

# **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 `collidirect` 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.

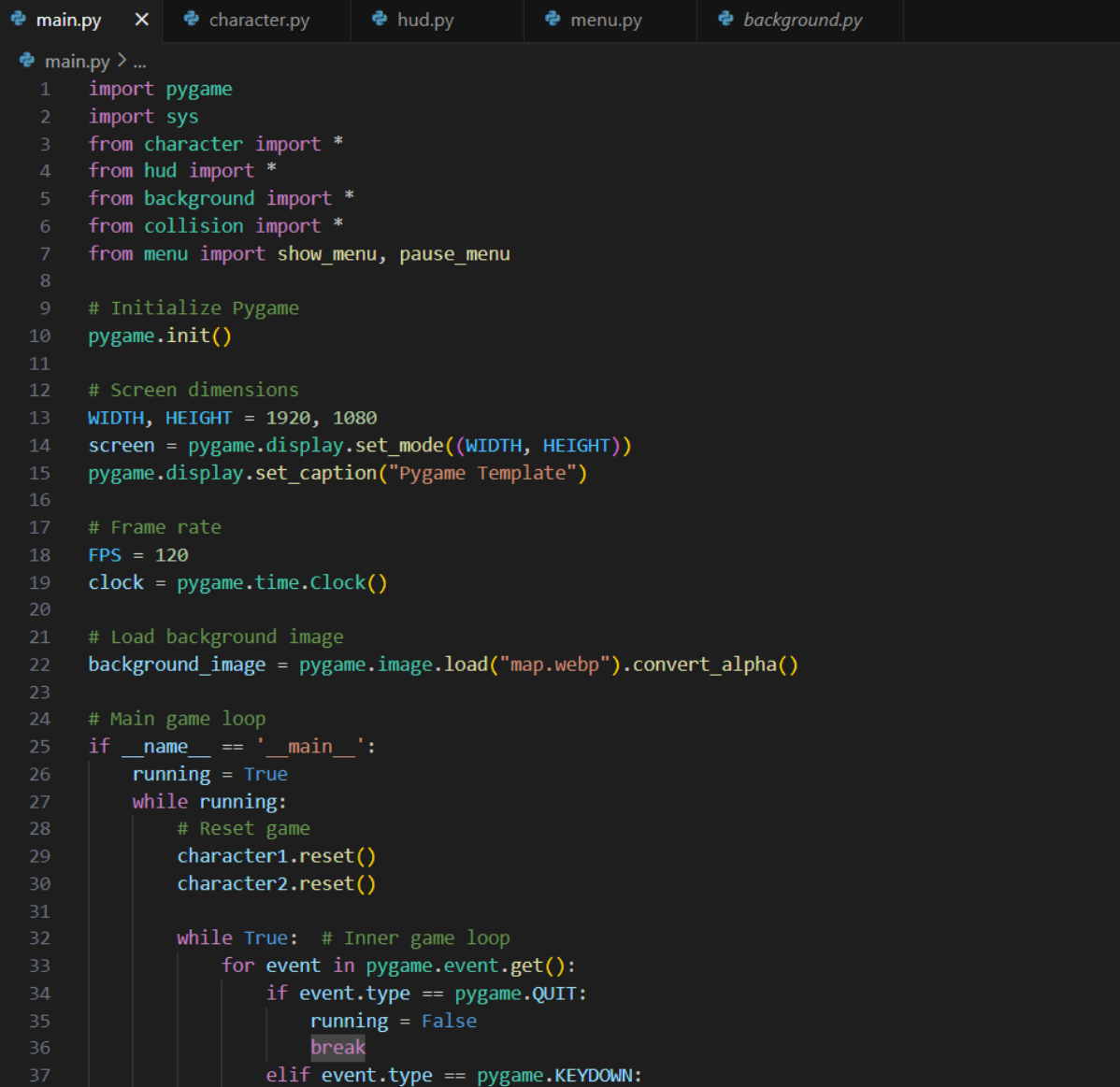
The character1 and character2 is then defined to pass all the arguments that are inside of the Character class. It also uses a dictionary to pass the animation argument for the Character class. It also helps handle to decide which animation is referred to in the update animation and set action function.

