

**An Action and Adventure Game using
the Pygame module**

**Final Project Report (Algorithm &
Programming)**



Stefan Luciano Kencana

(2802521314) 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 Zelda-like game in a simple pixelated art created by pixel boy (link is in credits). The game implements basic ideas of Python programming using the Pygame module to let players explore the map and fight enemies to gain exp and get stronger.

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 this game came from when I just bought a Nintendo switch. I wanted to build a game where it combines the concept of Zelda and a bit of the Elden Ring aspect where both of these games require players to play for at least 50+ hours of gameplay to actually finish the game. I was amazed by the graphics and mechanics of the game hence making this game. While AAA game's graphics can't be replicated in this manner using python's pygame module, it could resemble it in a different way just by completing the mechanics. Additionally, the idea of creating a challenging game, emphasizing the importance of fun and engaging content in what it feels like designing a game. Overall, 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

My goal when I was trying to make this project is to make a game that's similar or not, try to replicate the game called Zelda and add a bit of the Elden Ring's fighting elements whose objective is to complete the game's story and lore and achieve its ultimate objective. In this game, the goal is only to defeat every enemy and get as strong as possible. Both of the games are AAA games which include amazingly beautiful graphics and immerse players in the game's world, unfortunately pygame doesn't support those types of graphics which is why I'm using a pixelated 64 by 64 sprite.

- 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. There are:
 - First of all, the main game loop
 - The Level class which draws the object, map and characters using YSortCameraGroup with the help of Tile class
 - YSortCameraGroup which is a custom draw system that sorts the sprites level in which is front which is back in the Y axis that allows the camera system to be made
 - The Player and Enemy class which inherits from Entity Class (Inputs and how the enemies moves and all)
 - Magic and Weapon Class (which damages the enemies)
 - Particle Class which beautify and makes the game makes sense
 - UI (User Interface) Class which draws the health and mana bar, current weapon, current magic and experience
 - Upgrade Class which allows the player to increase their stats with the cost of experience by the monsters they defeat
 - Skin Class which allows the player to change how they appear (incomplete)
 - Last but not least, The settings which stores all set data

- Both Entities have several attributes such as:
 - Health, Damage, Speed
 - Hitbox (collision)
 - State dependencies (attack state, jump state, alive state,...)
 - Animations
 - A knockback and cooldown attribute
 - I-frame (invulnerability time after getting attacked)
- Enemy Specific attributes:
 - Agitated Radius
- Player Specific attributes:
 - Mana, Energy, Weapon damage, Magic Damage, Experience
 - Input keys:
 - Arrow Keys (up, down, left, right) → move up,down,left,right
 - Space Bar → Weapon Attack
 - Left Control (LCtrl) → Magic Attack
 - E → change magic
 - Q → change weapon
 - M → open upgrade menu or pause game
 - left and right arrow keys to navigate
 - space to increase selected stat
 - N → open skin menu
 - left and right arrow keys to navigate
 - space to use skin

- Design Challenges:
 - Weapon and Magic
 - Outputting the damage output from the player to the enemy while also giving options to change both weapon and magic that's being used is really hard as it requires me to position both of them to fit the character that is being used
 - Damage and Stats system
 - Creating stats, keeping tabs on the player and enemies stats is also quite challenging as it is a real time difference and it needs contact between hitboxes while also calculating base stats and weapon stats
 - Particle Controls
 - Positioning and scaling the particles to make it more realistic, while also choosing between available graphics is required as to make the game look more realistic
 - Upgrade, Skin, and Pause game system
 - Pausing the game and showing another UI (User Interface) is also a challenging part where it needs the game loop to be interfered. and also taking variable changes from stats and file directory to change real time stats and skin graphics
 - Level Maps:
 - Creating the game's map using **Tiled**, a third party software which provides a csv file and format and connecting it to pygame, while also designing the level is quite a challenging task and different section by itself, not only that but also learning on how to use the program.
 - Camera:
 - Creating the logic for the camera and sorting group really made me feel like starting over the project all over again as making groups for each sprite is kind of challenging, ensuring that the level can be drawn and the player can move freely around the map.
 - 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.
 - Collision detection:
 - Designing the collision mechanics requires offsets and creating hitboxes based on where the player stands, creating offset from the original sprite while also rescaling the size of the hitboxes
- Future Improvements:
 - Completing the Skin system
 - Introducing more characters and animations
 - More Weapon and Magic
 - Introduce different maps and multiplayer

Discussion

Starting the code for this project, we need to import the pygame module and setup the game loop as a game works is that, they go in an infinite loop waiting for an input from the user and stops the loop when the game quits. after setting up the main game loop in main.py I went to level.py to start creating the code on how to display the map and characters. I made another files called support.py and tile.py to help me import all data from csv files and assign tiles from the map that i made using Tiled (Third Party software app). I made level.py as the base game where it draws and displays all the graphics. The settings.py is where all the configs are and static data stored (resolution, FPS, colors, folder directory etc.) and it is used by to simplify the creation of the game so that it stays consistent.

After that I created the player.py where the player stats, inputs, and all player mechanics reside but after that when I created enemy.py the code seems repetitive so I made a new class called Entity from entity.py and made both player and enemy classes inherit from entity. I also made the player animations, hitbox and collisions there where all player features are there. After that the enemy.py is basically like the player code but the difference is that it is moved by a certain radius I set. So basically there are 2 set of radius, the first one is "aggro" zone in which the enemy will get agitated and go to the player in that radius, and there is also the attacking zone which means the radius is short enough so that the enemy's attack can reach the player. I also set cooldowns and I-frame timer for both the player and enemies. Later on after a few things are done i made the knockback and flickering when the enemies are on active i-frame using sin diagram and math library.

What comes next is the weapon mechanics in which I set the position of the sprite and set collision between the player's weapon and the enemy and vice versa. And then adding the damage logic of the player base stats to the weapon, while also being able to change the current weapon that is being used. After that I created both magic.py and particles.py as they are pretty much connected to each other, because for this game the magic sprites are particles. I set up all the particles that are being used, including the ones that are attached to the level and when the enemy's attack got to the player.

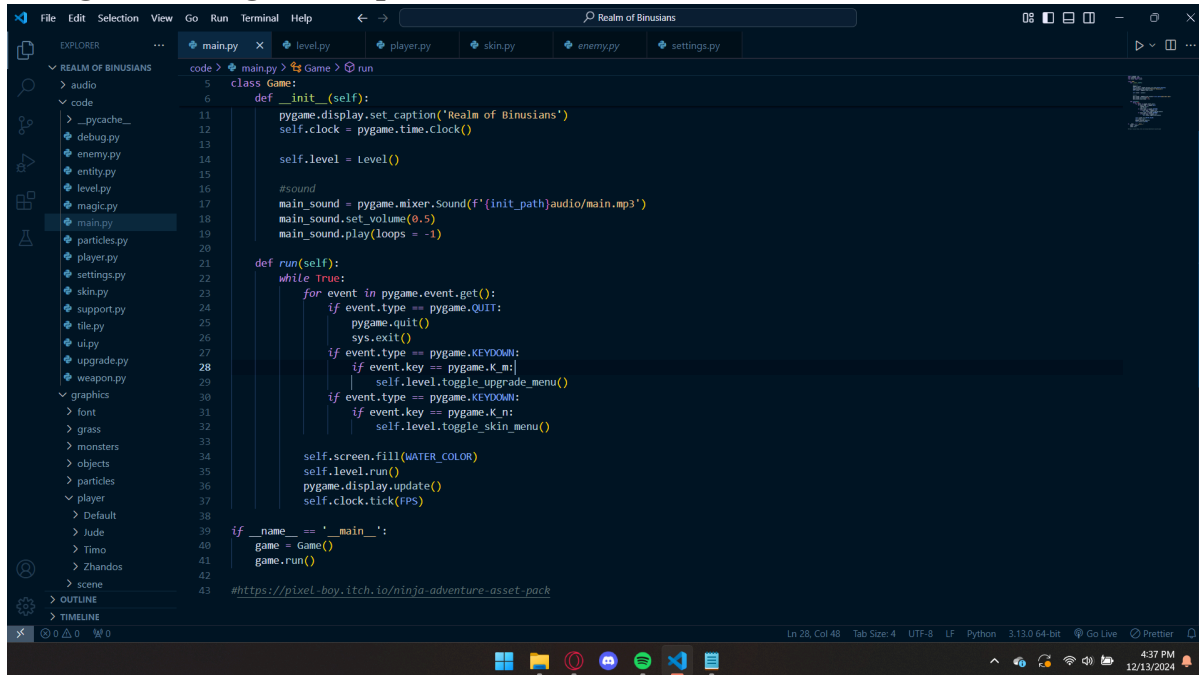
After all the game functions are working, I started making the UI (User Interface) and upgrading the system. Creating it requires me interfering with the main game loop as it also pauses the game. The upgrade system basically modifies the stats of the player to a certain limit using accumulated experience by killing enemies. The skin system is the

hard part where I need to change the directory folder of the player's animation so that the display of the player changes, it is not quite complete yet as of now. After all of that is done the final touch is adding the sound for the interactions and background sounds of the game.

