

A PVP Fighting Game using the Pygame module

Final Project Report (Algorithm & Programming)



Timothy Jonathan Immanuel (2802521825)

L1BC

Jude Joseph Lamug Martinez MCS

Binus International
University Jakarta

Abstract

The following report is the final project documentation for Algorithm and Programming class, designed to show a student's ability in coding with Python. This project is about making a fighting game in PvP (Player Vs Player) mode. The game implements basic ideas of Python programming using the Pygame module to let players fight in real time by managing their health and using attack combos.

Introduction

The goal of the Project is to apply all that the students have learned about the Python Programming language and problem solving to solve a relatively small but interesting problem. The project could be in the form of an app, game or program that encourages students to extend their comfort zone and look for topics beyond what was taught in class. The objective of this project is to design, implement, and test a Python-based solution that demonstrates mastery of algorithms, programming concepts, and problem-solving techniques.

Project Inspiration

The inspiration for creating this PvP game stemmed from my general interest in playing PvP games (eg: Tekken, Mortal Kombat, Shadow fight, etc..), in which some of the mechanics provided challenges and certain strategies for the player to master. These fighting games have a long-standing popularity due to its intense and competitive gameplay which ultimately gave me the idea of creating a game that captures the essence of an intense and competitive gameplay with some simple physics and game mechanics. PvP fighting games provide an engaging form of entertainment that also acts as a stress reliever by creating immersive gameplay experiences that allow players to disconnect from the real world. Additionally, the idea of creating social interaction and friendly competition among different players was also a driving force behind the project, emphasizing the importance of fun and engaging content in what it feels like designing a game. Lastly, This project allowed me to also apply the programming skills that I have learnt in class as well as on my own such as learning how to use the Python module, *Pygame*.

Project Design

Before going into this project, my first intention was by making a clone of a similar fighting game called *Shadow Fight* which more or less introduces the same game mechanics such as a health bar, keeps track of player scores, collision mechanics and swift animations which makes the game incredibly addictive. Similarly, this project is a 2D-fighting game that features two fighters with unique animations and engaging gameplay. This game will emphasize responsive controls, smooth animations and a simple scoring mechanic creating a rather competitive atmosphere.

- Framework and modules used for this project:
 - **Pygame**: module used to handle character animations, movement and attack inputs, attacking logic, as well as importing the game environment and sound effects.
 - **Python**: The core programming language used to program the game mechanics such as scoring, game logic and animation logic.
- The game has several main components that are part of the game's structure and also ensures that the game runs smoothly. These are:
 - A main game loop
 - The Character class
 - Environment and sound effects
 - User input (eg: move, jump and attack)
- The fighters would also have several attributes such as:
 - Health bar
 - State dependencies (attack state, jump state, alive state,...)
 - Animations (Idle, Attack 1 & Attack 2, Jump, Run, Drop)
 - A knockback and cooldown attribute
 - Movement keys:
 - Player 1 (Ninja):
 - Movement: W, A, D
 - Attack 1: G
 - Attack 2: T
 - Player 2 (Bulky Guy)
 - Movement: Up arrow key, Left arrow key, Right arrow key
 - Attack 1: L
 - Attack 2: P

- Design Challenges:
 - Animation handling:
 - Managing the animation and the frame updates requires creating several loops and ensuring that each of the fighter's animation runs smoothly and accounted for.
 - Editing and resizing the spritesheet requires careful precision to ensure that each frame and pixels fit with the overall animation.
 - Collision detection and attacking mechanics:
 - Designing the collision mechanics requires creating perfectly sized hitboxes that fit with the character's animation and ensures that the collisions are detected on each of the player's rectangles.
 - Screen boundaries and gravity mechanics:
 - Assigning the screen boundaries ensures that the player stays on screen while the game is running. This makes use of marking the player's current position on the screen which can get a bit confusing.
 - The jumping mechanics require an essential element called gravity which is assigned a value. Implementing gravity to ensure that the player lands back down once they have finished jumping requires a lot of conditions that can get overwhelming.
- Future Improvements:
 - Introducing more characters and animations
 - More attack types that deals more damage
 - Introduce different maps
 - A character switch option
 - Special moves tailored to each character
 - 3D
 - Multiplayer network for remote playing