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  $\frac{1}{3}$  of the arc for the second attack and fills  $\frac{1}{8}$  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 x character.py hud.py menu.py background.py
main.py > ...
1 import pygame
2 import sys
3 from character import *
4 from hud import *
5 from background import *
6 from collision import *
7 from menu import show_menu, pause_menu
8
9 # Initialize Pygame
10 pygame.init()
11
12 # Screen dimensions
13 WIDTH, HEIGHT = 1920, 1080
14 screen = pygame.display.set_mode((WIDTH, HEIGHT))
15 pygame.display.set_caption("Pygame Template")
16
17 # Frame rate
18 FPS = 120
19 clock = pygame.time.Clock()
20
21 # Load background image
22 background_image = pygame.image.load("map.webp").convert_alpha()
23
24 # Main game loop
25 if __name__ == '__main__':
26     running = True
27     while running:
28         # Reset game
29         character1.reset()
30         character2.reset()
31
32         while True: # Inner game loop
33             for event in pygame.event.get():
34                 if event.type == pygame.QUIT:
35                     running = False
36                     break
37                 elif event.type == pygame.KEYDOWN:
```

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

```
main.py character.py x hud.py menu.py background.py
character.py > Character > player_direction
1 import pygame
2 from collision import * # Imoport the collisions to check for collisions
3
4 WIDTH, HEIGHT = 1920, 1080 # Set the screen height and width
5 screen = pygame.display.set_mode((WIDTH, HEIGHT))
6 pygame.display.set_caption("Pygame Template")
7 # Initialize sound mixer
8 pygame.mixer.init()
9 character1_attack1_sound = pygame.mixer.Sound("sound/character1_attack1_sound.mp3") # Adjust the path to your sound file
10 character1_attack2_sound = pygame.mixer.Sound("sound/character1_attack2_sound.mp3") # Adjust the path to your sound file
11 character2_attack1_sound = pygame.mixer.Sound("sound/character2_attack1_sound.mp3") # Adjust the path to your sound file
12 character2_attack2_sound = pygame.mixer.Sound("sound/character2_attack2_sound.mp3") # Adjust the path to your sound file
13 jump_sound = pygame.mixer.Sound("sound/jump_sound.mp3") # Adjust the path to your sound file
14 land_sound = pygame.mixer.Sound("sound/land_sound.mp3") # Adjust the path to your sound file
15 jump_sound.set_volume(0.2)
16 land_sound.set_volume(0.2)
17
18 death_sound = pygame.mixer.Sound("sound/death_sound.mp3") # Adjust the path to your sound file
19
20 # Load all the necessary animations for character 1
21 character1_walk = [pygame.image.load(f"character1/walk/walk_{i}.png").convert_alpha() for i in range(1, 6)]
22 character1_idle = [pygame.image.load(f"character1/idle/idle_{i}.png").convert_alpha() for i in range(1, 7)]
23 character1_jump = [pygame.image.load(f"character1/jump/jump_{i}.png").convert_alpha() for i in range(1, 9)]
24 character1_dead = [pygame.image.load(f"character1/dead/dead_{i}.png").convert_alpha() for i in range(1, 6)]
25 # Attack1 animation
26 character1_attack1_side = [pygame.image.load(f"character1/attack_1/side/attack_{i}.png").convert_alpha() for i in range(1, 4)]
27 character1_attack1_up = [pygame.image.load(f"character1/attack_1/up/attack_{i}.png").convert_alpha() for i in range(1, 4)]
28 # Attack2 animation
29 character1_attack2_side = [pygame.image.load(f"character1/attack_2/side/attack_{i}.png").convert_alpha() for i in range(1, 14)]
30 character1_attack2_up = [pygame.image.load(f"character1/attack_2/up/attack_{i}.png").convert_alpha() for i in range(1, 12)]
31 # Ult animation
32 character1_ult_side = [pygame.image.load(f"character1/ult/side/attack_{i}.png").convert_alpha() for i in range(1, 10)]
33 character1_ult_up = [pygame.image.load(f"character1/ult/up/attack_{i}.png").convert_alpha() for i in range(1, 10)]
34
35 character1_hurt = [pygame.image.load(f"character1/hurt/hurt_{i}.png").convert_alpha() for i in range(1, 3)]
36 character1_run = [pygame.image.load(f"character1/run/run_{i}.png").convert_alpha() for i in range(1, 8)]
37
```

## Image 3: hud.py: Drawing the HUD display for the characters