All the Assets used in this project are from:
<https://pixel-boy.itch.io/ninja-adventure-asset-pack>

Screenshots

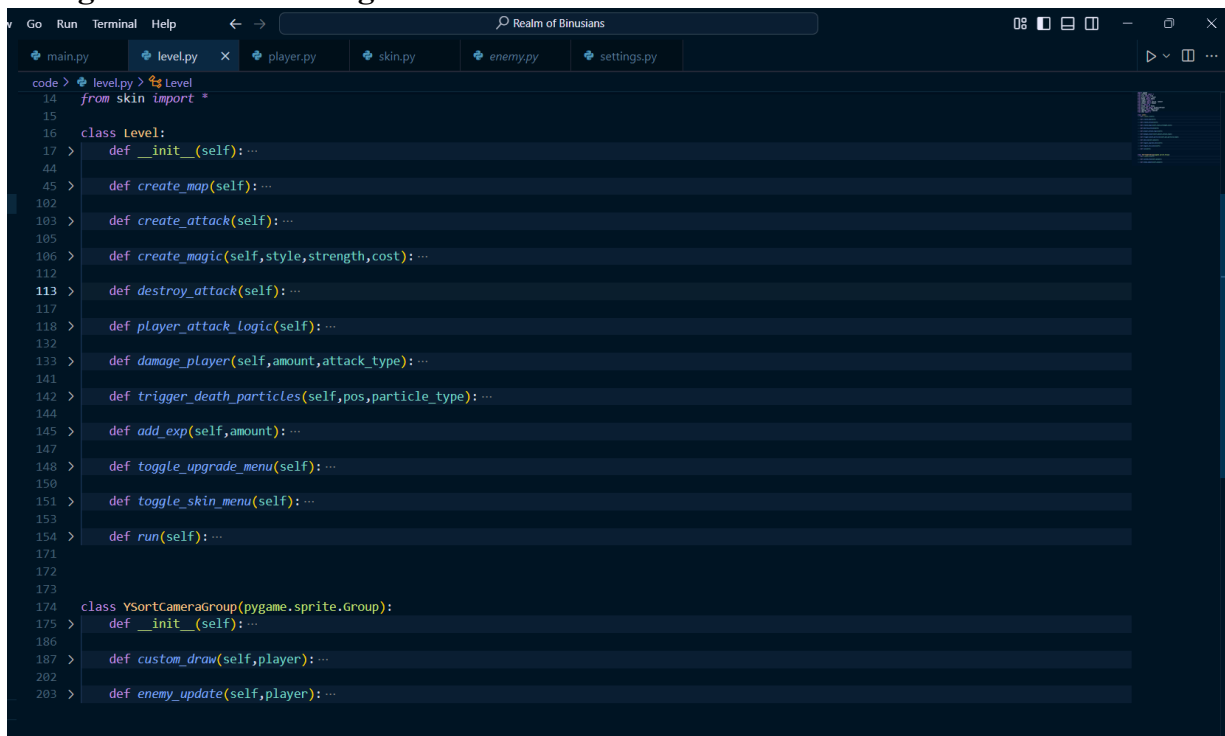
Image 1: Main game loop



The screenshot shows a code editor with the file explorer on the left displaying a project named 'REALM OF BINUSIANS'. The main editor window shows the code for 'main.py'. The code defines a 'Game' class with an '__init__' method and a 'run' method. The 'run' method contains a 'while True' loop that handles pygame events, including quitting, key presses for menu toggles, and screen updates. The status bar at the bottom indicates 'Ln 28, Col 48', 'Tab Size: 4', 'UTF-8', 'LF', 'Python', '3.13.0 64-bit', 'Go Live', and 'Prettier'.

```
code > main.py > Game > run
5 class Game:
6     def __init__(self):
11         pygame.display.set_caption('Realm of Binusians')
12         self.clock = pygame.time.Clock()
13
14         self.level = Level()
15
16         #sound
17         main_sound = pygame.mixer.Sound(f'{init_path}audio/main.mp3')
18         main_sound.set_volume(0.5)
19         main_sound.play(loops = -1)
20
21     def run(self):
22         while True:
23             for event in pygame.event.get():
24                 if event.type == pygame.QUIT:
25                     pygame.quit()
26                     sys.exit()
27                 if event.type == pygame.KEYDOWN:
28                     if event.key == pygame.K_m:
29                         self.level.toggle_upgrade_menu()
30                     if event.type == pygame.KEYDOWN:
31                         if event.key == pygame.K_n:
32                             self.level.toggle_skin_menu()
33
34                 self.screen.fill(WATER_COLOR)
35                 self.level.run()
36                 pygame.display.update()
37                 self.clock.tick(FPS)
38
39 if __name__ == '__main__':
40     game = Game()
41     game.run()
42
43 #https://pixel-boy.itch.io/ninja-adventure-asset-pack
```

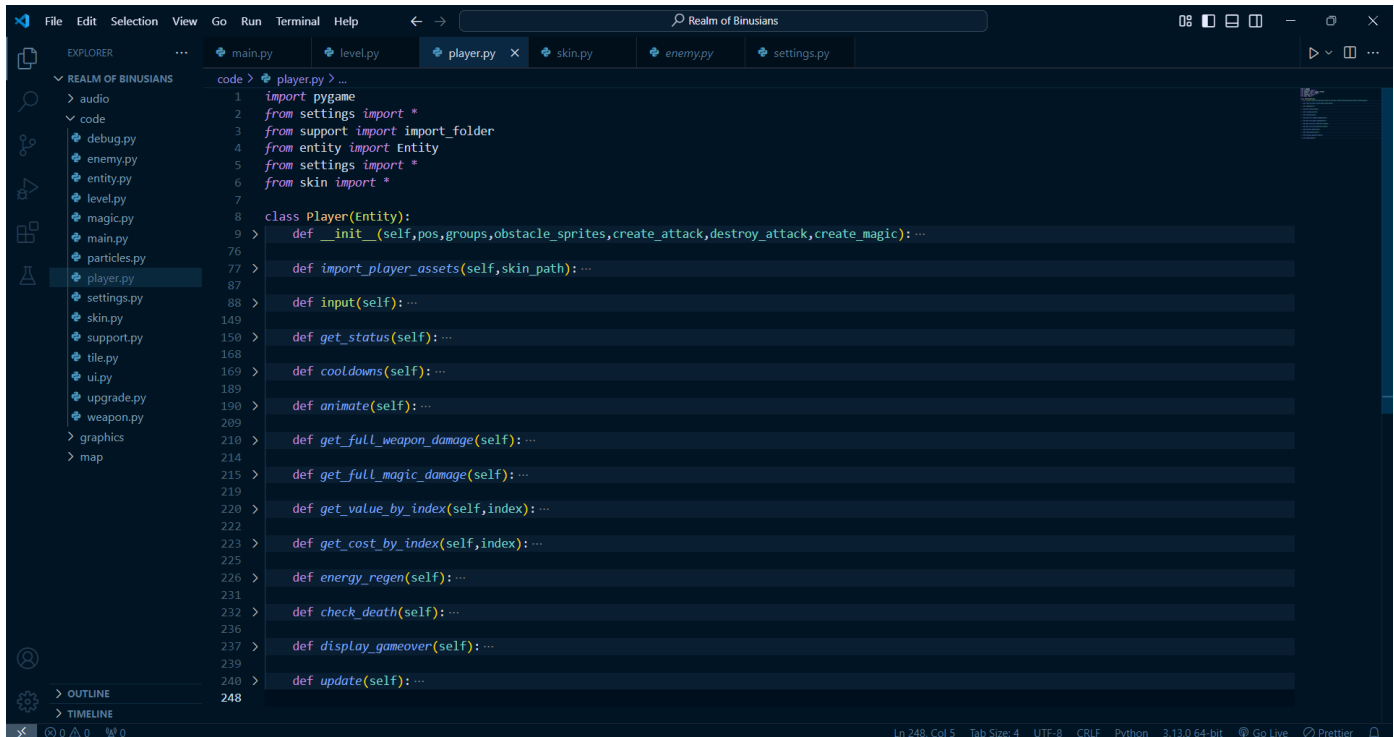
Image 2: Level Drawing and Camera



The screenshot shows a code editor with the file explorer on the left. The main editor window shows the code for 'level.py'. The code defines a 'Level' class with an '__init__' method and several methods for creating and managing game elements like maps, attacks, magic, and player logic. It also defines a 'YSortCameraGroup' class for camera logic. The status bar at the bottom indicates 'Ln 28, Col 48', 'Tab Size: 4', 'UTF-8', 'LF', 'Python', '3.13.0 64-bit', 'Go Live', and 'Prettier'.

```
code > level.py > Level
14 from skin import *
15
16 class Level:
17     def __init__(self): ...
18
19     def create_map(self): ...
20
21     def create_attack(self): ...
22
23     def create_magic(self, style, strength, cost): ...
24
25     def destroy_attack(self): ...
26
27     def player_attack_logic(self): ...
28
29     def damage_player(self, amount, attack_type): ...
30
31     def trigger_death_particles(self, pos, particle_type): ...
32
33     def add_exp(self, amount): ...
34
35     def toggle_upgrade_menu(self): ...
36
37     def toggle_skin_menu(self): ...
38
39     def run(self): ...
40
41
42 class YSortCameraGroup(pygame.sprite.Group):
43     def __init__(self): ...
44
45     def custom_draw(self, player): ...
46
47     def enemy_update(self, player): ...
```

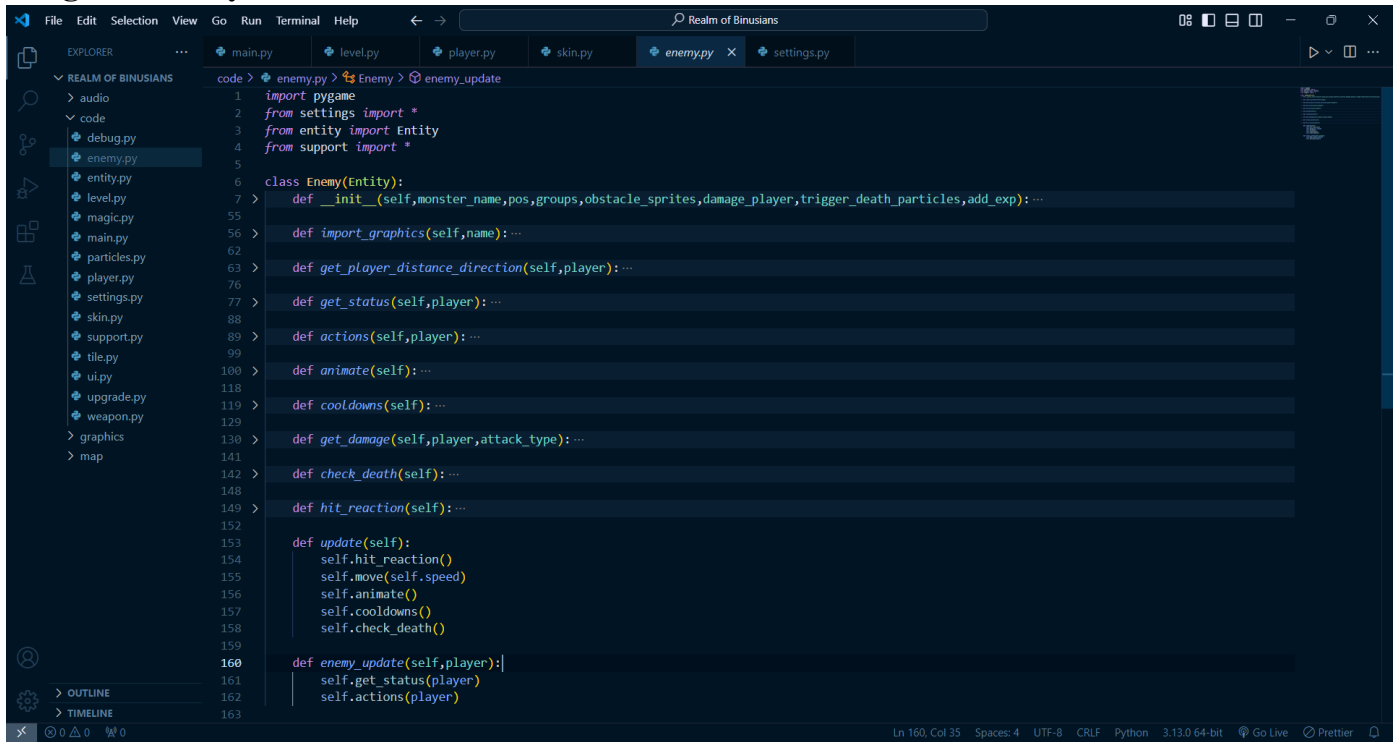
Image 3: Player Functions and Features



This screenshot shows a code editor with the file explorer on the left displaying a project named 'REALM OF BINUSIANS'. The 'code' folder is expanded, showing various Python files. The 'player.py' file is selected and open in the main editor. The code defines a 'Player' class that inherits from 'Entity'. The class includes an '__init__' method and several other methods for player management and game logic.

```
1 import pygame
2 from settings import *
3 from support import import_folder
4 from entity import Entity
5 from settings import *
6 from skin import *
7
8 class Player(Entity):
9     def __init__(self, pos, groups, obstacle_sprites, create_attack, destroy_attack, create_magic): ...
10
11     def import_player_assets(self, skin_path): ...
12
13     def input(self): ...
14
15     def get_status(self): ...
16
17     def cooldowns(self): ...
18
19     def animate(self): ...
20
21     def get_full_weapon_damage(self): ...
22
23     def get_full_magic_damage(self): ...
24
25     def get_value_by_index(self, index): ...
26
27     def get_cost_by_index(self, index): ...
28
29     def energy_regen(self): ...
30
31     def check_death(self): ...
32
33     def display_gameover(self): ...
34
35     def update(self): ...
```

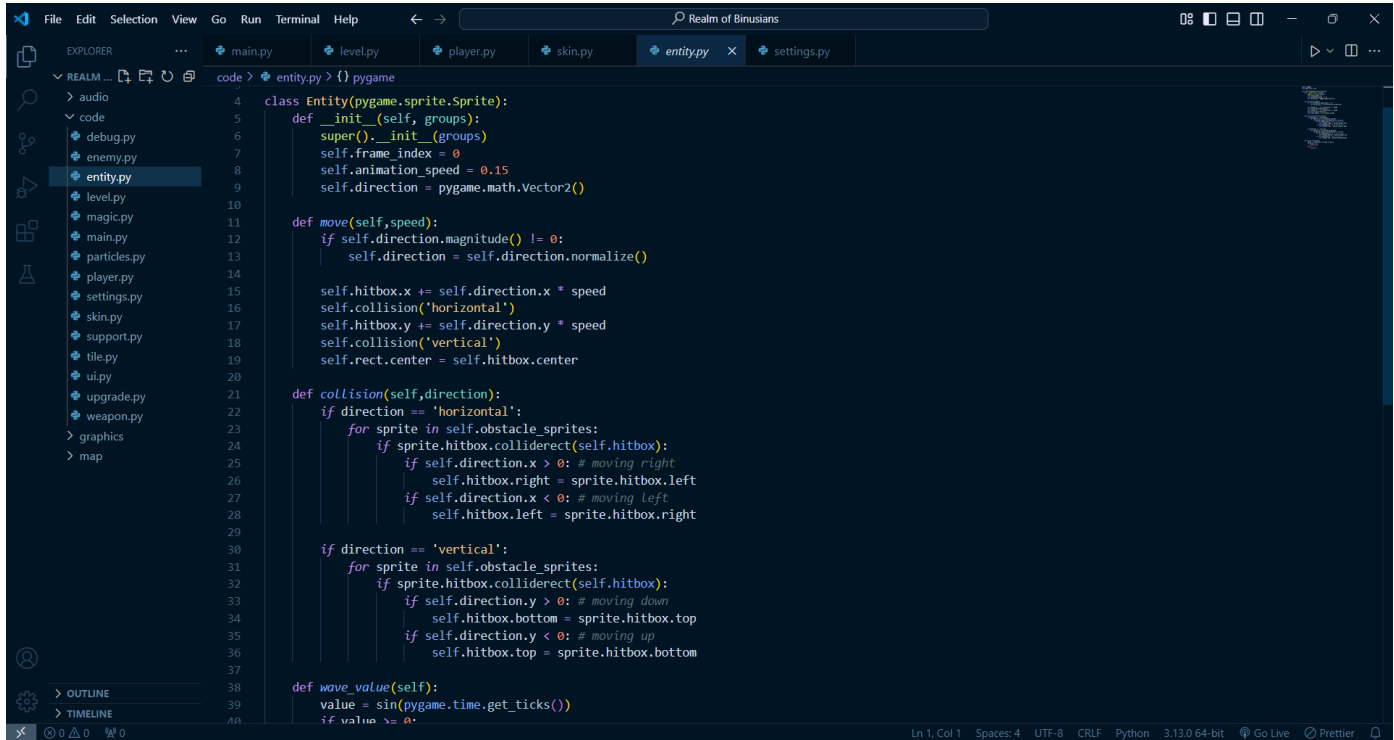
Image 4: Enemy functions and features



This screenshot shows the same code editor with the 'enemy.py' file selected and open. The code defines an 'Enemy' class that inherits from 'Entity'. The class includes an '__init__' method and several other methods for enemy management and game logic. The 'enemy_update' method is also defined, which calls other methods like 'get_status', 'actions', and 'check_death'.

```
1 import pygame
2 from settings import *
3 from entity import Entity
4 from support import *
5
6 class Enemy(Entity):
7     def __init__(self, monster_name, pos, groups, obstacle_sprites, damage_player, trigger_death_particles, add_exp): ...
8
9     def import_graphics(self, name): ...
10
11     def get_player_distance_direction(self, player): ...
12
13     def get_status(self, player): ...
14
15     def actions(self, player): ...
16
17     def animate(self): ...
18
19     def cooldowns(self): ...
20
21     def get_damage(self, player, attack_type): ...
22
23     def check_death(self): ...
24
25     def hit_reaction(self): ...
26
27     def update(self):
28         self.hit_reaction()
29         self.move(self.speed)
30         self.animate()
31         self.cooldowns()
32         self.check_death()
33
34     def enemy_update(self, player):
35         self.get_status(player)
36         self.actions(player)
```

Image 5: Entity Class

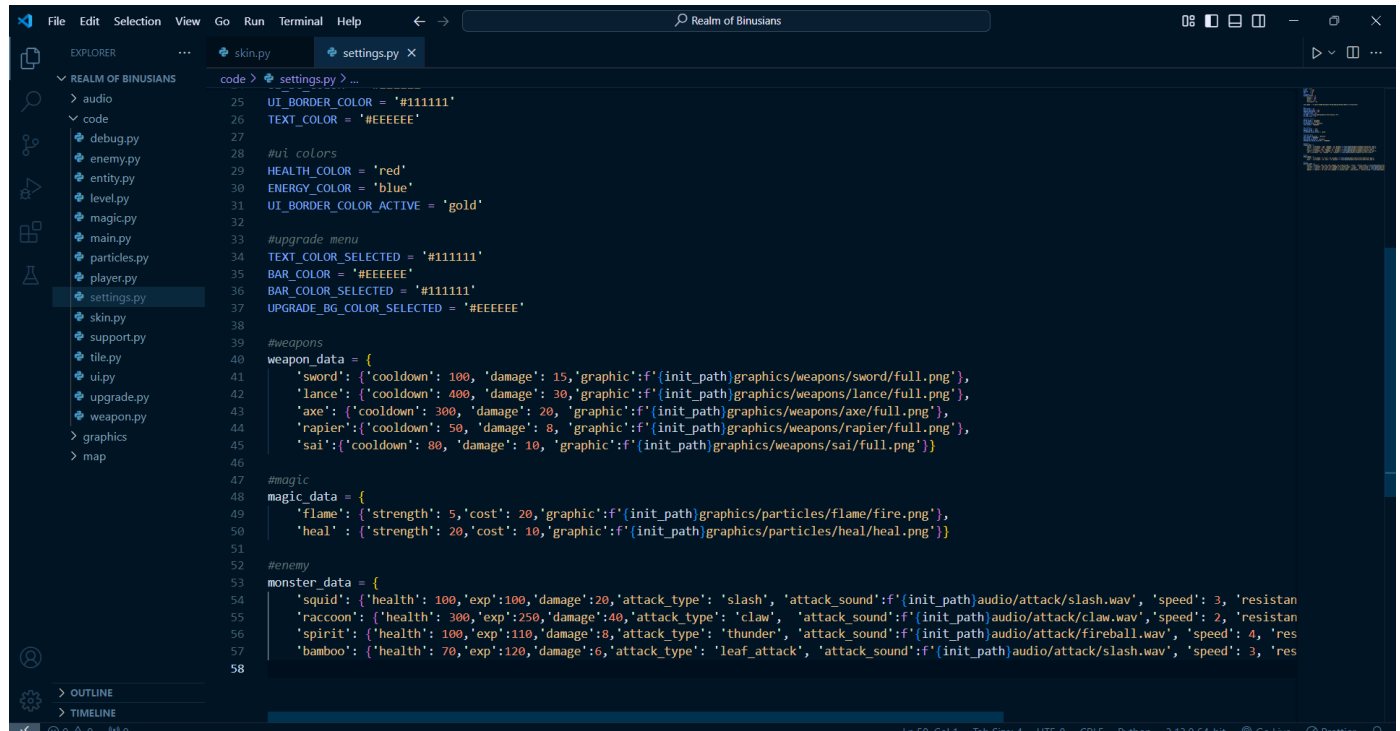


The screenshot shows a code editor with the following components:

- Explorer Panel:** Lists files in a project named 'REALM ...'. The 'code' folder is expanded, showing files like audio, debug.py, enemy.py, **entity.py** (selected), level.py, magic.py, main.py, particles.py, player.py, settings.py, skin.py, support.py, tile.py, ui.py, upgrade.py, and weapon.py. There are also folders for 'graphics' and 'map'.
- Code Editor:** Displays the content of 'entity.py'. The code defines a class 'Entity' that inherits from 'pygame.sprite.Sprite'. It includes methods for initialization, movement, collision detection, and wave value calculation.
- Terminal Panel:** Shows the command 'code > entity.py > () pygame'.
- Status Bar:** Indicates the current line and column (Ln 1, Col 1), encoding (UTF-8), line endings (CRLF), and the Python interpreter path.

```
4 class Entity(pygame.sprite.Sprite):
5     def __init__(self, groups):
6         super().__init__(groups)
7         self.frame_index = 0
8         self.animation_speed = 0.15
9         self.direction = pygame.math.Vector2()
10
11     def move(self, speed):
12         if self.direction.magnitude() != 0:
13             self.direction = self.direction.normalize()
14
15         self.hitbox.x += self.direction.x * speed
16         self.collision('horizontal')
17         self.hitbox.y += self.direction.y * speed
18         self.collision('vertical')
19         self.rect.center = self.hitbox.center
20
21     def collision(self, direction):
22         if direction == 'horizontal':
23             for sprite in self.obstacle_sprites:
24                 if sprite.hitbox.colliderect(self.hitbox):
25                     if self.direction.x > 0: # moving right
26                         self.hitbox.right = sprite.hitbox.left
27                     if self.direction.x < 0: # moving left
28                         self.hitbox.left = sprite.hitbox.right
29
30             if direction == 'vertical':
31                 for sprite in self.obstacle_sprites:
32                     if sprite.hitbox.colliderect(self.hitbox):
33                         if self.direction.y > 0: # moving down
34                             self.hitbox.bottom = sprite.hitbox.top
35                         if self.direction.y < 0: # moving up
36                             self.hitbox.top = sprite.hitbox.bottom
37
38     def wave_value(self):
39         value = sin(pygame.time.get_ticks())
40         if value >= 0:
```

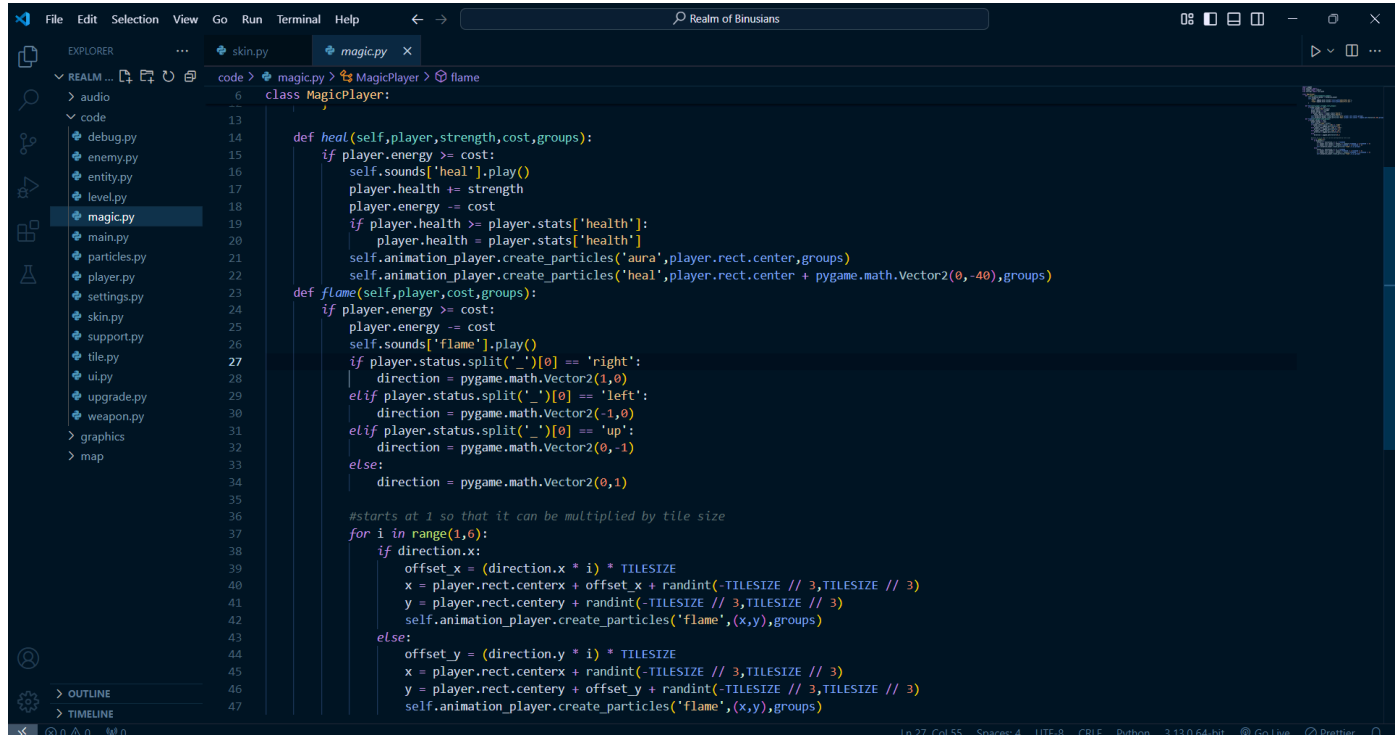
Image 6: Settings



The screenshot shows a code editor with the file explorer on the left displaying a project structure for 'REALM OF BINUSIANS'. The 'code' folder contains various Python files, with 'settings.py' selected. The main editor area shows the contents of 'settings.py', which defines various game settings and data structures. The settings include colors for UI, health, and energy, as well as weapon and magic data. The monster data section defines several enemy types with their respective stats and attack types.

```
25 UI_BORDER_COLOR = '#111111'
26 TEXT_COLOR = '#EEEEEE'
27
28 #ui colors
29 HEALTH_COLOR = 'red'
30 ENERGY_COLOR = 'blue'
31 UI_BORDER_COLOR_ACTIVE = 'gold'
32
33 #upgrade menu
34 TEXT_COLOR_SELECTED = '#111111'
35 BAR_COLOR = '#EEEEEE'
36 BAR_COLOR_SELECTED = '#111111'
37 UPGRADE_BG_COLOR_SELECTED = '#EEEEEE'
38
39 #weapons
40 weapon_data = {
41     'sword': {'cooldown': 100, 'damage': 15, 'graphic': f'{init_path}graphics/weapons/sword/full.png'},
42     'lance': {'cooldown': 400, 'damage': 30, 'graphic': f'{init_path}graphics/weapons/lance/full.png'},
43     'axe': {'cooldown': 300, 'damage': 20, 'graphic': f'{init_path}graphics/weapons/axe/full.png'},
44     'rapier': {'cooldown': 50, 'damage': 8, 'graphic': f'{init_path}graphics/weapons/rapier/full.png'},
45     'sai': {'cooldown': 80, 'damage': 10, 'graphic': f'{init_path}graphics/weapons/sai/full.png'}}
46
47 #magic
48 magic_data = {
49     'flame': {'strength': 5, 'cost': 20, 'graphic': f'{init_path}graphics/particles/flame/fire.png'},
50     'heal': {'strength': 20, 'cost': 10, 'graphic': f'{init_path}graphics/particles/heal/heal.png'}}
51
52 #enemy
53 monster_data = {
54     'squid': {'health': 100, 'exp': 100, 'damage': 20, 'attack_type': 'slash', 'attack_sound': f'{init_path}audio/attack/slash.wav', 'speed': 3, 'resistan
55     'raccoon': {'health': 300, 'exp': 250, 'damage': 40, 'attack_type': 'claw', 'attack_sound': f'{init_path}audio/attack/claw.wav', 'speed': 2, 'resistan
56     'spirit': {'health': 100, 'exp': 110, 'damage': 8, 'attack_type': 'thunder', 'attack_sound': f'{init_path}audio/attack/fireball.wav', 'speed': 4, 'res
57     'bamboo': {'health': 70, 'exp': 120, 'damage': 6, 'attack_type': 'leaf_attack', 'attack_sound': f'{init_path}audio/attack/slash.wav', 'speed': 3, 'res
58
```

Image 7: Player's magic attack



The screenshot shows a code editor with the file explorer on the left displaying a project structure for 'REALM OF BINUSIANS'. The 'code' folder contains various Python files, with 'magic.py' selected. The main editor area shows the contents of 'magic.py', which defines the 'MagicPlayer' class. The class has two methods: 'heal' and 'flame'. The 'heal' method checks if the player has enough energy to heal, plays a sound, and increases the player's health. The 'flame' method checks if the player has enough energy to cast a flame attack, plays a sound, and creates a flame particle at the player's position. The 'flame' method also includes a loop to create multiple flame particles in the direction of the attack.

```
6 class MagicPlayer:
13
14     def heal(self, player, strength, cost, groups):
15         if player.energy >= cost:
16             self.sounds['heal'].play()
17             player.health += strength
18             player.energy -= cost
19             if player.health >= player.stats['health']:
20                 player.health = player.stats['health']
21             self.animation_player.create_particles('aura', player.rect.center, groups)
22             self.animation_player.create_particles('heal', player.rect.center + pygame.math.Vector2(0, -40), groups)
23
24     def flame(self, player, cost, groups):
25         if player.energy >= cost:
26             player.energy -= cost
27             self.sounds['flame'].play()
28             if player.status.split('.')[0] == 'right':
29                 direction = pygame.math.Vector2(1, 0)
30             elif player.status.split('.')[0] == 'left':
31                 direction = pygame.math.Vector2(-1, 0)
32             elif player.status.split('.')[0] == 'up':
33                 direction = pygame.math.Vector2(0, -1)
34             else:
35                 direction = pygame.math.Vector2(0, 1)
36
37             #starts at 1 so that it can be multiplied by tile size
38             for i in range(1, 6):
39                 if direction.x:
40                     offset_x = (direction.x * i) * TILESIZE
41                     x = player.rect.centerx + offset_x + randint(-TILESIZE // 3, TILESIZE // 3)
42                     y = player.rect.centery + randint(-TILESIZE // 3, TILESIZE // 3)
43                     self.animation_player.create_particles('flame', (x, y), groups)
44                 else:
45                     offset_y = (direction.y * i) * TILESIZE
46                     x = player.rect.centerx + randint(-TILESIZE // 3, TILESIZE // 3)
47                     y = player.rect.centery + offset_y + randint(-TILESIZE // 3, TILESIZE // 3)
48                     self.animation_player.create_particles('flame', (x, y), groups)
```

Image 8: Particle animation

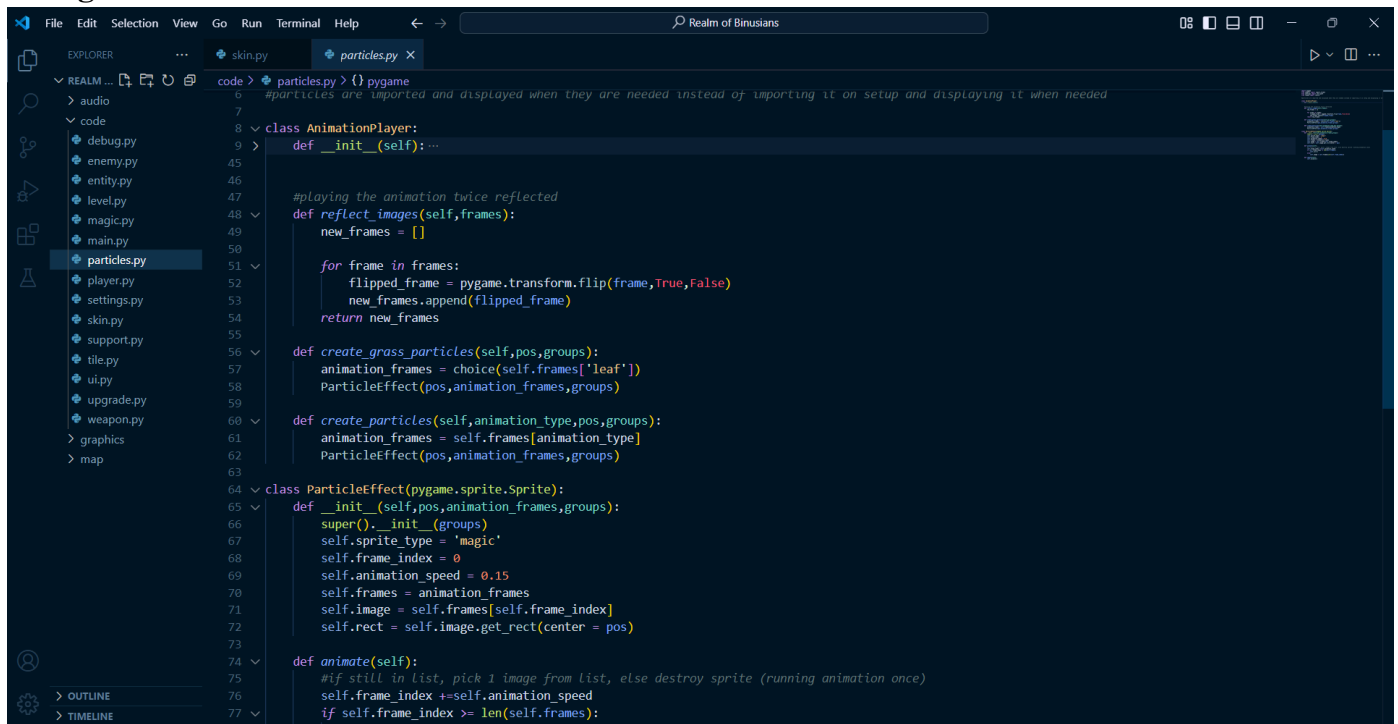


Image 9: support.py to import game map

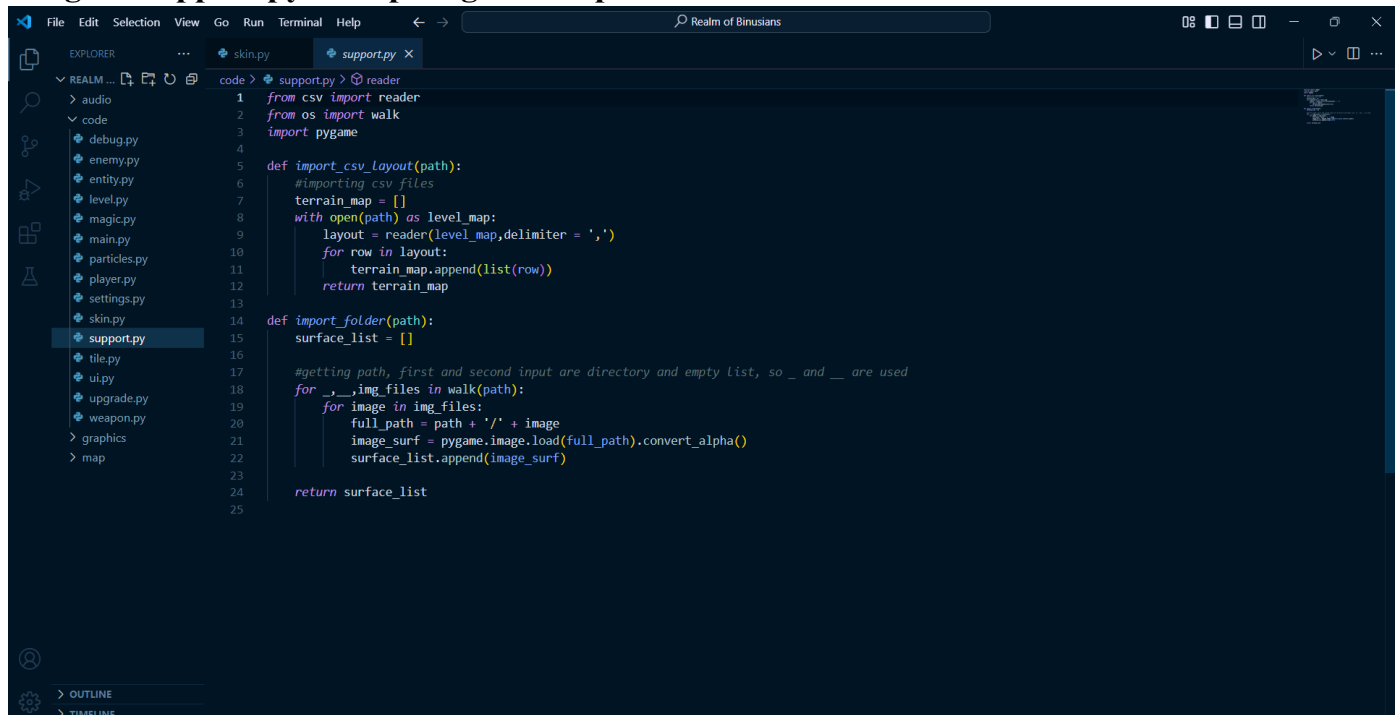


Image 10: Tile.py to set hitbox and offset of each tile

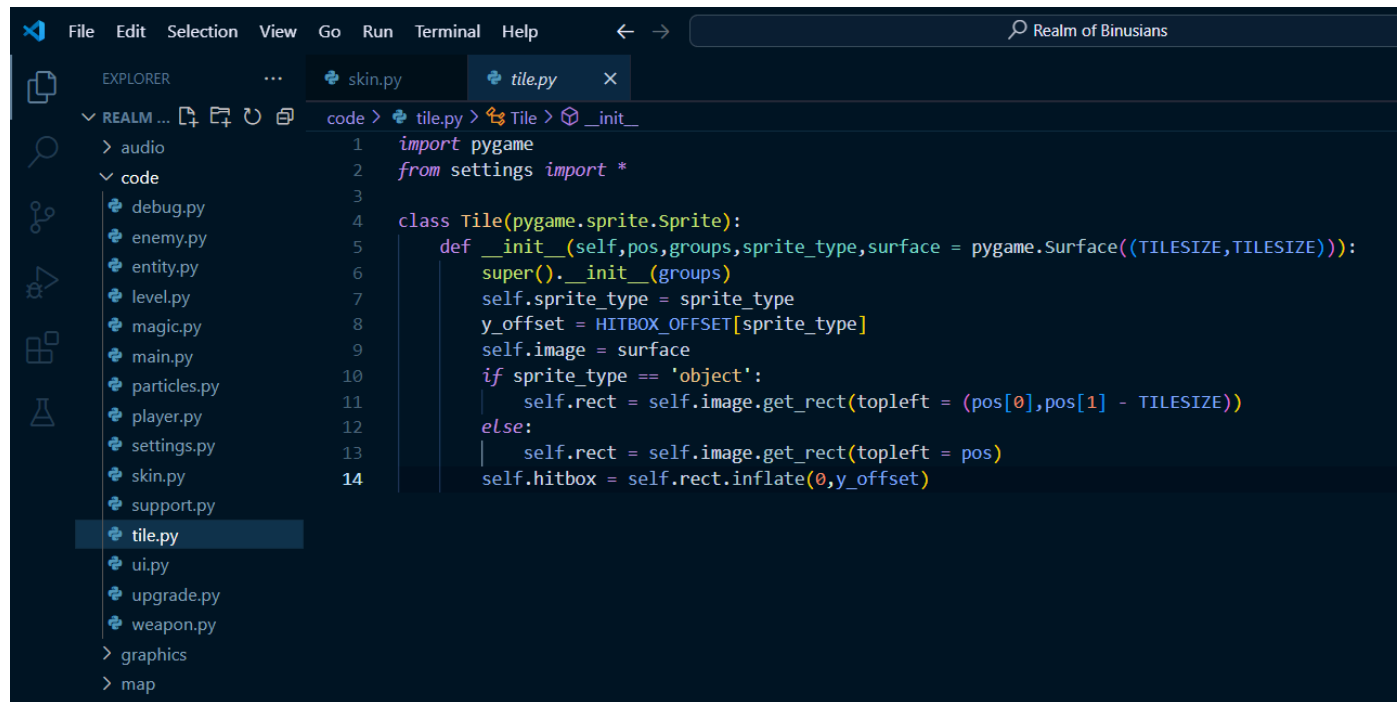


Image 11: Weapon section (displays the weapon)

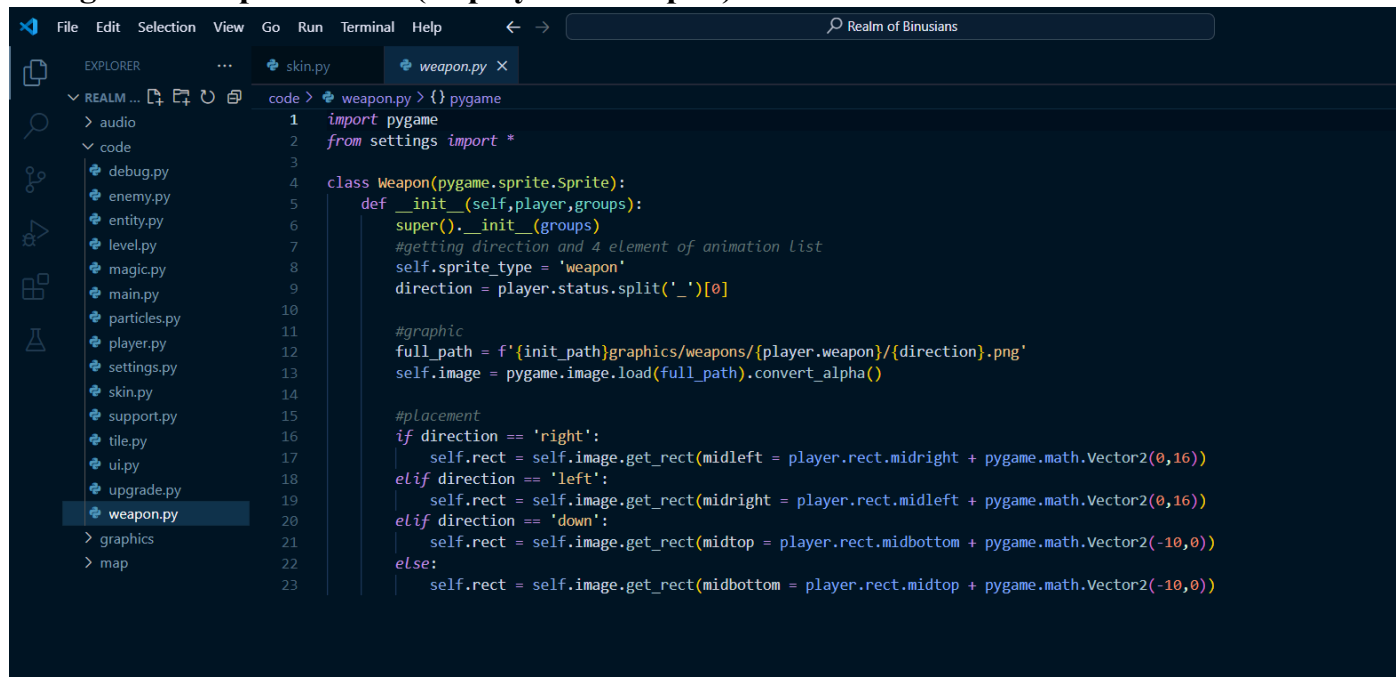
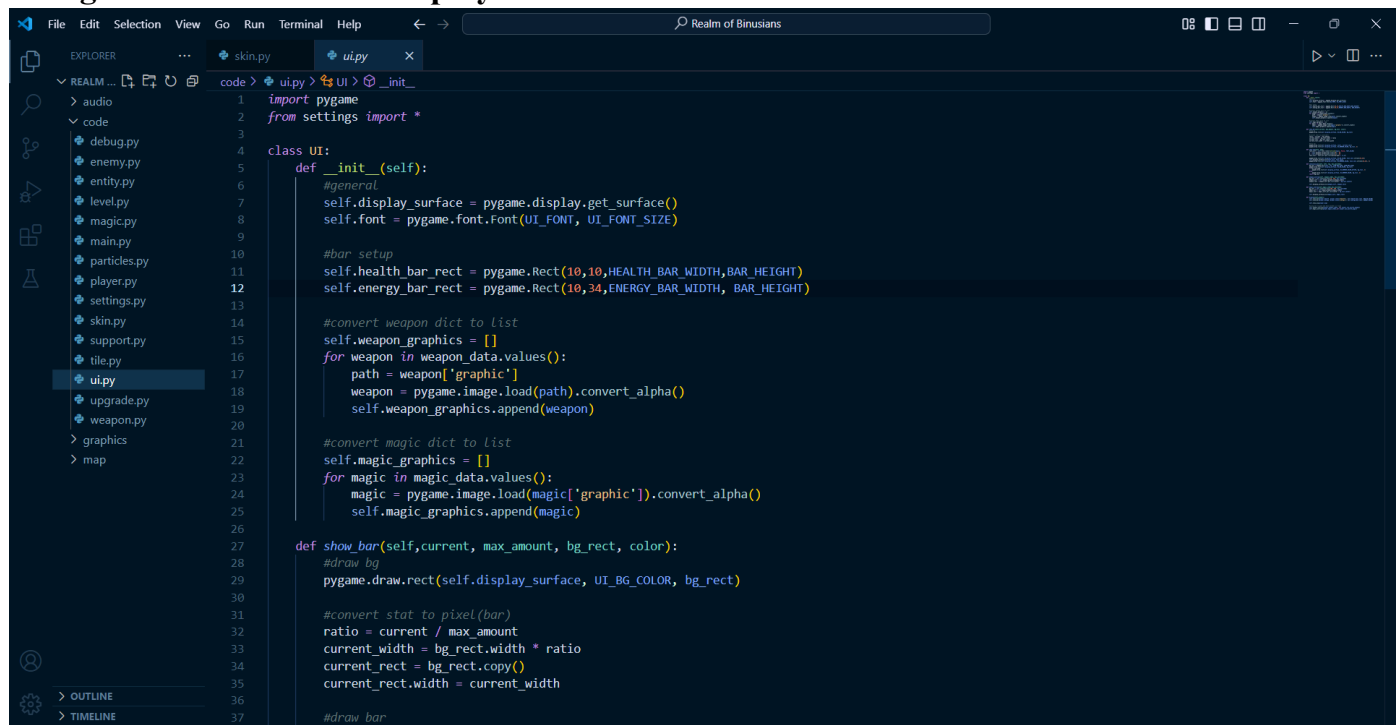
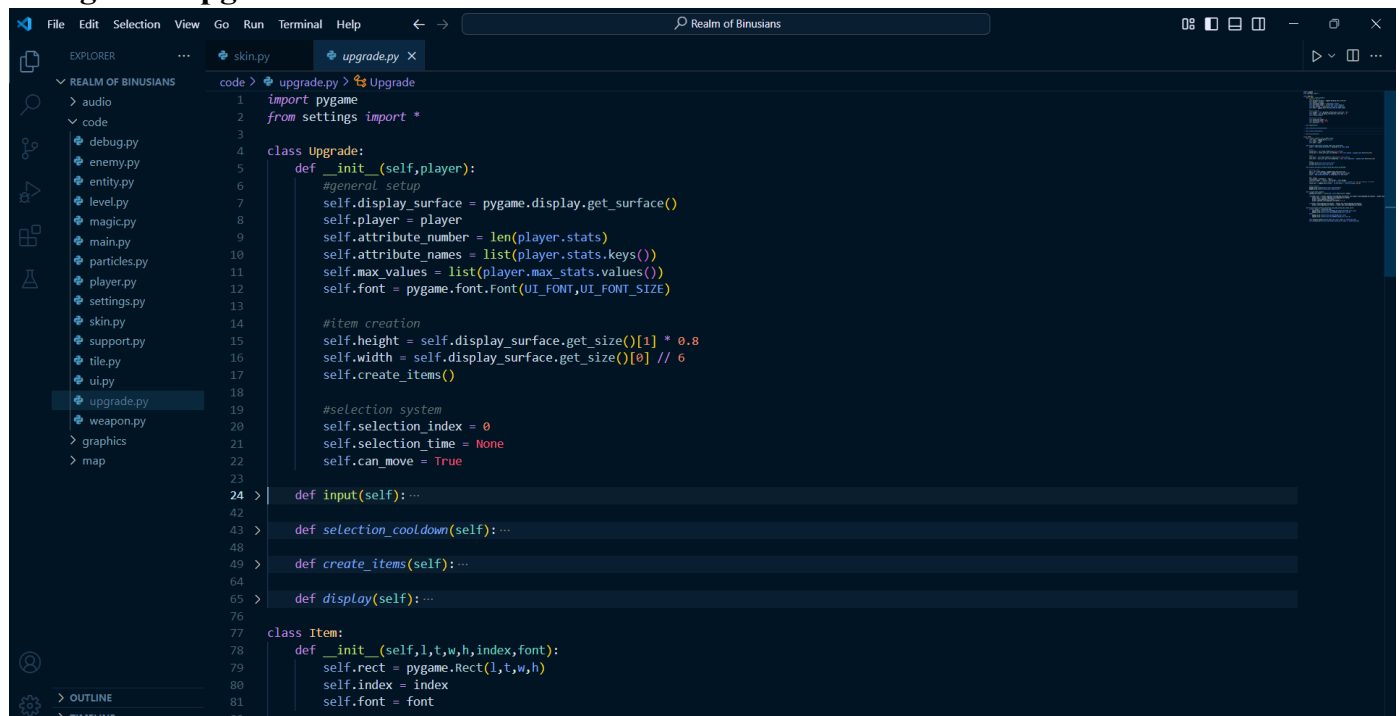


Image 12: User Interface display



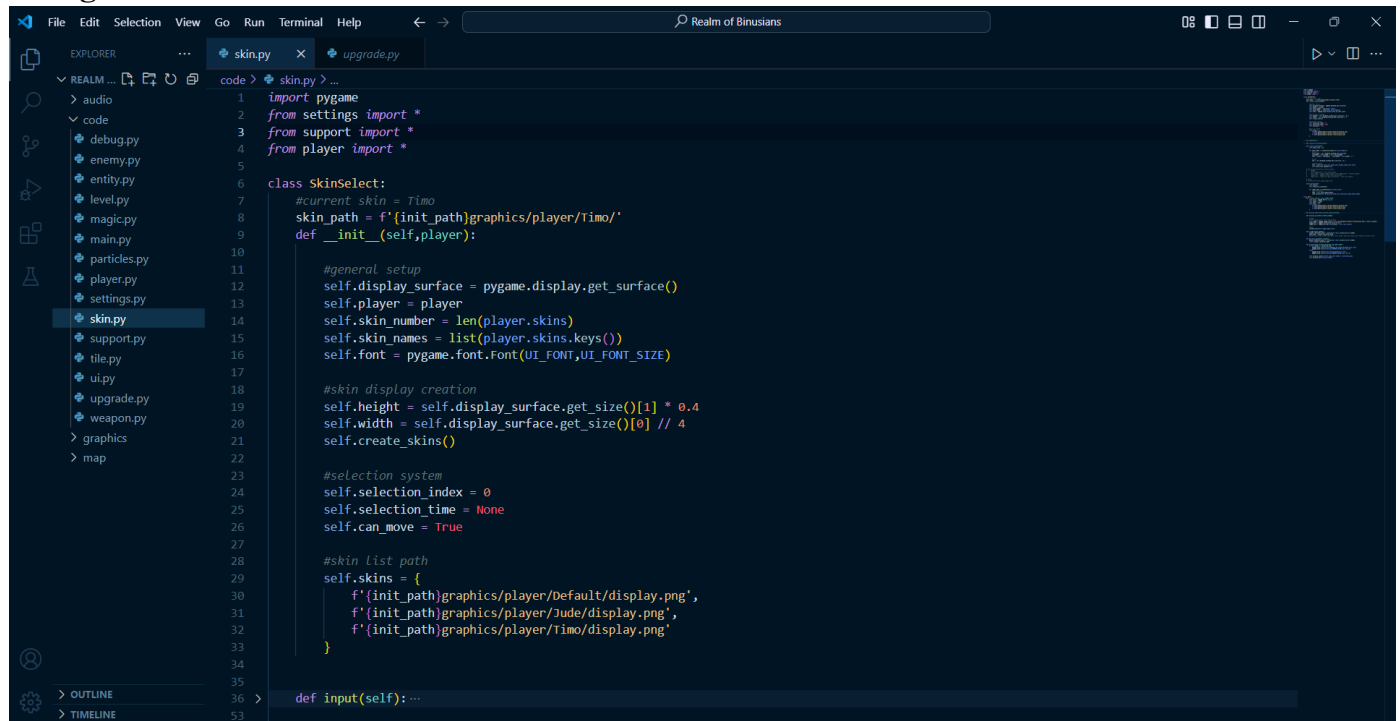
```
1 import pygame
2 from settings import *
3
4 class UI:
5     def __init__(self):
6         #general
7         self.display_surface = pygame.display.get_surface()
8         self.font = pygame.font.Font(UI_FONT, UI_FONT_SIZE)
9
10        #bar setup
11        self.health_bar_rect = pygame.Rect(10,10,HEALTH_BAR_WIDTH,BAR_HEIGHT)
12        self.energy_bar_rect = pygame.Rect(10,34,ENERGY_BAR_WIDTH, BAR_HEIGHT)
13
14        #convert weapon dict to list
15        self.weapon_graphics = []
16        for weapon in weapon_data.values():
17            path = weapon['graphic']
18            weapon = pygame.image.load(path).convert_alpha()
19            self.weapon_graphics.append(weapon)
20
21        #convert magic dict to list
22        self.magic_graphics = []
23        for magic in magic_data.values():
24            magic = pygame.image.load(magic['graphic']).convert_alpha()
25            self.magic_graphics.append(magic)
26
27    def show_bar(self,current, max_amount, bg_rect, color):
28        #draw bg
29        pygame.draw.rect(self.display_surface, UI_BG_COLOR, bg_rect)
30
31        #convert stat to pixel(bar)
32        ratio = current / max_amount
33        current_width = bg_rect.width * ratio
34        current_rect = bg_rect.copy()
35        current_rect.width = current_width
36
37        #draw bar
```

Image 12: Upgrade Menu



```
1 import pygame
2 from settings import *
3
4 class Upgrade:
5     def __init__(self,player):
6         #general setup
7         self.display_surface = pygame.display.get_surface()
8         self.player = player
9         self.attribute_number = len(player.stats)
10        self.attribute_names = list(player.stats.keys())
11        self.max_values = list(player.max_stats.values())
12        self.font = pygame.font.Font(UI_FONT,UI_FONT_SIZE)
13
14        #item creation
15        self.height = self.display_surface.get_size()[1] * 0.8
16        self.width = self.display_surface.get_size()[0] // 6
17        self.create_items()
18
19        #selection system
20        self.selection_index = 0
21        self.selection_time = None
22        self.can_move = True
23
24    def input(self):...
25
26    def selection_cooldown(self):...
27
28    def create_items(self):...
29
30    def display(self):...
31
32 class Item:
33     def __init__(self,l,t,w,h,index,font):
34         self.rect = pygame.Rect(l,t,w,h)
35         self.index = index
36         self.font = font
```

Image 14: Skin Menu



The screenshot shows a code editor with a dark theme. The Explorer panel on the left shows a project structure with folders 'audio' and 'code', and a file 'skin.py' selected. The main editor displays the code for 'skin.py'. The code includes imports for pygame, settings, support, and player. It defines a 'SkinSelect' class with methods for initialization, general setup, skin display creation, and a selection system. The 'skins' list contains three entries: 'Default/display.png', 'Jude/display.png', and 'Timo/display.png'. The 'input' method is partially visible at the bottom.

```
1 import pygame
2 from settings import *
3 from support import *
4 from player import *
5
6 class SkinSelect:
7     #current skin = Timo
8     skin_path = f'{init_path}graphics/player/Timo/'
9     def __init__(self,player):
10
11         #general setup
12         self.display_surface = pygame.display.get_surface()
13         self.player = player
14         self.skin_number = len(player.skins)
15         self.skin_names = list(player.skins.keys())
16         self.font = pygame.font.Font(UI_FONT,UI_FONT_SIZE)
17
18         #skin display creation
19         self.height = self.display_surface.get_size()[1] * 0.4
20         self.width = self.display_surface.get_size()[0] // 4
21         self.create_skins()
22
23         #selection system
24         self.selection_index = 0
25         self.selection_time = None
26         self.can_move = True
27
28         #skin list path
29         self.skins = {
30             f'{init_path}graphics/player/Default/display.png',
31             f'{init_path}graphics/player/Jude/display.png',
32             f'{init_path}graphics/player/Timo/display.png'
33         }
34
35     def input(self): ...
```

Image 15: Final Game