Discussion

When beginning to write the code for this project, the *Pygame* module is imported in a file called *blitzbattle.py* in order to import all the necessary assets, tools and functions that could then be used to construct this PvP game. The first part of the code contains all the things that are necessary to initialize the *Pygame* module in the program, such as the *pygame.init()* function, setting up the window and its size (*pygame.display.set_mode(window_size)*) as well as setting up the main game loop (*def main_game()*) which ensures that the program keeps running and the window stays on opening. The *clock.tick()* function was also used to determine the fps (frames per second) the game will run on, in this case, the game runs on 60 fps. Next, *exit* was imported from the *sys* module to ensure that there is a way for the player to close the game/program once they feel like they are finished playing and then the loop will break, this is also part of the main game loop. A couple lists are created to keep track of the players' score and also which player has won the round. The two instances of the characters are also created in this file in which these instances are taken from a separate file called *Charactersbackup.py*, in which it contains a *Character* class for the players and makes use of OOP (Object-Oriented Programming). *Charactersbackup.py* will be explained more later

Several functions are created for displaying text and blitting images onto the screen that was initialized previously, these functions include *def display_assets()* that blits the character and the background map onto the screen, *def display_text()* which prints out the numbers for the count down and the scores for each player, and finally, *def health_bar()* which draws the health bars for the characters. All these functions are called inside the main game loop, *def main_game()*.

The main game loop consists of several game mechanics logic that is responsible for resetting the game and introducing a timer system between each round as well as displaying the scores of each player after each round. This mainly consists of several *if-else* statements that dictates whether the round has started or ended depending on the current game time and when the character's health reaches zero. This is determined by *pygame.time.get_ticks()*. If the round has ended, the game would reset and the countdown would reset down from three again, this is also determined by using the *get_ticks()* function in *Pygame* where it takes the current time in the round and subtract it from the overall time of the game. A list is created to keep track of players' scores by adding one point after each round depending on who wins the round.

Some crucial game elements are also found inside *blitzbattle.py*, for example, importing music and the sound effects for the game are done by importing *mixer* from the *Pygame* module as well. The volume and when the sounds should play are both determined inside *blitzbattle.py* and *Charactersbackup.py*.

The *Charactersbackup.py* file contains the *Character* class in which the instances of the fighters are taken from. This class contains several *self* variables that are used to define the character attributes such as the health, jump state, attack state, alive state and many more. These *self* variables will later be used in methods initialized inside the class as well. Several methods such as the *move()* and *attack()* methods are used to define the character movements when different keys on the computer are pressed. The *attack()* method contains all the necessary collision and health logic for the fighter as well as creating a hit box when attacking.

Character animations are loaded into the program from an external sprite sheet that is found online and each picture is appended to a list and then iterated through a loop creating an animation for the character. In total, there are 8 animations (idle, running, jumping, falling, attack1, attack2, take damage and death). The list that the animations are stored in are stored in a temporary list that is then appended to a main list, creating a list inside a list, thus, creating an index for every set of animation in the main list. This process is repeated for every action the character has. Then, a method called *update()* is created to ensure that the animations in the lists are animating properly by iterating from using the indexes of both the main list and the temporary list. Next the method *diff_action()* is created to enable the change of animations when a key is pressed on the computer (eg: pressing the 'W' key triggers the running animation and changes it from the idle animation). This method is called under the *move()* method.

The fighter mechanics such as attacking and jumping are created inside the *move()* method and the *attack()* method. Initially, both fighters are not jumping which is defined by a variable (*self.jump_state*). Then, this variable is set to True whenever the assigned 'jump' key is pressed on the keyboard. Finally, the gravity variable contains a value that would immediately bring the fighter back down to the ground when it reaches a certain height. This is initialized inside the *move()* method as well. As for the attacking, a rectangle is drawn in front of the fighter everytime the assigned 'attack' button is pressed. The collision is checked by using the *Pygame* function *colliderect()* which checks if the attack rectangle of the player collides with the opponent's character rectangle. If they collide, the health of the fighters would be reduced.