```
main.py character.py x hud.py menu.py background.py
hud.py > draw_hud
5 def draw_hud(offset_x, offset_y):
34     score_font = pygame.font.Font(None, 48)
35     hud_font = pygame.font.Font(None, 36)
36
37     # Define colors for HUD
38     health_bar_color = (255, 0, 0)
39     skill_bar_color = (0, 255, 0)
40     background_bar_color = (50, 50, 50)
41     # Bar size
42     health_bar_width = 300
43     bar_height = 40
44     max_skill_arc = 360 # Max degrees for the skill
45
46     # Character 1's attack2 circle properties
47     skill_circle_radius_1 = 30 # Radius for Character 1's attack2 counter
48     skill_circle_thickness_1 = 10 # Thickness for Character 1's attack2 circle
49     skill_circle_color_1 = (211, 211, 211)
50
51
52     ulti_circle_radius_1 = 50 # Radius for Character 1's skill counter
53     ulti_circle_thickness_1 = 20 # Thickness for Character 1's skill circle
54
55     # Character 2's attack2 skill circle properties
56     skill_circle_radius_2 = 30 # Radius for Character 2's skill counter
57     skill_circle_thickness_2 = 10 # Thickness for Character 2's skill circle
58     skill_circle_color_2 = (211, 211, 211)
59
60
61     ulti_circle_radius_2 = 50 # Radius for Character 1's skill counter
62     ulti_circle_thickness_2 = 20 # Thickness for Character 1's skill circle
63
64     # Draw Character 1's HUD
65     # Score for character 1
66
67     screen.blit(score_font.render(f"Score: {character1.point}", True, (0, 0, 255)), (WIDTH // 3 - 100, 120))
68
```

Image 4: menu.py: Showing the winning menu with play again and exit button

```
main.py character.py hud.py menu.py x background.py
menu.py > show_menu
4 # Function to blur the background
5 def blur_surface(surface, amount):
6     blur = pygame.transform.smoothscale(surface, (surface.get_width() // amount, surface.get_height() // amount))
7     return pygame.transform.smoothscale(blur, surface.get_size())
8
9 # Menu function to the winner and points
10 def show_menu(screen, background_image, player1_points, player2_points, winner, WIDTH, HEIGHT):
11     blurred_background = blur_surface(background_image, 10)
12     winner_font = pygame.font.Font(None, 130)
13     points_font = pygame.font.Font(None, 120)
14
15     # Determine winner text color
16     if winner == 1:
17         winner_text = winner_font.render("Player 1 Wins!", True, (255, 0, 0)) # Red
18     else:
19         winner_text = winner_font.render("Player 2 Wins!", True, (0, 0, 255)) # Blue
20
21     # Points text
22     player1_points_text = points_font.render(f"{player1_points}", True, (255, 0, 0)) # Red
23     player2_points_text = points_font.render(f"{player2_points}", True, (0, 0, 255)) # Blue
24
25     # Button size
26     button_width, button_height = 300, 80
27     play_again_rect = pygame.Rect((WIDTH // 2 - button_width // 2, HEIGHT // 2), (button_width, button_height))
28     exit_rect = pygame.Rect((WIDTH // 2 - button_width // 2, HEIGHT // 2 + 120), (button_width, button_height))
29
30     while True:
31         screen.blit(blurred_background, (0, 0)) # Draw the blurred background
32
33         # Draw the winning texts
34         screen.blit(player1_points_text, (WIDTH // 2 - winner_text.get_width() - player1_points_text.get_width() + 50, HEIGHT // 2 - 200))
35         screen.blit(winner_text, (WIDTH // 2 - winner_text.get_width() // 2, HEIGHT // 2 - 200)) # winner text
36         screen.blit(player2_points_text, (WIDTH // 2 + winner_text.get_width() - 50, HEIGHT // 2 - 200)) # Player 2 score
37
38         # Draw buttons
39         pygame.draw.rect(screen, (0, 128, 0), play_again_rect) # Green play again button
40         pygame.draw.rect(screen, (128, 0, 0), exit_rect) # Red exit button
```

Image 5: collision.py: Defining all the collisions in the map

