Algorithm and Programming

Final Project Report

Student Information:

Name: Gabriel Anderson **Student Id:** 2702256315

Course Code: COMP6047001 **Course Name:** Algorithm and Programming

Class: L1AC **Lecturer:** Jude Joseph Lamug Martinez,MCS

Type of Assignment: Final Project report

Table of Contents

Table of Contents

A. **Introduction**

B. **Project Specification**

C. **Solution Design**

D. **Algorithms**

E. **Evidence of working program**

F. **Lesson Learned**

G. **Resources**

A. Introduction

Background

Students are expected to create a program using what was taught in Algorithm and Programming course and Students are also expected to use concept that was beyond what was taught in the Algorithm and Programming course

After some research, I decided to make a game named “Heart Eater” which is a platformer game by using unfamiliar functions from pygame library which was not covered in the Algorithm and Programming course.

B. Project Specification

All of the image were not made by me

1. Game Overview

Title: Heart Eater
Genre: 2D Platformer
Platform: Developed using Pygame
Objective: Create an engaging and challenging platformer game with a focus on player controls, level design, and visual appeal where users must complete levels

2. Game Components

Player Character

Animated character with idle, walking, and jumping sprites.
Responsive controls for movement and jumping.
Gravity-based physics for realistic jumping and falling.

Obstacles

There are multiple platforms for the player to navigate through the level and obstacles such as lava, moving platforms and enemies

User interface

Score display

Level Progression information

Audio

Background music

Sound effects for player actions (jumping, eating heart, dies)

3. Game Mechanics

Players are able to control the character by using keys

“A” for moving to the left

“W” for jumping

“D” for moving to the right

Game Physics

Collision detection between objects in the game is by comparing the position of the rectangle of both objects. The game detects the collision and does different actions depending on the objects.

For example:

If the player collides with the ground or a platform from below, their vertical velocity is set to 0, and they are considered to be on the ground or if the player collides with the ground or a platform from above, their vertical velocity is set to 0, and they start falling.

If the player collides with enemies or lava, the game ends

4. Game Libraries/Modules

Pygame - Pygame is a free and open-source cross-platform library for the development of multimedia applications like video games using Python. It uses the Simple DirectMedia Layer library and several other popular libraries to abstract the most common functions, making writing these programs a more intuitive task.

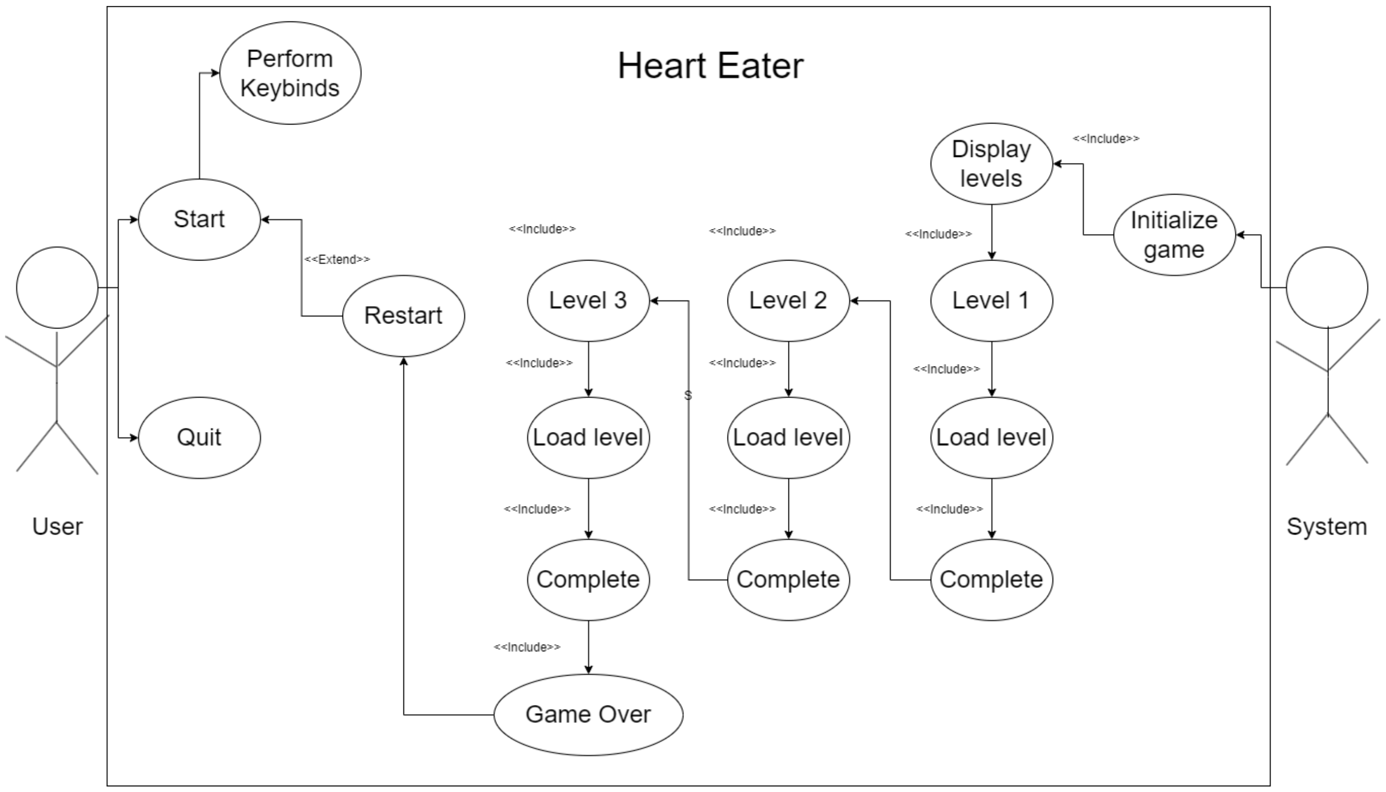
Mixer - mixer pygame module is used for loading and playing sounds that are available and initialised before using it.

Sys - to terminate the game when the user exits

C. Solution Design

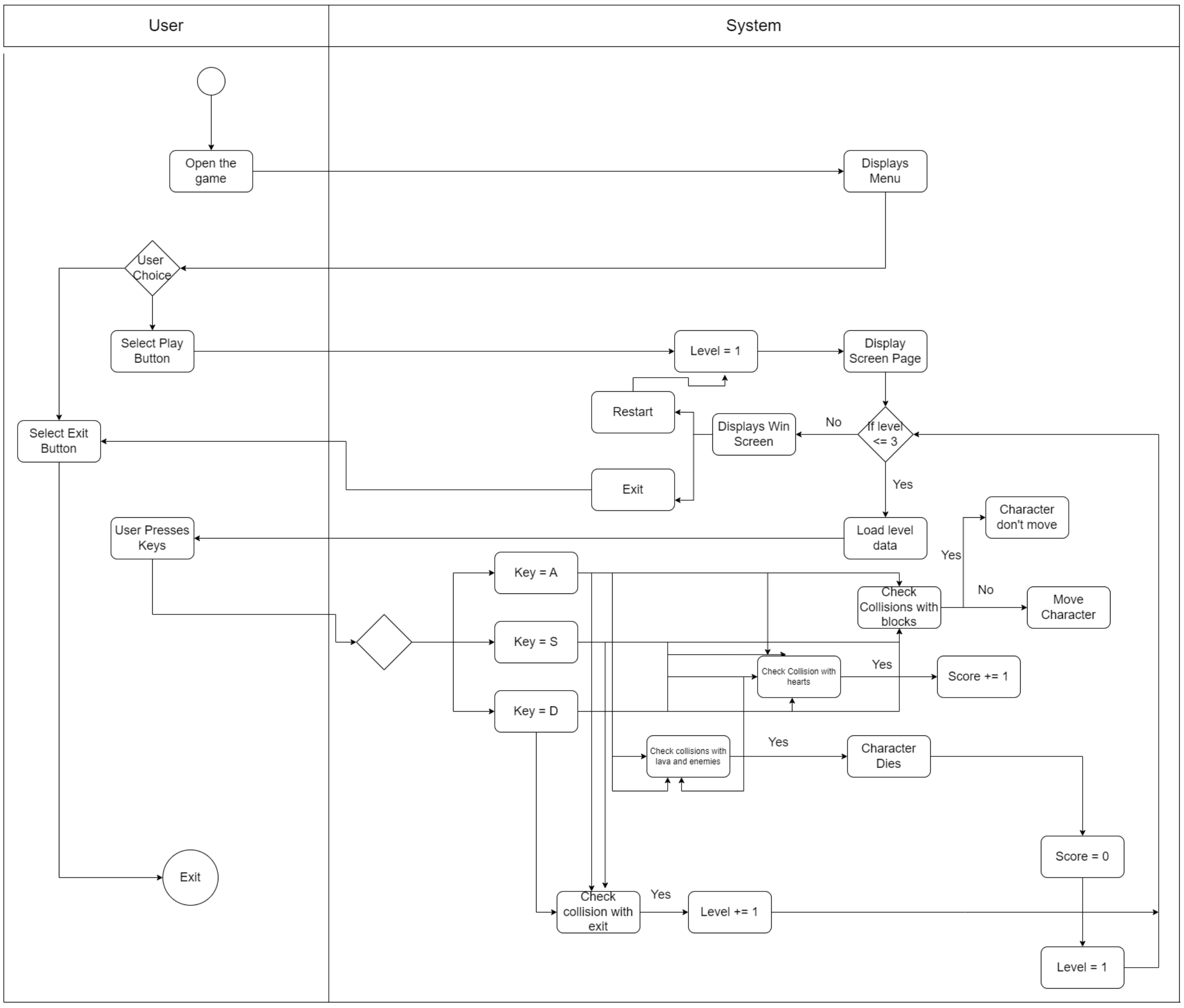
Use Case Diagram

The Use Case Diagram summarizes the action of the user and the system within the game

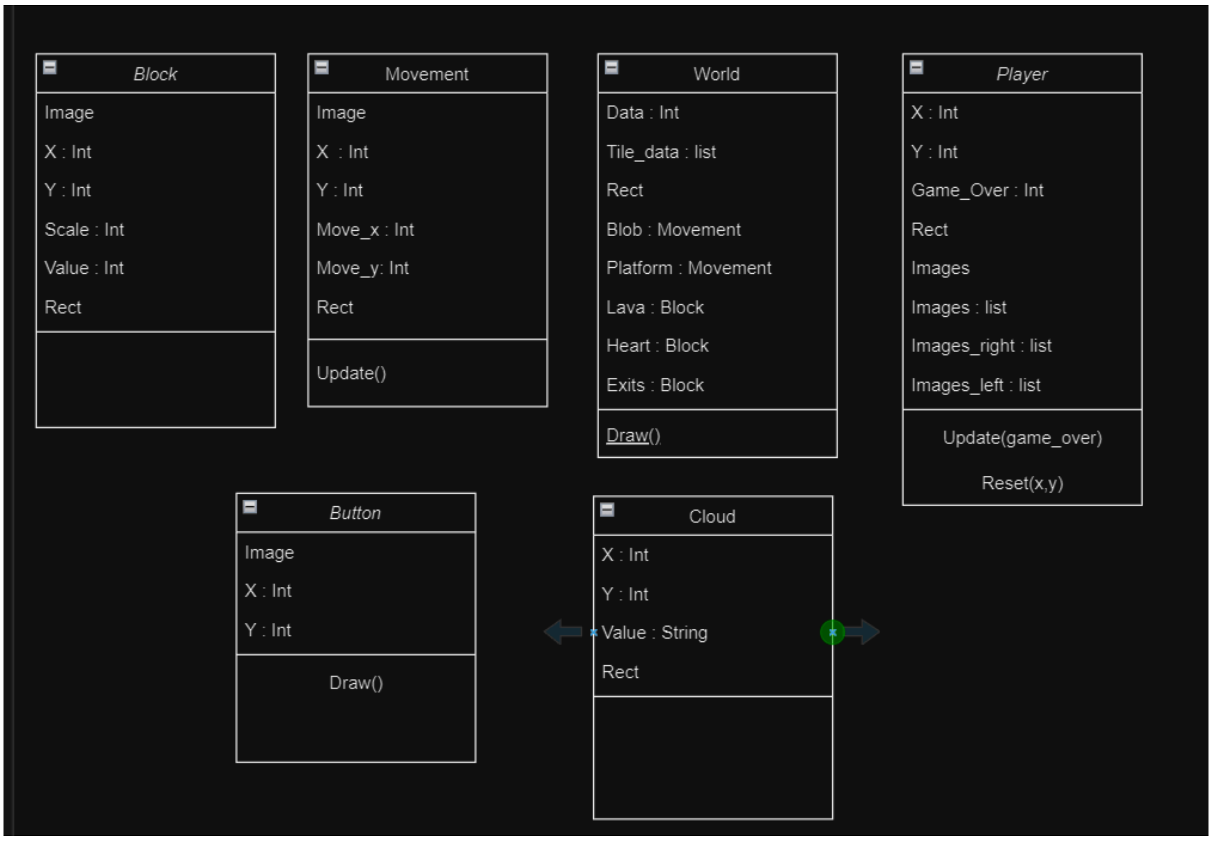


Activity Diagram

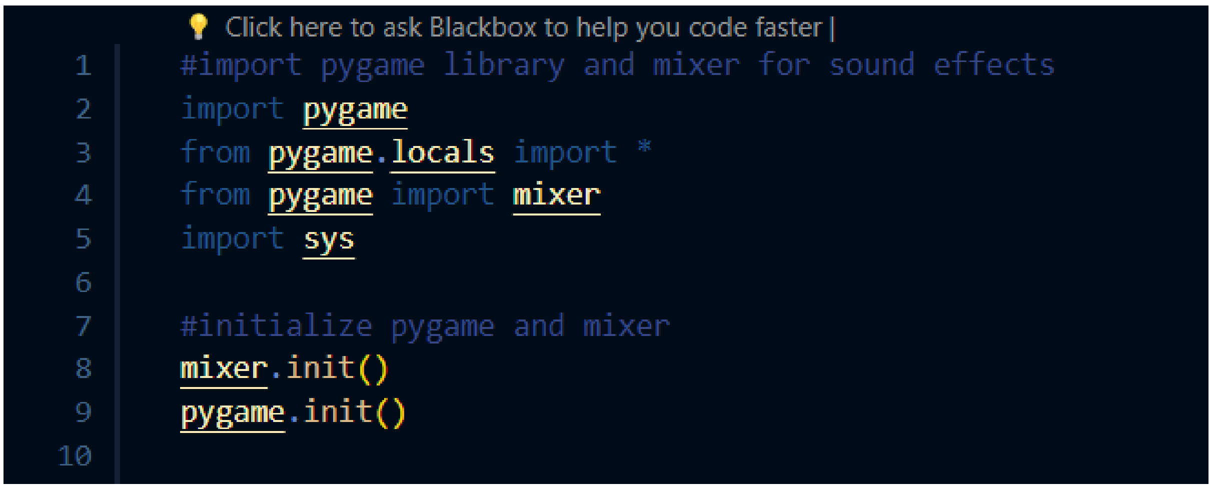
The Activity Diagram shows flow of activity of one activity to another within the system



Class Diagram
The Class Diagram shows the structure of the classes that exists in the program

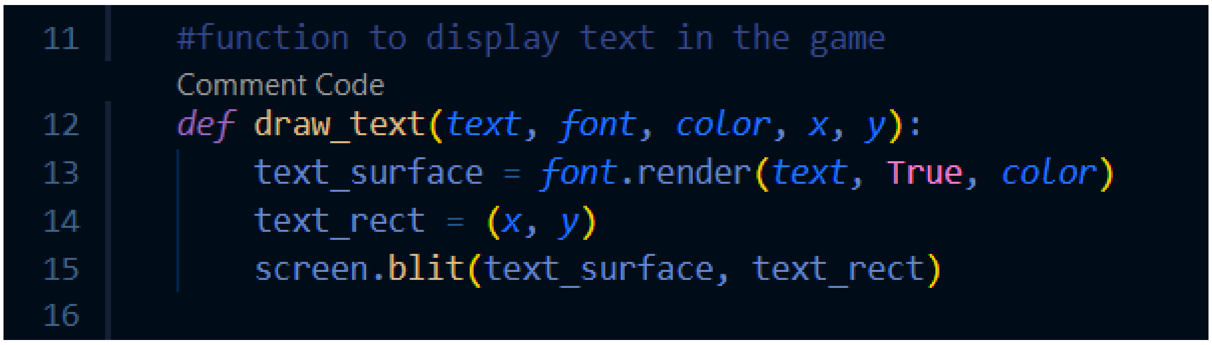


D. Algorithms


💡 Click here to ask Blackbox to help you code faster |

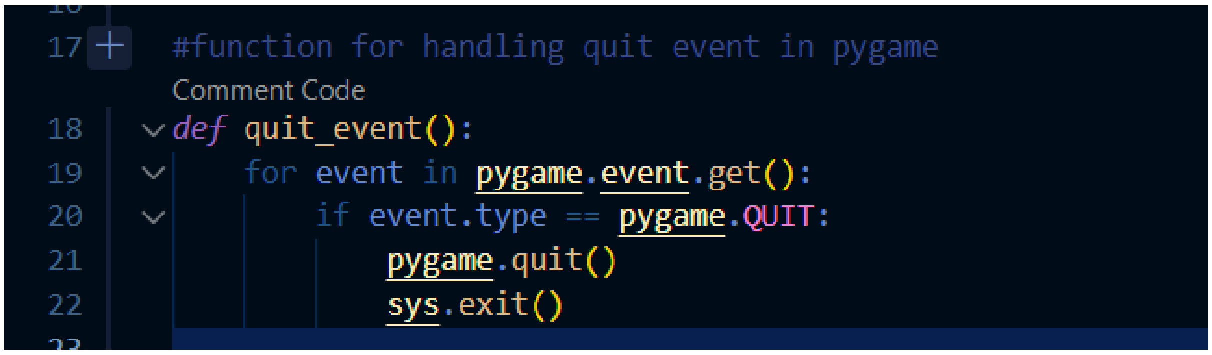
```
#import pygame library and mixer for sound effects
import pygame
from pygame.locals import *
from pygame import mixer
import sys

#initialize pygame and mixer
mixer.init()
pygame.init()

```

Imports and initialize pygame library and mixer module



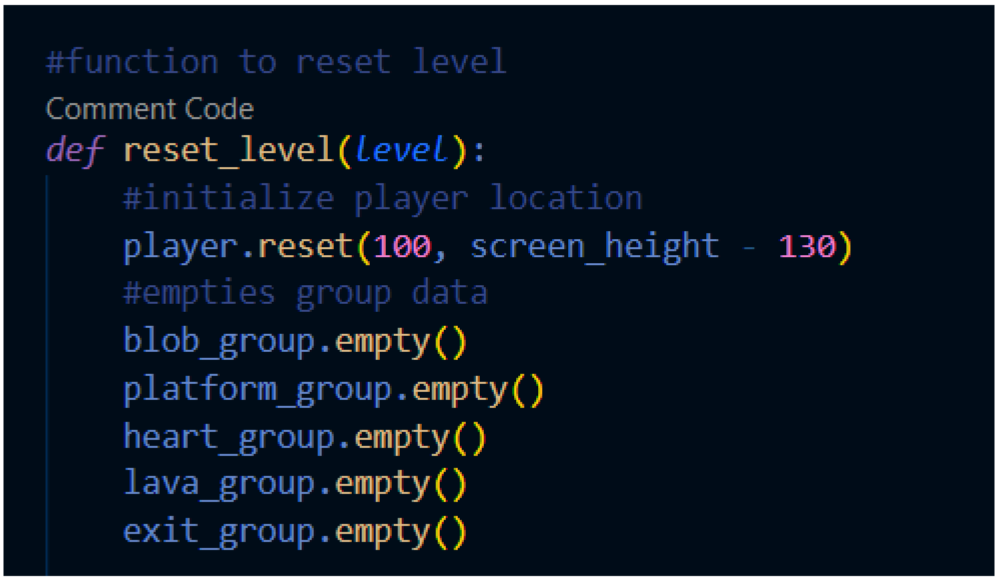
```
#function to display text in the game
Comment Code
def draw_text(text, font, color, x, y):
    text_surface = font.render(text, True, color)
    text_rect = (x, y)
    screen.blit(text_surface, text_rect)

```

Draw_text function is a function that receives 5 parameters which is (text, font, color, x, y)
text: The string of text that you want to display.
font: The font object used for rendering the text.
color: The color of the text.
x, y: The coordinates where the text will be drawn on the screen.
Draw_text function is used to draw text on the screen



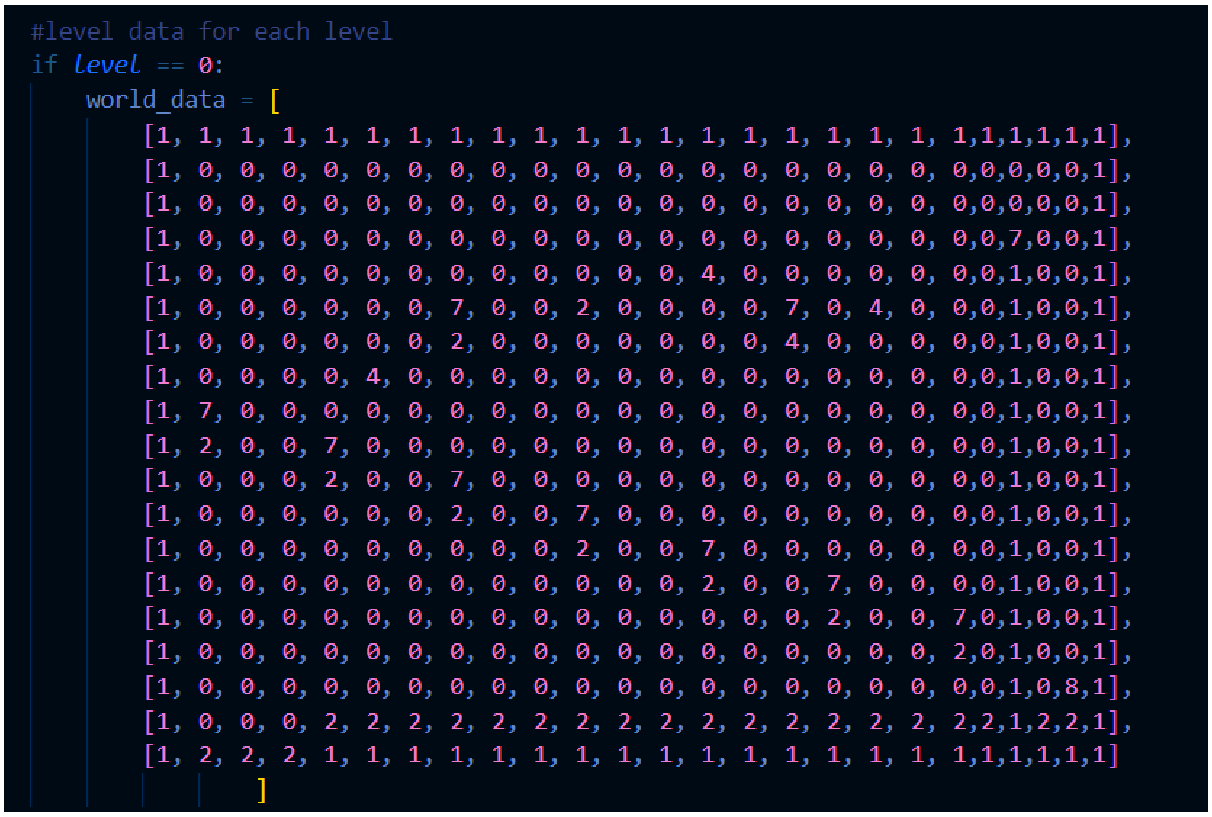
```
#function for handling quit event in pygame
Comment Code
def quit_event():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

Quit_event function is used to handle the quit event, ensuring that the game is properly closed when the user tries to exit it

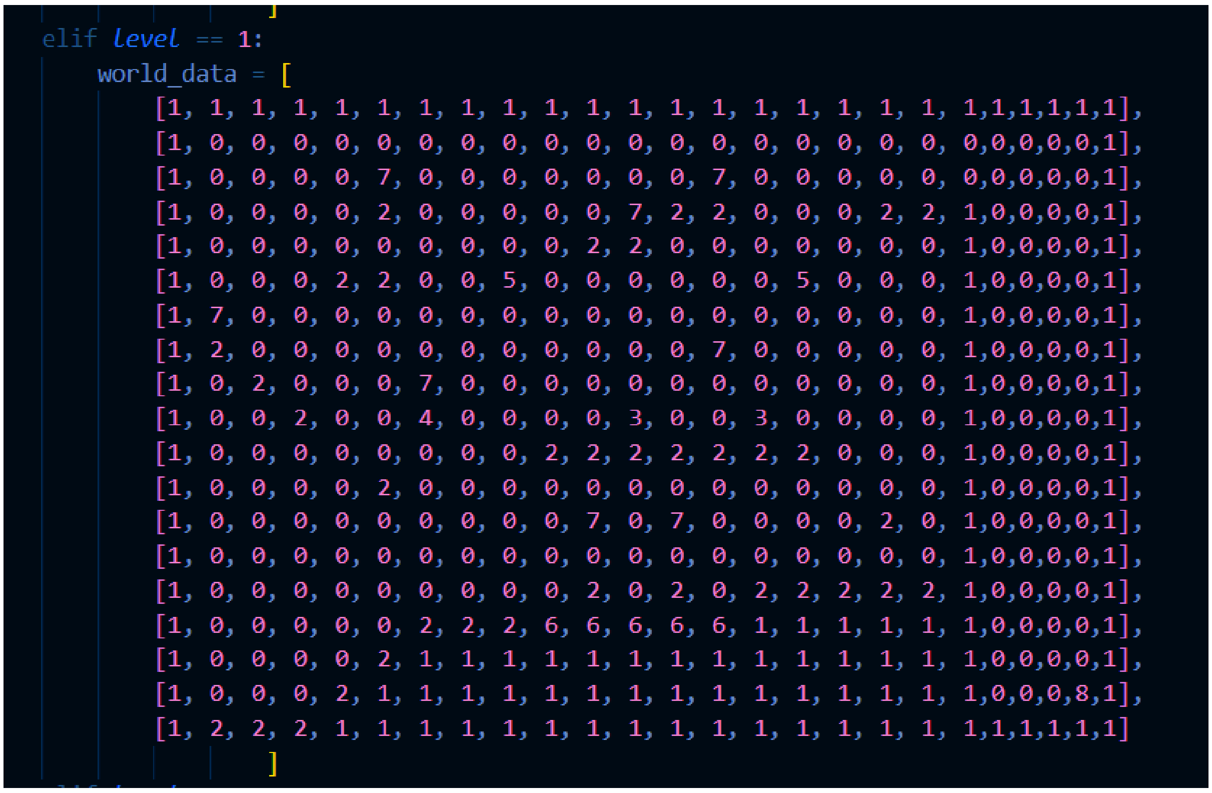


```python
#function to reset level
Comment Code
def reset_level(level):
    #initialize player location
    player.reset(100, screen_height - 130)
    #empties group data
    blob_group.empty()
    platform_group.empty()
    heart_group.empty()
    lava_group.empty()
    exit_group.empty()
```

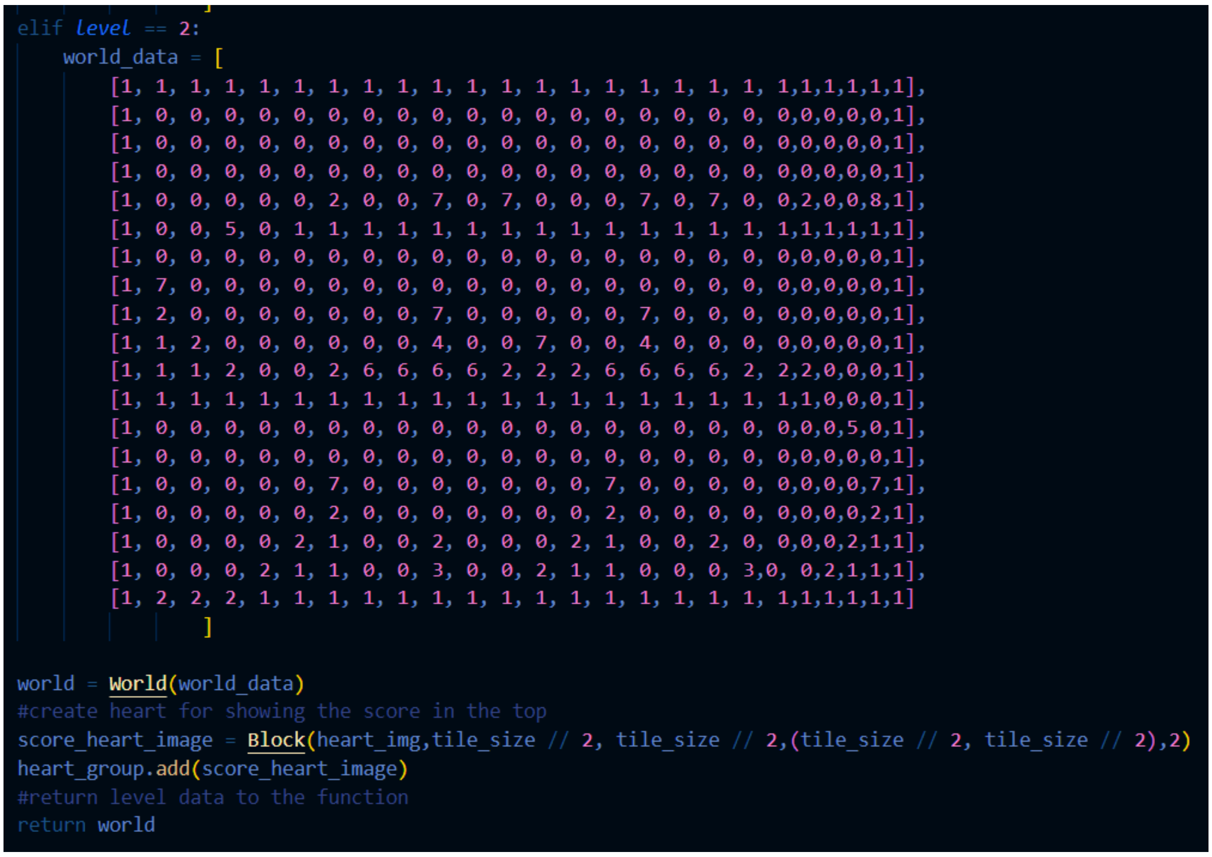
The reset_level function takes in level parameter and player.reset(100, screen_height - 130) is the initial position of the player location blob_group.empty(), platform_group.empty(), heart_group.empty(), lava_group.empty(), exit_group.empty() are used to remove all sprites from the respective groups



```python
#level data for each level
if level == 0:
    world_data = [
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,1,1,1,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,7,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 7, 0, 0, 2, 0, 0, 0, 0, 7, 0, 4, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,1,0,0,1],
        [1, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,1,0,0,1],
        [1, 2, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 2, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 7, 0, 0, 0, 0, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 7, 0, 0, 0,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 7,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2,0,1,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,1,0,8,1],
        [1, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,2,1,2,2,1],
        [1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,1,1,1,1]
            ]
```



```
elif level == 1:
    world_data = [
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,1,1,1,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 7, 2, 2, 0, 0, 0, 2, 2, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 1,0,0,0,0,1],
        [1, 0, 0, 0, 2, 2, 0, 0, 5, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 1,0,0,0,0,1],
        [1, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,0,0,0,0,1],
        [1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 1,0,0,0,0,1],
        [1, 0, 2, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,0,0,0,0,1],
        [1, 0, 0, 2, 0, 0, 4, 0, 0, 0, 0, 3, 0, 0, 3, 0, 0, 0, 0, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 7, 0, 0, 0, 0, 2, 0, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 2, 2, 2, 2, 2, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 2, 2, 2, 6, 6, 6, 6, 6, 1, 1, 1, 1, 1, 1,0,0,0,0,1],
        [1, 0, 0, 0, 0, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,0,0,0,0,1],
        [1, 0, 0, 0, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,0,0,0,8,1],
        [1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,1,1,1,1]
            ]
```



```
elif level == 2:
    world_data = [
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,1,1,1,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 2, 0, 0, 7, 0, 7, 0, 0, 0, 7, 0, 7, 0, 0,2,0,0,8,1],
        [1, 0, 0, 5, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,1,1,1,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 2, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0,0,0,0,0,1],
        [1, 1, 2, 0, 0, 0, 0, 0, 0, 4, 0, 0, 7, 0, 0, 4, 0, 0, 0, 0,0,0,0,0,1],
        [1, 1, 1, 2, 0, 0, 2, 6, 6, 6, 6, 2, 2, 2, 6, 6, 6, 6, 2, 2,2,0,0,0,1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,5,0,1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0,0,0,1],
        [1, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0,0,0,0,7,1],
        [1, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0,0,0,0,2,1],
        [1, 0, 0, 0, 0, 2, 1, 0, 0, 2, 0, 0, 0, 2, 1, 0, 0, 2, 0, 0,0,0,2,1,1],
        [1, 0, 0, 0, 2, 1, 1, 0, 0, 3, 0, 0, 2, 1, 1, 0, 0, 0, 3,0, 0,2,1,1,1],
        [1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,1,1,1,1,1]
            ]

world = World(world_data)
#create heart for showing the score in the top
score_heart_image = Block(heart_img,tile_size // 2, tile_size // 2,(tile_size // 2, tile_size // 2),2)
heart_group.add(score_heart_image)
#return level data to the function
return world
```

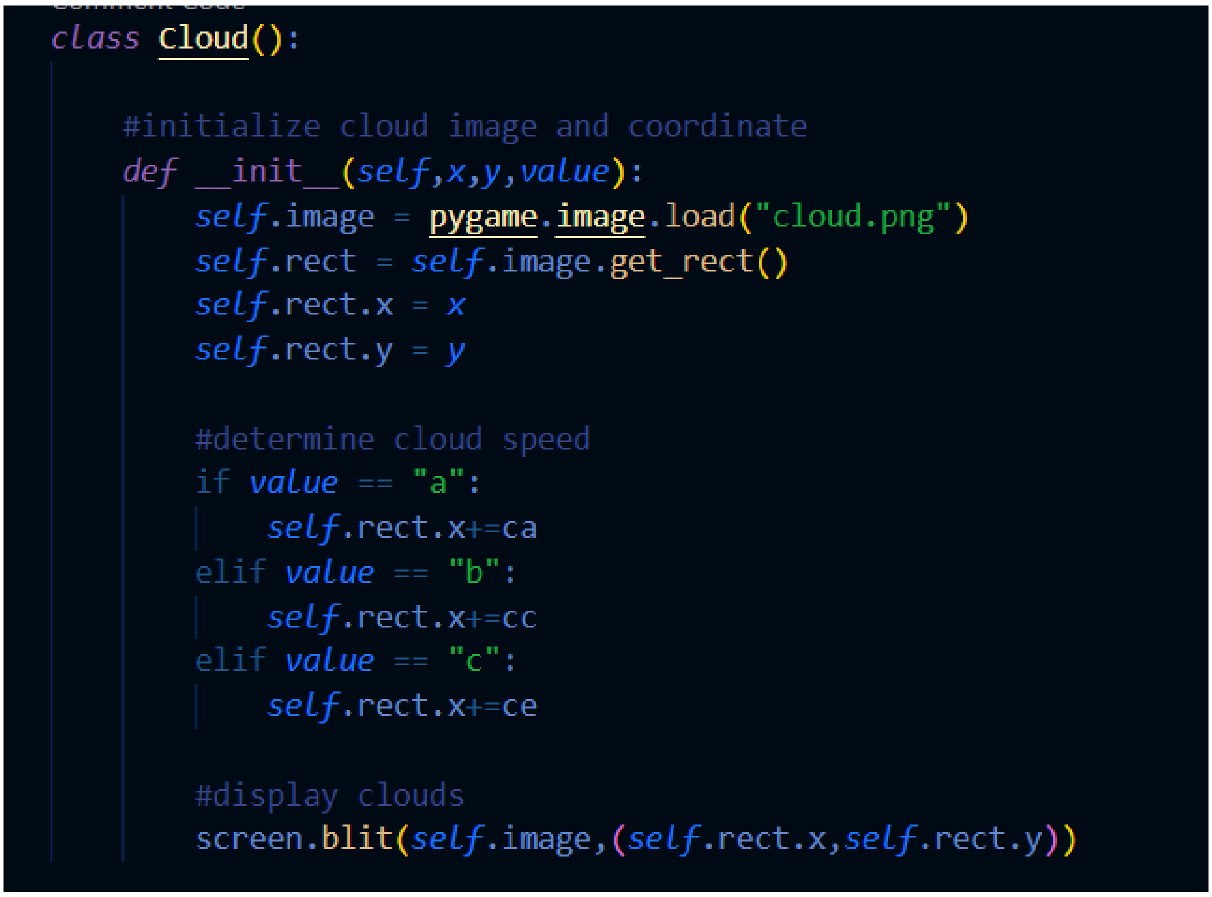
This code section defines the world data for the game where each number represents a specific tile in the game

0: Empty space

1: Dirt

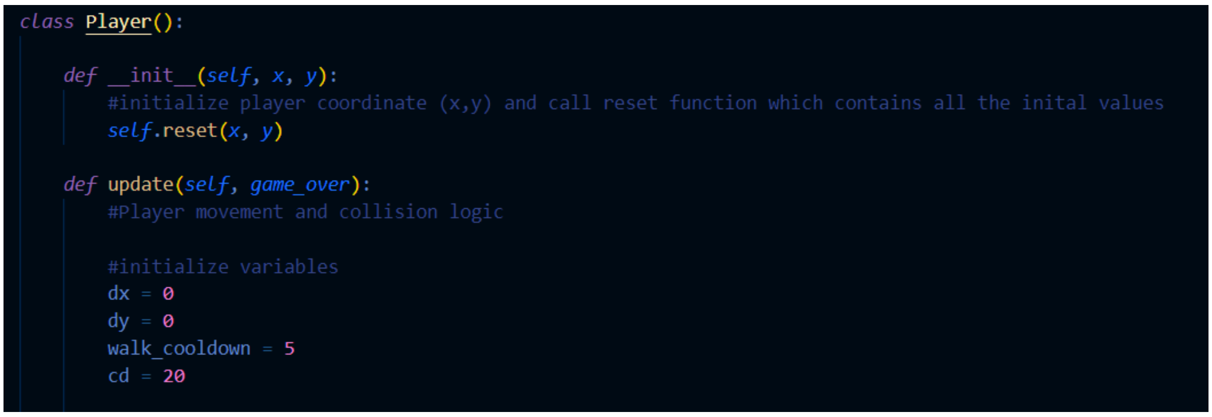
2: Grass
3: Enemy
4: Horizontal moving platform
5: Vertical moving platform
6: Lava
7: Heart
8: Exit
Score_heart_image is used to represent the score at the top of the screen and is added to the heart group

The return world line returns the world_data value depending on the levels



```python
class Cloud():

    #initialize cloud image and coordinate
    def __init__(self,x,y,value):
        self.image = pygame.image.load("cloud.png")
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y

        #determine cloud speed
        if value == "a":
            self.rect.x+=ca
        elif value == "b":
            self.rect.x+=cc
        elif value == "c":
            self.rect.x+=ce

        #display clouds
        screen.blit(self.image,(self.rect.x,self.rect.y))
```

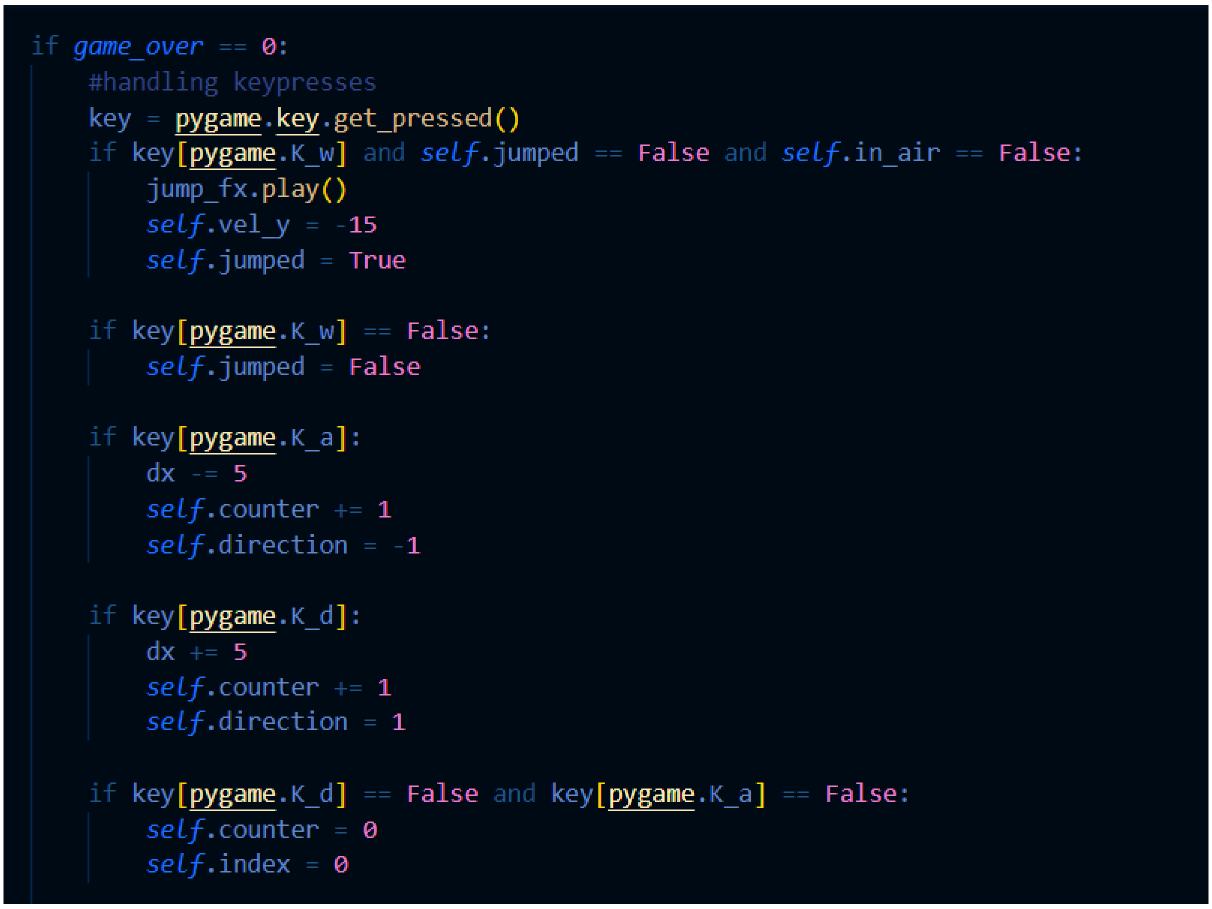
The cloud class is used to represent a cloud in the game. Self.image is the cloud image, rect represents the rectangle that encloses the cloud image, with x and y coordinates. X and y is the initial coordinate of the cloud and the value is used to determine the speed of the cloud in the x direction and screen.blit() method is used to display the cloud in the screen



```python
class Player():

    def __init__(self, x, y):
        #initialize player coordinate (x,y) and call reset function which contains all the inital values
        self.reset(x, y)

    def update(self, game_over):
        #Player movement and collision logic

        #initialize variables
        dx = 0
        dy = 0
        walk_cooldown = 5
        cd = 20
```

The player class is used to control the mechanics of the player in the game. X and Y are used to represent the coordinate of the player and self.reset(x,y) is used to call the reset function which will be covered later.

The update function contains a game_over value which represents the status of game_over and initializes variables. Dx for movement in x direction, dy for movement in y direction, walk_cooldown for slowing down the walking animation and cd for preventing error in the collision



```python
if game_over == 0:
    #handling keypresses
    key = pygame.key.get_pressed()
    if key[pygame.K_w] and self.jumped == False and self.in_air == False:
        jump_fx.play()
        self.vel_y = -15
        self.jumped = True

    if key[pygame.K_w] == False:
        self.jumped = False

    if key[pygame.K_a]:
        dx -= 5
        self.counter += 1
        self.direction = -1

    if key[pygame.K_d]:
        dx += 5
        self.counter += 1
        self.direction = 1

    if key[pygame.K_d] == False and key[pygame.K_a] == False:
        self.counter = 0
        self.index = 0
```

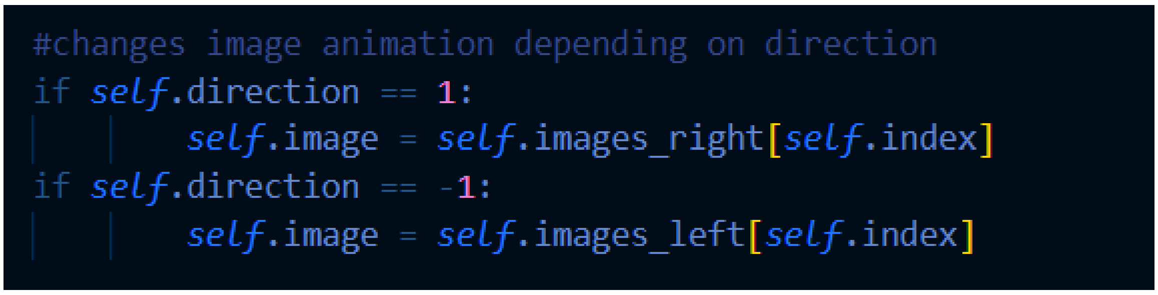
If game_over == 0 (if the player is still alive), key pressed will be received and if the "W" key is pressed (key[pygame.K_w]) and if the player hasn't jumped (self.jumped == False) and is not in the air (self.in_air == False).

a jump sound (jump_fx.play()) will be played and sets the self.vel_y = -15 and self.jumped = True indicating that the player has jumped (key[pygame.K_w] == False) indicates that "W" key is not pressed and resets the self.jumped value to false

if the "A" key is pressed (key[pygame.K_a]), this decreases the x-coordinate change (dx -= 5), increments a counter (self.counter), and sets the direction (self.direction) to -1.

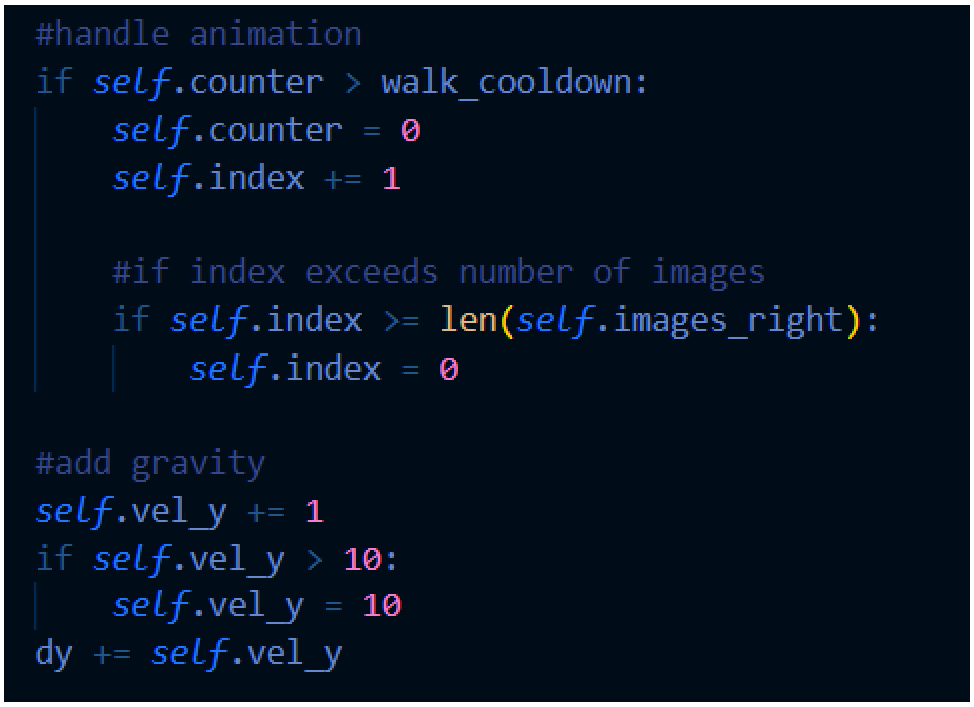
if the "D" key is pressed (key[pygame.K_d]), this increases the x-coordinate change (dx += 5), increments a counter (self.counter), and sets the direction (self.direction) to 1.

(if key[pygame.K_d] == False and key[pygame.K_a] == False) is used to reset the animation by resetting the self.counter and self.index value when both "A" and "D" keys are not pressed



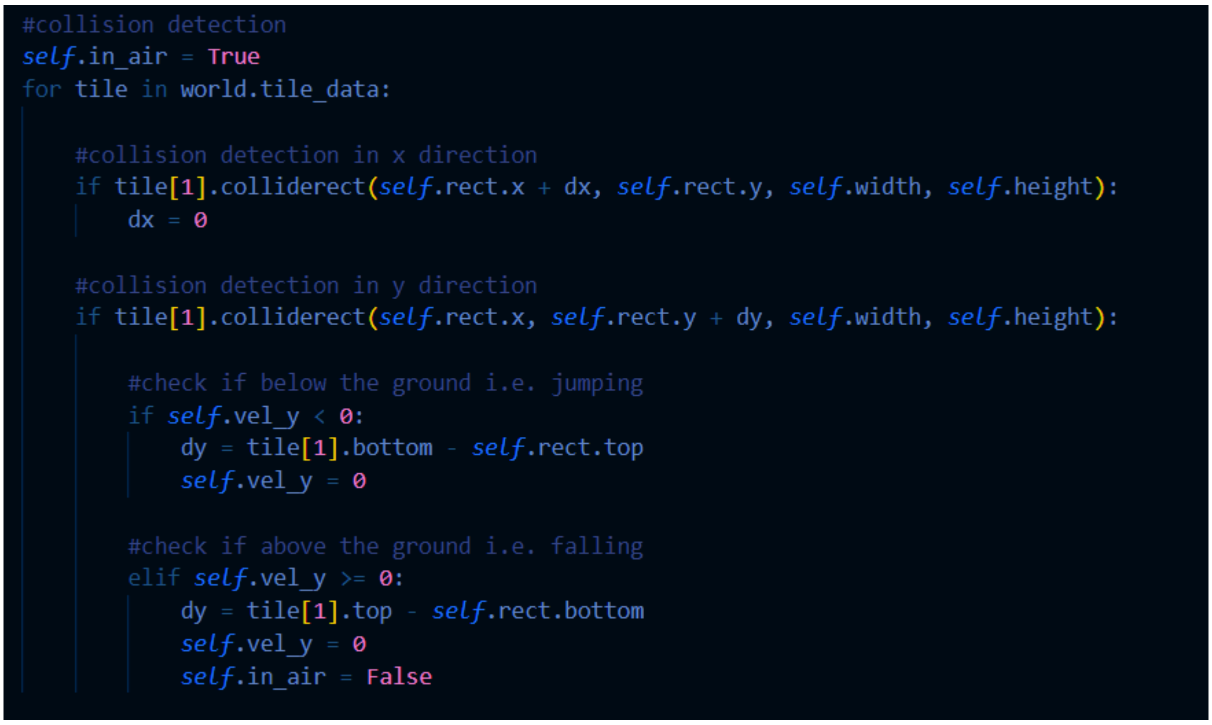
```python
#changes image animation depending on direction
if self.direction == 1:
        self.image = self.images_right[self.index]
if self.direction == -1:
        self.image = self.images_left[self.index]
```

This is used for displaying animation by changing the image depending on the direction the user is going



```
#handle animation
if self.counter > walk_cooldown:
    self.counter = 0
    self.index += 1

    #if index exceeds number of images
    if self.index >= len(self.images_right):
        self.index = 0

#add gravity
self.vel_y += 1
if self.vel_y > 10:
    self.vel_y = 10
dy += self.vel_y
```

This is used to handle animation where walk_cooldown is setting a time limit and self.counter must pass the limit for the index to change, this creates an animation that can be seen and if the index is greater than the number of images in the list, the index is reset to 0, looping the animation.
Self.vel_y causes a constant downward acceleration and it is capped at 10 so that the player doesn't fall too fast. dy += self.vel_y adjusts the vertical position (dy) of the player by adding the current vertical velocity

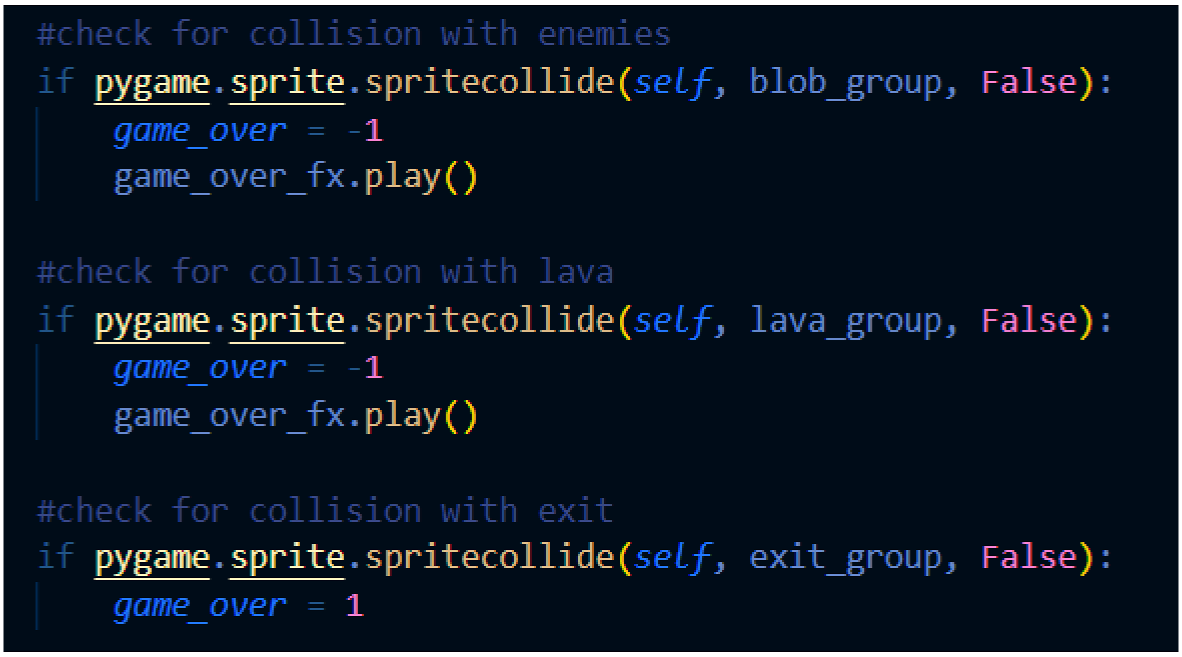


```
#collision detection
self.in_air = True
for tile in world.tile_data:

    #collision detection in x direction
    if tile[1].colliderect(self.rect.x + dx, self.rect.y, self.width, self.height):
        dx = 0

    #collision detection in y direction
    if tile[1].colliderect(self.rect.x, self.rect.y + dy, self.width, self.height):

        #check if below the ground i.e. jumping
        if self.vel_y < 0:
            dy = tile[1].bottom - self.rect.top
            self.vel_y = 0

        #check if above the ground i.e. falling
        elif self.vel_y >= 0:
            dy = tile[1].top - self.rect.bottom
            self.vel_y = 0
            self.in_air = False
```

(self.in_air = True) is used to keep track of the player when the player is currently airborne

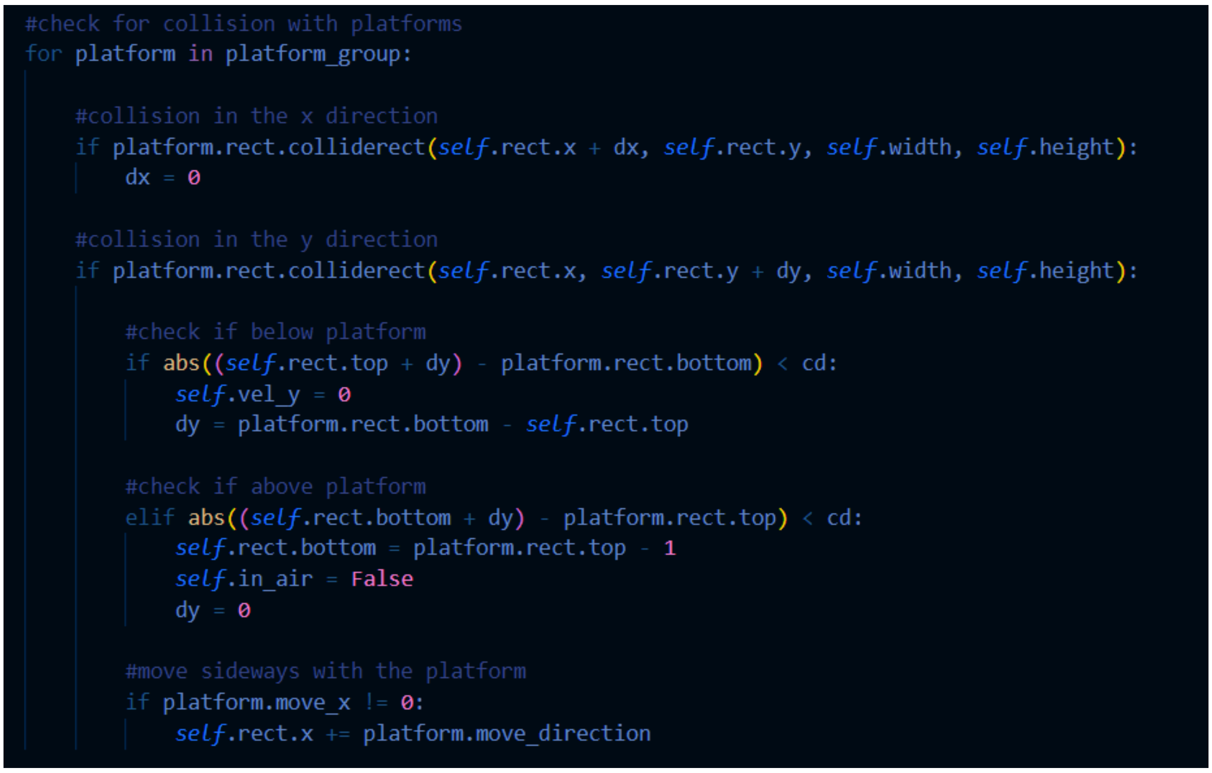
(if tile[1].colliderect(self.rect.x + dx, self.rect.y, self.width, self.height)) checks for collisions with tiles in the x direction after applying the change in x (dx). If a collision is detected, it sets dx to 0, effectively preventing movement in the x-direction.

(if tile[1].colliderect(self.rect.x, self.rect.y + dy, self.width, self.height)) checks for collisions with tiles in the y direction after applying the change in y (dy).
If the player is moving upward (self.vel_y < 0), it adjusts dy to position the player just above the colliding tile (dy = tile[1].bottom - self.rect.top) and sets vertical velocity to 0 (self.vel_y = 0).
If the player is moving downward or stationary (self.vel_y >= 0), it adjusts dy to position the player just below the colliding tile (dy = tile[1].top - self.rect.bottom), sets vertical velocity to 0, and marks the player as not in the air (self.in_air = False).



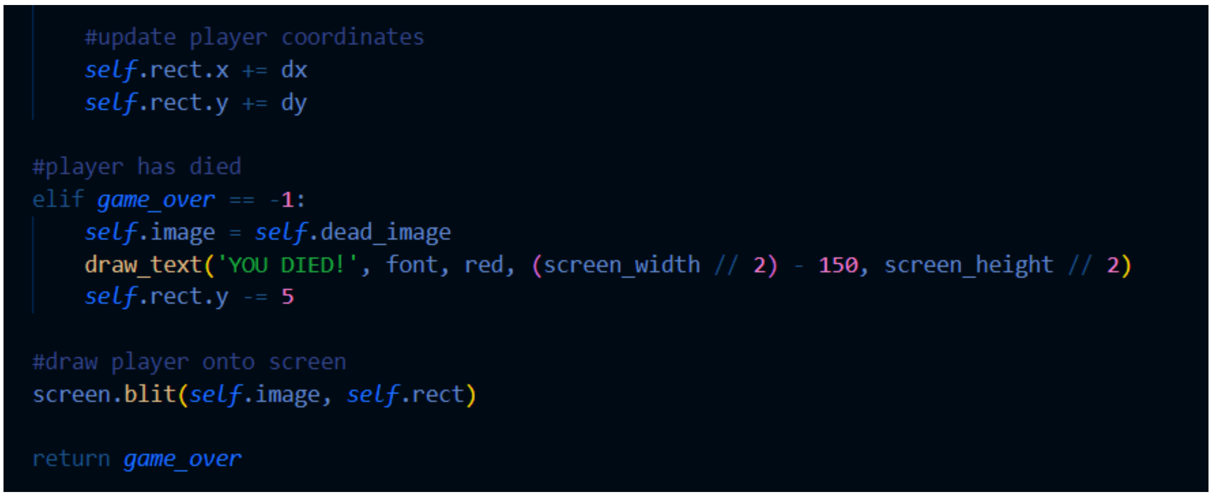
```python
#check for collision with enemies
if pygame.sprite.spritecollide(self, blob_group, False):
    game_over = -1
    game_over_fx.play()

#check for collision with lava
if pygame.sprite.spritecollide(self, lava_group, False):
    game_over = -1
    game_over_fx.play()

#check for collision with exit
if pygame.sprite.spritecollide(self, exit_group, False):
    game_over = 1
```

If the player sprite (self) collides with any sprite in the blob_group or lava_group, game_over = -1 indicating that the game is over and a game over sound is played (game_over_fx.play()), but if it collides with the exit_group then game_over = 1 meaning that the player has advanced to the next level. The false parameter is used to indicate that sprites must not be removed even after colliding.



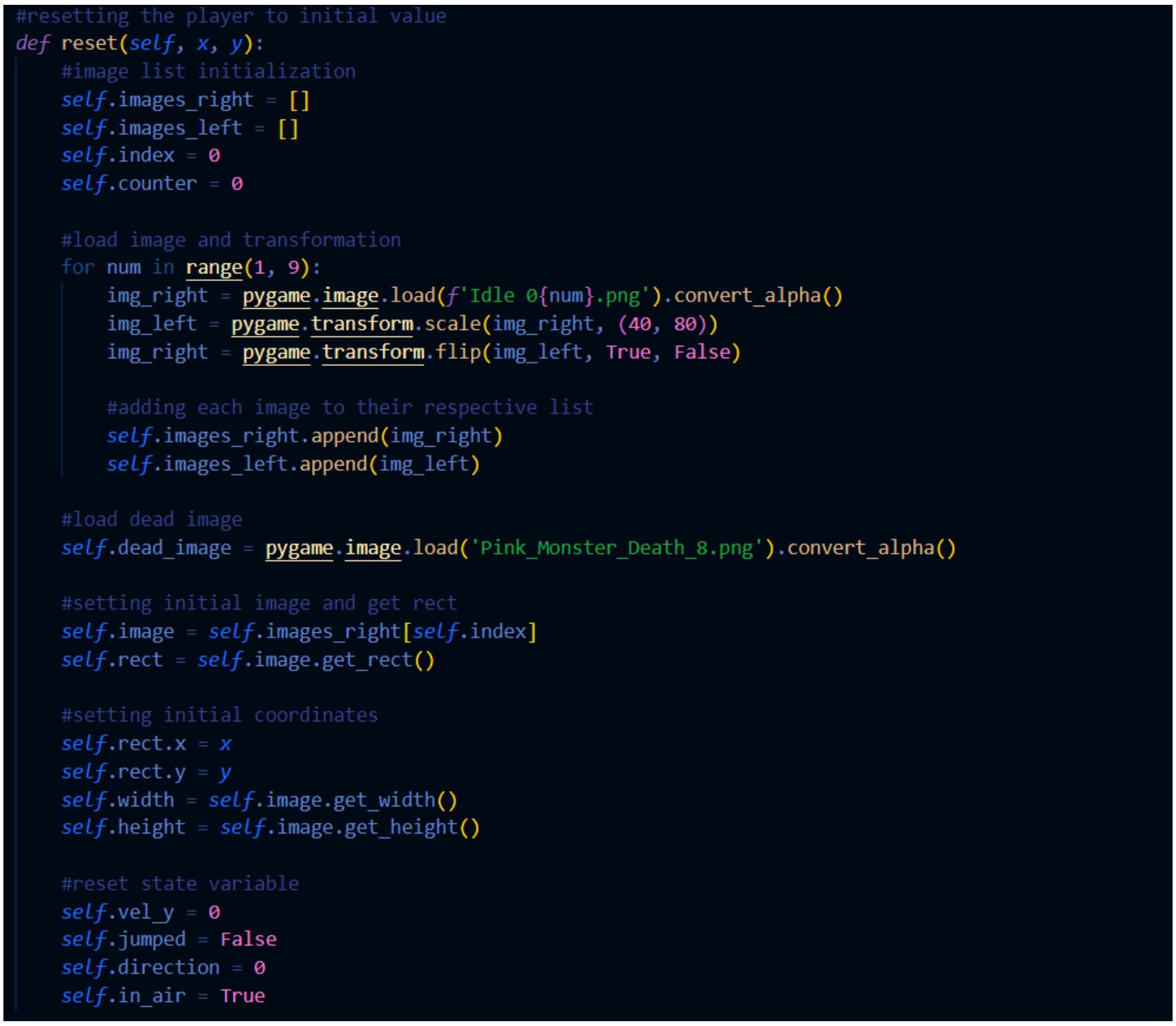
```
#check for collision with platforms
for platform in platform_group:

    #collision in the x direction
    if platform.rect.colliderect(self.rect.x + dx, self.rect.y, self.width, self.height):
        dx = 0

    #collision in the y direction
    if platform.rect.colliderect(self.rect.x, self.rect.y + dy, self.width, self.height):

        #check if below platform
        if abs((self.rect.top + dy) - platform.rect.bottom) < cd:
            self.vel_y = 0
            dy = platform.rect.bottom - self.rect.top

        #check if above platform
        elif abs((self.rect.bottom + dy) - platform.rect.top) < cd:
            self.rect.bottom = platform.rect.top - 1
            self.in_air = False
            dy = 0

        #move sideways with the platform
        if platform.move_x != 0:
            self.rect.x += platform.move_direction
```

This is used to check for collisions between the player and each platform in the platform_group in the X direction. If a collision is detected, it sets the horizontal movement (dx) to 0, preventing the player from moving through the platform and If the player is below a platform, it adjusts dy to land on the platform. If the player is above a platform, it sets the player's position just below the platform and marks the player as not in the air and resets the dy value preventing the player to move unnecessarily.
If a platform is moving horizontally (move_x is not 0), the player's X position is adjusted based on the direction of the platform's movement. This makes the player move with the platform



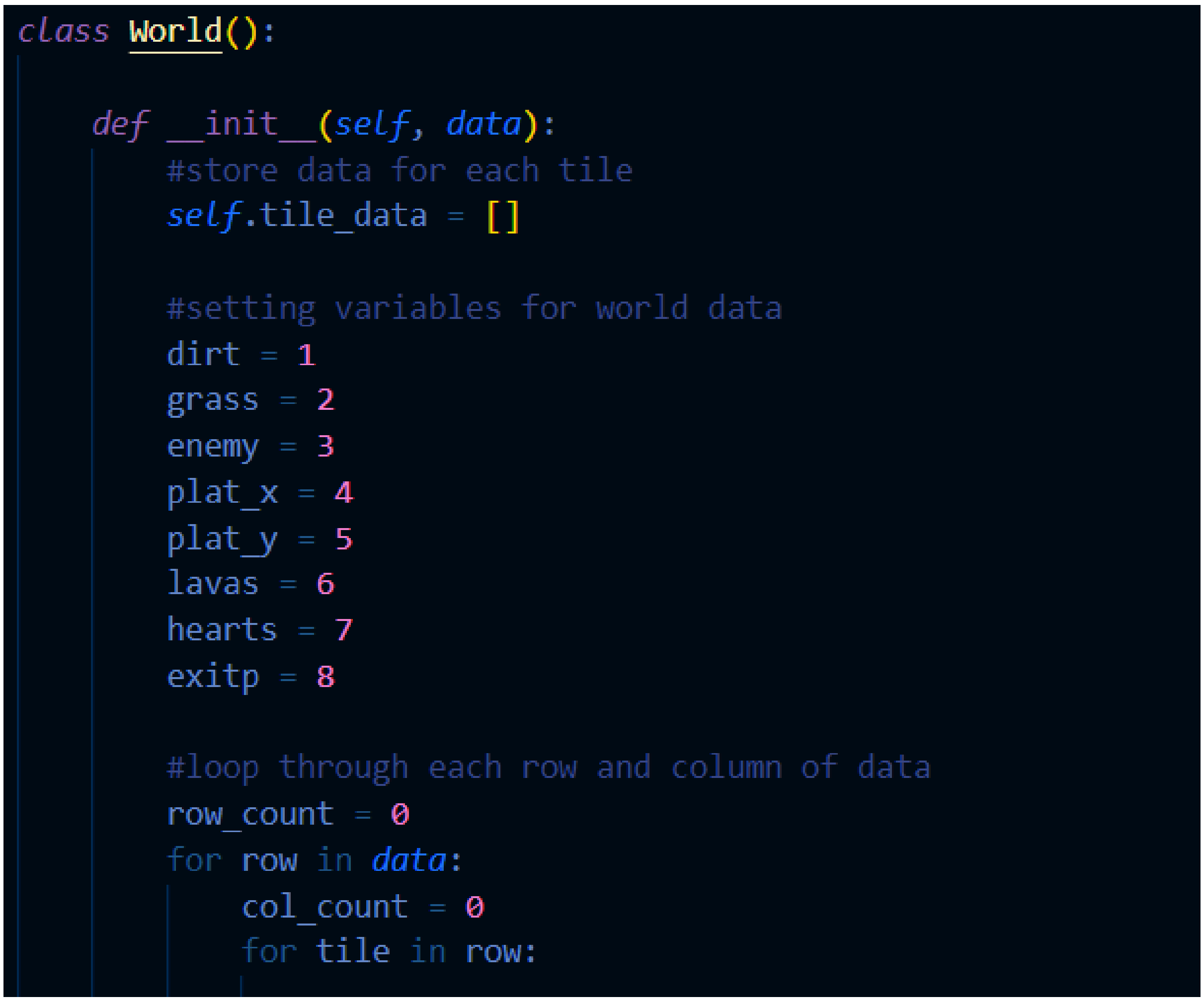
```
        #update player coordinates
        self.rect.x += dx
        self.rect.y += dy

    #player has died
    elif game_over == -1:
        self.image = self.dead_image
        draw_text('YOU DIED!', font, red, (screen_width // 2) - 150, screen_height // 2)
        self.rect.y -= 5

    #draw player onto screen
    screen.blit(self.image, self.rect)

    return game_over
```

This updates the player x and y location and if game_over == -1 (player has died), player image turns into a dead image and using the draw_text function, a text “YOU DIED!” appeared on the screen with self.rect.y -=5 causing the dead image to go upwards. This also returns the game_over value which keeps updating the game_over status



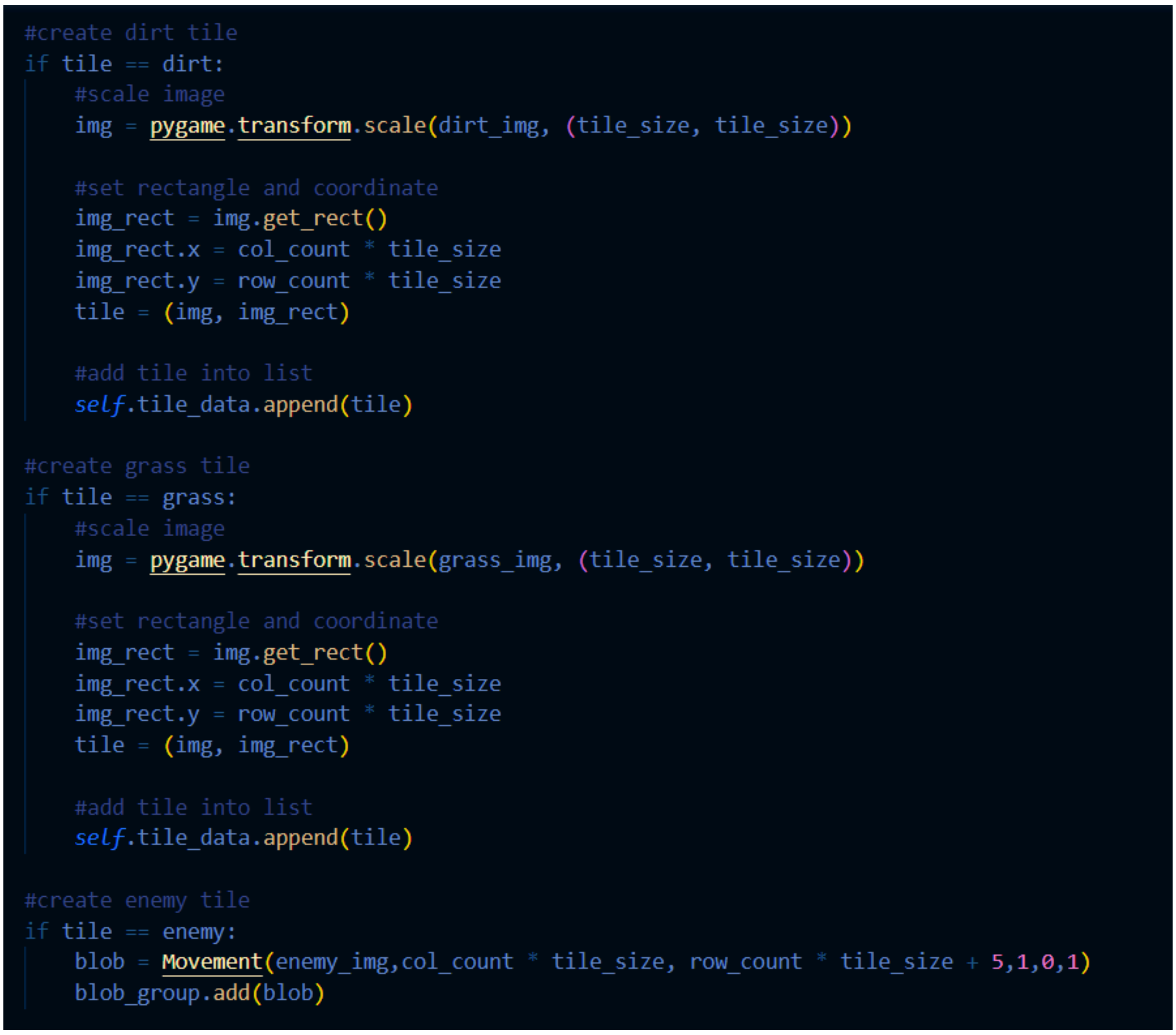
```
#resetting the player to initial value
def reset(self, x, y):
    #image list initialization
    self.images_right = []
    self.images_left = []
    self.index = 0
    self.counter = 0

    #load image and transformation
    for num in range(1, 9):
        img_right = pygame.image.load(f'Idle 0{num}.png').convert_alpha()
        img_left = pygame.transform.scale(img_right, (40, 80))
        img_right = pygame.transform.flip(img_left, True, False)

        #adding each image to their respective list
        self.images_right.append(img_right)
        self.images_left.append(img_left)

    #load dead image
    self.dead_image = pygame.image.load('Pink_Monster_Death_8.png').convert_alpha()

    #setting initial image and get rect
    self.image = self.images_right[self.index]
    self.rect = self.image.get_rect()

    #setting initial coordinates
    self.rect.x = x
    self.rect.y = y
    self.width = self.image.get_width()
    self.height = self.image.get_height()

    #reset state variable
    self.vel_y = 0
    self.jumped = False
    self.direction = 0
    self.in_air = True
```

Two lists (images_right and images_left) are initialized to store images for the player facing right and left, respectively and index and counter are initialized to keep track of the current image index and a counter.
Then a loop is made to load images for the player facing right and left, transforming the left-facing images and adding both versions to their respective lists and an image for the dead image is loaded.
The initial image is set to the first image in the right-facing list (images_right) and the rect attribute is initialized based on the current image. The initial coordinates of the player are set, and the width and height attributes are updated based on the current image and this resets the state variable whenever the reset function is called

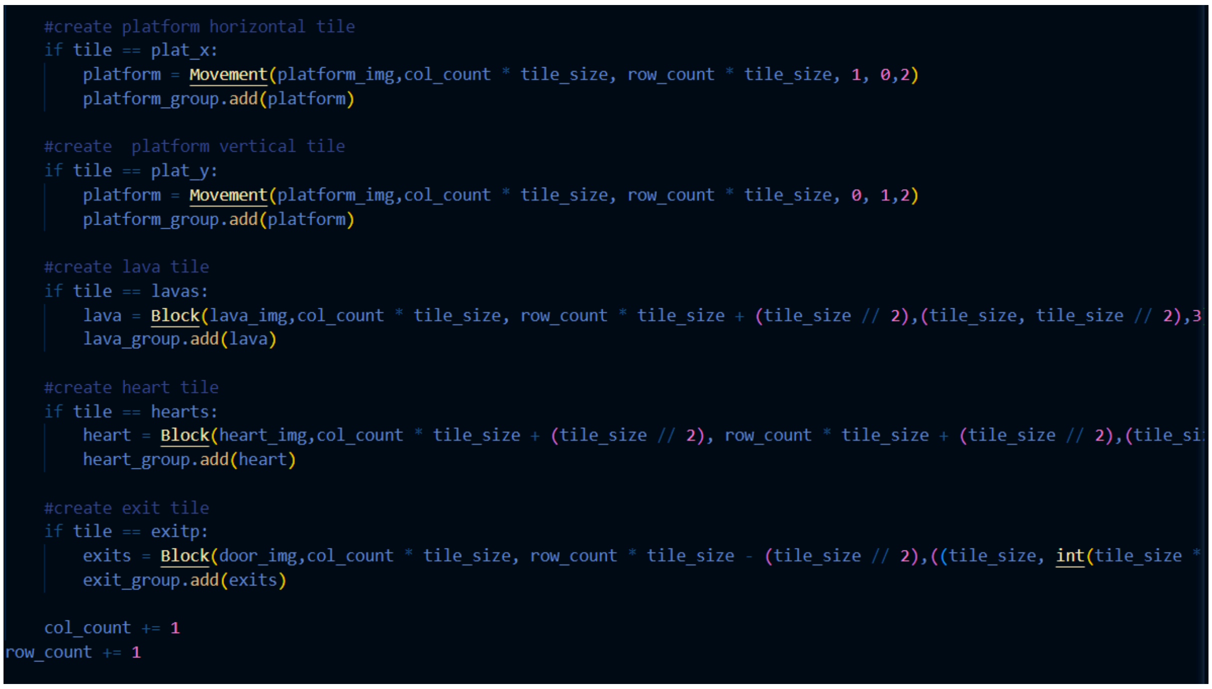


```
class World():

    def __init__(self, data):
        #store data for each tile
        self.tile_data = []

        #setting variables for world data
        dirt = 1
        grass = 2
        enemy = 3
        plat_x = 4
        plat_y = 5
        lavas = 6
        hearts = 7
        exitp = 8

        #loop through each row and column of data
        row_count = 0
        for row in data:
            col_count = 0
            for tile in row:
```

This creates a world class which receives a data parameter and creates a list (self.tile_data) and setting each variable to each number for the world data as shown before.
This loops for every list in data and every value inside the list

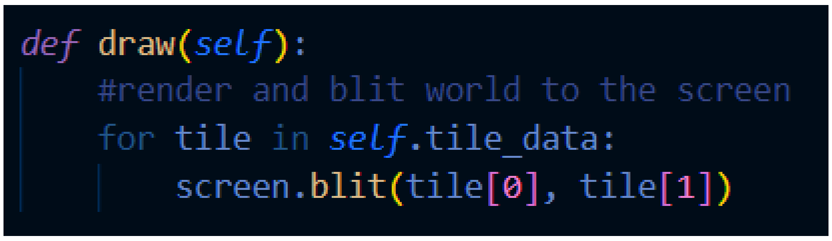