Knockback mechanics are also introduced by shifting the player's current x position backwards by a certain value when being hit by the other player, indicating that a hit has been successfully done.

Cooldown mechanics are also introduced which essentially prevents the players from constantly being able to attack, adding more strategy and rhythm to the gameplay. This is also done by using the *get_ticks()* function which records the last time a player's hit action has been performed and subtracting the time from the overall game time, allowing the players to have a cooldown between the attacks, which in this case is 1000 milliseconds/1 second between each attack.

Screenshots

Image 1: Pygame initialization (*blitzbattle.py*)

```
game.py x Characters.py
1 import pygame
2 from sys import exit
3 from Characters import Character # Import the Character class from character.py
4
5 pygame.init()
6
7 # Window Formatting
8 window_size = (800, 600)
9 window = pygame.display.set_mode(window_size)
10 pygame.display.set_caption("Fighting Game")
11
12 # Background image
13 background1 = pygame.image.load("Assets/Neon Background.png").convert_alpha()
14
15 # Characters
16 character1 = Character(image_link: "Assets/Character.png", size: (135, 235), position: (150, 300))
17 character2 = Character(image_link: "Assets/Character 2.png", size: (135, 235), position: (600, 300))
18
19 def display_assets(): 1 usage
20     # Draw background and characters
21     window.blit(background1, dest: (0, 0))
22     character1.draw(window)
23     character2.draw(window)
```

Image 2: Visual and sound effects (*blutzbattle.py*)

```
#set volume for music
pygame.mixer.music.load('Assets/03 Samurai Spirit.mp3')
pygame.mixer.music.set_volume(0.3)
pygame.mixer.music.play(-1, start: 0.0, fade_ms: 3000)

#game fonts for the score and countdown
countdown_font = pygame.font.Font(name: "Assets/Karate.ttf", size: 80)
score_font = pygame.font.Font(name: "Assets/Karate.ttf", size: 30)

#score mechanics
character1_score = [0] #character's 1 score (ninja)
character2_score = [0] #character 2 score (bulky guy)
round_end = False
round_cooldown = 2000

#round start sound effect
yoo_sound = pygame.mixer.Sound('Assets/Yooooo.mp3')
yoo_sound.set_volume(0.1)
yoo_sound.play()

#characters sword sound effects
ninja_fx = pygame.mixer.Sound('Assets/Katana Swing Cut - Sound Effect for editing.mp3')
ninja_fx.set_volume(0.3)
bulky_guy_fx = pygame.mixer.Sound('Assets/Sword Stab Sound Effect (HD).mp3')
bulky_guy_fx.set_volume(0.2)
```

Image 3: Character initialization and game functions (*blitzbattle.py*)

```
52 jump = pygame.mixer.Sound('Assets/Jump Sound Effect (High Quality).mp3')
53 jump.set_volume(0.3)
54
55 # Background image
56 background1 = pygame.image.load("Assets/bamboo forest (2).jpg").convert_alpha()
57 # Characters
58 character1 = Character(char_type: 'Character 1', size: (90,100), position: (100, 800), flip: False, ninja_fx, being_hit, jump)
59 character2 = Character(char_type: 'Character 2', size: (90, 100), position: (600, 800), flip: True, bulky_guy_fx, being_hit, jump)
60
61 def display_assets(): 1 usage
62     # Draw background and characters
63     window.blit(background1, dest: (2, 0))
64     character1.update()
65     character2.update()
66     character1.draw(window)
67     character2.draw(window)
68
69 def display_text(txt, font, color, x, y): 3 usages
70     img = font.render(txt, True, color)
71     window.blit(img, dest: (x, y))
72
73
74 def health_bar(health, x, y): 2 usages
75     health_percentage = health /1000
76     pygame.draw.rect(window, color: (255, 255, 255), rect: (x-15, y-10, 230, 60))
77     pygame.draw.rect(window, color: (0, 0, 0), rect: (x, y, 200, 40))
78     pygame.draw.rect(window, color: (0, 255, 0), rect: (x, y, 200 * health_percentage, 40))
```

Image 4: Main game loop (*blitzbattle.py*)

```

def main_game(): 1 usage
    global countdown_update, countdown, countdown_font, score_font, yoo_sound, character2_score, character1_score, round_end, round_cooldown, character1
    clock = pygame.time.Clock()
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()

        window.fill((0, 0, 0))

        display_assets() #displays the background

        health_bar(character1.health, x: 30, y: 30)
        health_bar(character2.health, x: 570, y: 30)

        display_text(str(character1_score[0]), countdown_font, color: (255,0,0), 800/4.5, 600/7)
        display_text(str(character2_score[0]), countdown_font, color: (255,0,0), x: 575, 600/7 )

        keys = pygame.key.get_pressed()
        if countdown <= 0:
            # Character movements
            character1.move(keys, pygame.K_a, pygame.K_d, pygame.K_w, window, character2)
            character2.move(keys, pygame.K_LEFT, pygame.K_RIGHT, pygame.K_UP, window, character1)
        else:
            display_text(str(countdown), countdown_font, color: (255,0,0), 800 / 2, 600 / 3)

```

```

def main_game(): 1 usage
    if round_end == False:
        if character1.alive == False:
            character2_score[0] += 1
            round_end = True
            round_time = pygame.time.get_ticks()
            print(character2_score)
        elif character2.alive == False:
            character1_score[0] += 1
            round_end = True
            round_time = pygame.time.get_ticks()
        else:
            if pygame.time.get_ticks() - round_time > round_cooldown:
                round_end = False
                countdown = 3
                character1 = Character(char_type: 'Character 1', size: (90, 100), position: (100, 800), flip: False, ninja_fx, being_hit, jump)
                character2 = Character(char_type: 'Character 2', size: (90, 100), position: (600, 800), flip: True, bulky_guy_fx, being_hit, jump)

            # #makes sure that the characters face each other
            # character1.flip_character(character2)
            # character2.flip_character(character1)

        pygame.display.flip()
        clock.tick(60)

#run game
main_game()

```

Image 5: The Character class (*Charactersbackup.py*)

```

import pygame

class Character(pygame.sprite.Sprite): 5 usages
    def __init__(self, char_type, size, position, flip, sound, ouch, jump_sound):
        super().__init__() # Initialize sprite class
        self.char_type = char_type
        self.animation_list = []
        self.frame_index = 0
        self.update_time = pygame.time.get_ticks()
        self.actions = 0

```

Image 6: Animation handling process (*Charactersbackup.py*)

```
temp_list = []
for i in range(4): # idle animation (0)
    img = pygame.image.load(f'Assets/{char_type}/idle/{i}.png').convert_alpha()
    img = pygame.transform.scale(img, size) # Scale the image
    temp_list.append(img)
self.animation_list.append(temp_list)
temp_list = []
for i in range(8): # running animation (1)
    img = pygame.image.load(f'Assets/{char_type}/run/{i}.png').convert_alpha()
    img = pygame.transform.scale(img, size) # Scale the image
    temp_list.append(img)
self.animation_list.append(temp_list)
temp_list = []
for i in range(2): # jumping animation (2)
    img = pygame.image.load(f'Assets/{char_type}/jump/{i}.png').convert_alpha()
    img = pygame.transform.scale(img, size) # Scale the image
    temp_list.append(img)
self.animation_list.append(temp_list)
temp_list = []
for i in range(2): # falling animation (3)
    img = pygame.image.load(f'Assets/{char_type}/drop/{i}.png').convert_alpha()
    img = pygame.transform.scale(img, size) # Scale the image
    temp_list.append(img)
self.animation_list.append(temp_list)
temp_list = []
```

Image 7: Character attributes (*Charactersbackup.py*)

```
# Character attributes
self.jump_state = False
self.jump_velocity = 0
self.gravity = 1.75
self.attack_state = False
self.health = 1000
self.flip = flip
self.sound_fx = sound
self.hit_sound = ouch
self.jumping = jump_sound
self.cooldown_duration = 1000 # Cooldown duration in milliseconds
self.last_attack_time = 0 # Timestamp of the last attack
self.attack_type = 0
self.hit = False
self.alive = True
```

Image 8: Animation updates (*Charactersbackup.py*)


```
80
81  def update(self): 2 usages
82     cooldown = 50
83     self.image = self.animation_list[self.actions][self.frame_index]
84     if pygame.time.get_ticks() - self.update_time > cooldown:
85         self.update_time = pygame.time.get_ticks()
86         self.frame_index += 1
87     if self.frame_index >= len(self.animation_list[self.actions]):
88         if self.alive == False:
89             self.frame_index = len(self.animation_list[self.actions]) - 1
90         else:
91             self.frame_index = 0
92             if self.actions in [4, 5]:
93                 self.attack_state = False
94             if self.actions == 6:
95                 self.hit = False
96
97     def diff_action(self, new_action): 7 usages
98         if new_action != self.actions: # Ensure current action is new
99             self.actions = new_action
100             self.frame_index = 0
101             self.update_time = pygame.time.get_ticks()
102
```

Image 9: Move method (*Charactersbackup.py*)

```
def move(self, keys, left, right, up, surface, target, speed=5): 2 usages
    # Screen boundaries
    if self.rect.x < 0:
        self.rect.x = 0
    if self.rect.x + self.rect.width > surface.get_width():
        self.rect.x = surface.get_width() - self.rect.width

    # If the character is attacking or hit, prioritize those actions
    if self.attack_state == True and self.alive == True:
        if self.attack_type == 1:
            self.diff_action(4) # Attack 1 animation
        elif self.attack_type == 2:
            self.diff_action(5) # Attack 2 animation
    elif self.health == 0:
        self.alive = False
        self.health = 0
        self.diff_action(7)
    elif self.hit:
        self.diff_action(6) # Getting hit animation
    elif keys[up]:
        self.diff_action(2) # Jumping animation
    elif keys[right] or keys[left]:
        self.diff_action(1) # Running animation
    else:
        self.diff_action(0) # Idle animation
```

Image 10: Attack method (*Charactersbackup.py*)

```
194
195     def attack(self, width_multiplier, target, surface): 2 usages
196         self.attack_state = True
197         self.sound_fx.play()
198         if not self.flip: # Character facing right
199             attack_area = pygame.Rect(self.rect.centerx, self.rect.y, width_multiplier * self.rect.width, self.rect.height)
200         else: # Character facing left
201             attack_area = pygame.Rect(self.rect.centerx - width_multiplier * self.rect.width, self.rect.y, width_multiplier * self.rect.width, self.rect.height)
202         pygame.draw.rect(surface, color=(255, 0, 0), attack_area, width=2)
203         if attack_area.colliderect(target.rect):
204             self.hit_sound.play()
205             if target.rect.centerx > self.rect.centerx:
206                 target.rect.x += 50
207                 target.health -= 50
208                 target.hit = True
209             if self.rect.centerx > target.rect.centerx:
210                 target.rect.x -= 50
211                 target.health -= 50
212                 target.hit = True
213             if target.health == 0:
214                 target.alive = False
215                 print("VICTORY!!")
```

Image 11: Blitting Characters to the screen

```
def draw(self, surface): 2 usages
    # Draw the character image
    if self.flip:
        flipped_image = pygame.transform.flip(self.image, flip_x=True, flip_y=False)
        surface.blit(flipped_image, self.rect)
    else:
        surface.blit(self.image, self.rect)
```

Image 12: Final Game