```
main.py character.py hud.py menu.py collision.py x
collision.py > ...
1 import pygame
2
3 WIDTH, HEIGHT = 1920, 1080
4
5 # Collision class to produce the collisions for every rect and return the value
6 class Collision:
7     def __init__(self, name): # Sets the name of each rect initially
8         self.name = name
9
10     def rect(self, x, y, w, h): # Creates the rect based on the values given and returning the value
11         rect = pygame.Rect(x, y, w, h)
12         return rect
13
14 # All the rects with the names and values so that it can be used to check for collision
15 ground = Collision('ground').rect((WIDTH // 3), (1.14 * HEIGHT // 2), (WIDTH // 3), 1)
16 bottom = Collision('bottom').rect((WIDTH // 3), (1.14 * HEIGHT // 2) + 1, (WIDTH // 3), (HEIGHT // 2))
17 platform1 = Collision('platform1').rect(460, 470, 275, 1)
18 platform2 = Collision('platform2').rect(715, 370, 485, 1)
19 platform3 = Collision('platform3').rect(1190, 470, 275, 1)
20 border_bottom = Collision('border_bottom').rect(-1000, HEIGHT + 1000, WIDTH + 2000, 1000)
21 border_top = Collision('border_top').rect(-1000, HEIGHT - 4000, WIDTH + 2000, 1000)
22 border_left = Collision('border_left').rect(-3000, HEIGHT - 1000, 2000, HEIGHT + 2000)
23 border_right = Collision('border_right').rect(WIDTH + 1000, -1000, 2000, HEIGHT + 2000)
24
25
26
```

## Image 6: background.py: Setting the background and offsets

```
main.py character.py hud.py menu.py background.py X
background.py > ...
1 import pygame
2 from character import character1, character2 # Import the characters
3
4 WIDTH, HEIGHT = 1920, 1080
5 screen = pygame.display.set_mode((WIDTH, HEIGHT))
6 zoom_factor = 1.2
7
8 def zoom(image, zoom_factor): # Zooms the background image
9     zoomed_image = pygame.transform.scale(image, (int(WIDTH * zoom_factor), int(HEIGHT * zoom_factor)))
10    return zoomed_image
11
12 def camera(): # Sets the offset for the camera
13     init_mid_x = 1.425 * WIDTH // 3 # The initial average or midpoint of the two characters
14     init_mid_y = 0.99 * HEIGHT // 2
15
16     mid_x = (character1.xpos + character2.xpos) / 2 # Calculates the change of the average of the two characters in real time
17     mid_y = (character1.ypos + character2.ypos) / 2
18
19     offset_x = mid_x - init_mid_x # Calculates the coordinates of the midpoint where the camera should be
20     offset_y = mid_y - init_mid_y
21     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
22     offset_y = max(-200, min(200, offset_y))
23
24     return offset_x, offset_y # Return the offsets
25
26 def center_background(zoomed_image): # Center the background initially
27     bg_width, bg_height = zoomed_image.get_size()
28     x_offset = (WIDTH - bg_width) // 2
29     y_offset = (HEIGHT - bg_height) // 2
30
31     # Return the live offset values along with the offsets
32     return x_offset, y_offset
33
```

## Final Result:

