1.14 * HEIGHT // 2), (WIDTH // 3), 1)
bottom = Collision('bottom').rect((WIDTH // 3), (1.14 * HEIGHT // 2) + 1, (WIDTH // 3), (HEIGHT // 2))
platform1 = Collision('platform1').rect(460, 470, 275, 1)
platform2 = Collision('platform2').rect(715, 370, 485, 1)
platform3 = Collision('platform3').rect(1190, 470, 275, 1)
border_bottom = Collision('border_bottom').rect(-1000, HEIGHT + 1000, WIDTH + 2000, 1000)
border_top = Collision('border_top').rect(-1000, HEIGHT - 4000, WIDTH + 2000, 1000)
border_left = Collision('border_left').rect(-3000, HEIGHT - 1000, 2000, HEIGHT + 2000)
border_right = Collision('border_right').rect(WIDTH + 1000, -1000, 2000, HEIGHT + 2000)
```

Image 20: background.py: Setting the background and offsets



```python
import pygame
from character import character1, character2 # Import the characters

WIDTH, HEIGHT = 1920, 1080
screen = pygame.display.set_mode((WIDTH, HEIGHT))
zoom_factor = 1.2

def zoom(image, zoom_factor): # Zooms the background image
    zoomed_image = pygame.transform.scale(image, (int(WIDTH * zoom_factor), int(HEIGHT * zoom_factor)))
    return zoomed_image

def camera(): # Sets the offset for the camera
    init_mid_x = 1.425 * WIDTH // 3 # The initial average or midpoint of the two characters
    init_mid_y = 0.99 * HEIGHT // 2

    mid_x = (character1.xpos + character2.xpos) / 2 # Calculates the change of the average of the two characters in real time
    mid_y = (character1.ypos + character2.ypos) / 2

    offset_x = mid_x - init_mid_x # Calculates the coordinates of the midpoint where the camera should be
    offset_y = mid_y - init_mid_y
    offset_x = max(-200, min(200, offset_x)) # Sets the max and the min of the camera so that it doesnt exceed the background image
    offset_y = max(-200, min(200, offset_y))

    return offset_x, offset_y # Return the offsets

def center_background(zoomed_image): # Center the background initially
    bg_width, bg_height = zoomed_image.get_size()
    x_offset = (WIDTH - bg_width) // 2
    y_offset = (HEIGHT - bg_height) // 2

    # Return the live offset values along with the offsets
    return x_offset, y_offset
```

Final Result:


Score: 0
Score: 0
1000
1000