```
#create dirt tile
if tile == dirt:
    #scale image
    img = pygame.transform.scale(dirt_img, (tile_size, tile_size))

    #set rectangle and coordinate
    img_rect = img.get_rect()
    img_rect.x = col_count * tile_size
    img_rect.y = row_count * tile_size
    tile = (img, img_rect)

    #add tile into list
    self.tile_data.append(tile)

#create grass tile
if tile == grass:
    #scale image
    img = pygame.transform.scale(grass_img, (tile_size, tile_size))

    #set rectangle and coordinate
    img_rect = img.get_rect()
    img_rect.x = col_count * tile_size
    img_rect.y = row_count * tile_size
    tile = (img, img_rect)

    #add tile into list
    self.tile_data.append(tile)

#create enemy tile
if tile == enemy:
    blob = Movement(enemy_img,col_count * tile_size, row_count * tile_size + 5,1,0,1)
    blob_group.add(blob)
```



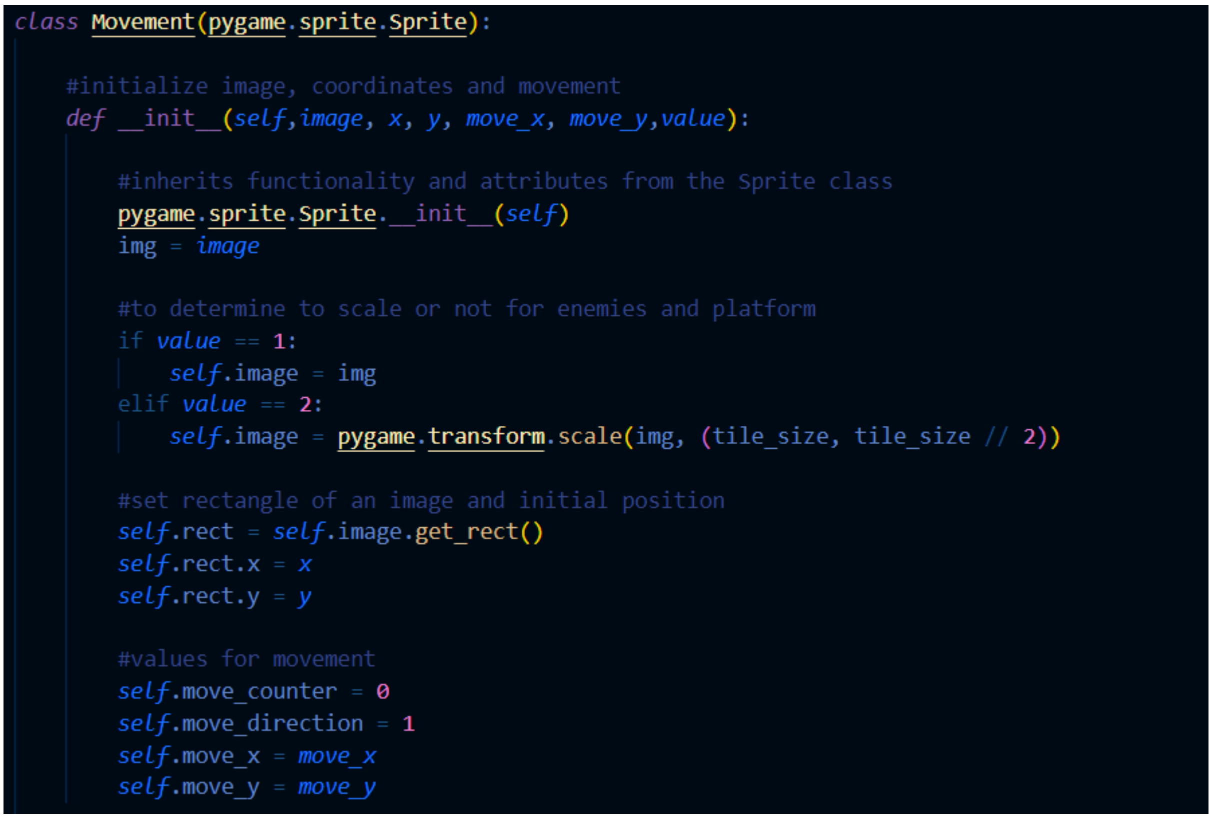
```
    #create platform horizontal tile
    if tile == plat_x:
        platform = Movement(platform_img,col_count * tile_size, row_count * tile_size, 1, 0,2)
        platform_group.add(platform)

    #create  platform vertical tile
    if tile == plat_y:
        platform = Movement(platform_img,col_count * tile_size, row_count * tile_size, 0, 1,2)
        platform_group.add(platform)

    #create lava tile
    if tile == lavas:
        lava = Block(lava_img,col_count * tile_size, row_count * tile_size + (tile_size // 2),(tile_size, tile_size // 2),3
        lava_group.add(lava)

    #create heart tile
    if tile == hearts:
        heart = Block(heart_img,col_count * tile_size + (tile_size // 2), row_count * tile_size + (tile_size // 2),(tile_si
        heart_group.add(heart)

    #create exit tile
    if tile == exitp:
        exits = Block(door_img,col_count * tile_size, row_count * tile_size - (tile_size // 2),((tile_size, int(tile_size *
        exit_group.add(exits)

    col_count += 1
row_count += 1
```

Dirts and grass are added to the tile_data list and every number inside the list is changed with blocks with each of them excluding dirt and grass adding to their own perspective group, using the Movement and Block class, the tiles are set to have their own image, positioned and scaled.



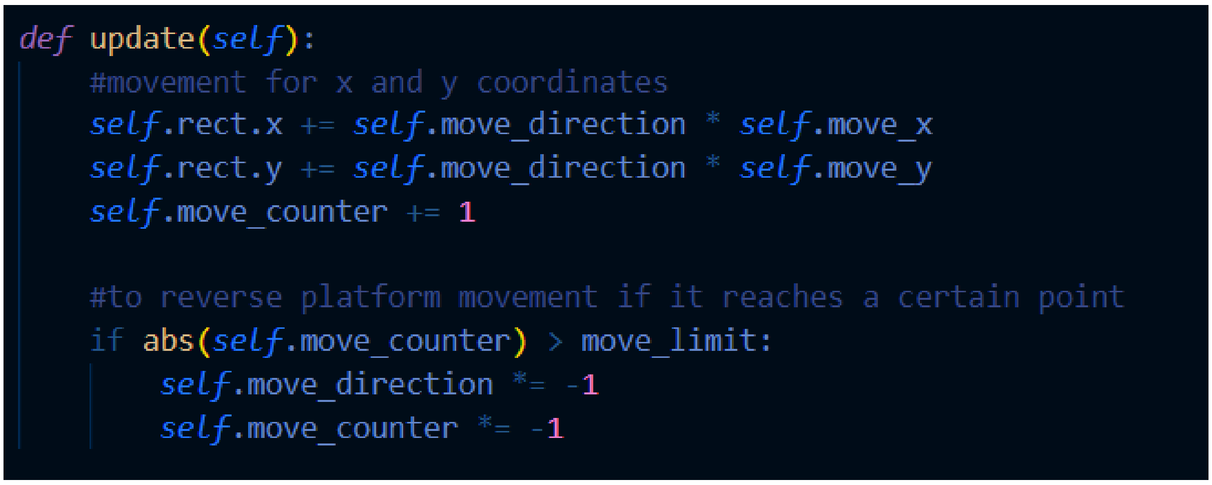
```python
def draw(self):
    #render and blit world to the screen
    for tile in self.tile_data:
        screen.blit(tile[0], tile[1])
```

This displays the dirt and grass inside the tile_data list on to the screen



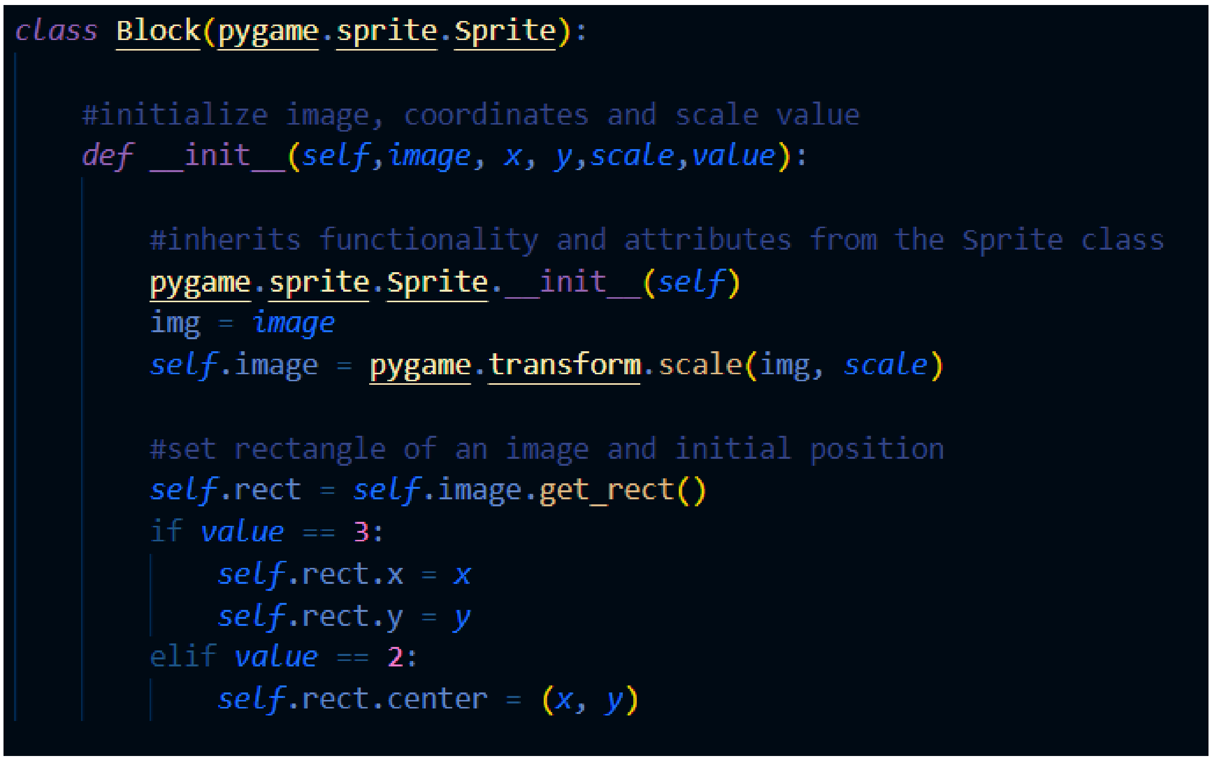
```python
class Movement(pygame.sprite.Sprite):

    #initialize image, coordinates and movement
    def __init__(self,image, x, y, move_x, move_y,value):

        #inherits functionality and attributes from the Sprite class
        pygame.sprite.Sprite.__init__(self)
        img = image

        #to determine to scale or not for enemies and platform
        if value == 1:
            self.image = img
        elif value == 2:
            self.image = pygame.transform.scale(img, (tile_size, tile_size // 2))

        #set rectangle of an image and initial position
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y

        #values for movement
        self.move_counter = 0
        self.move_direction = 1
        self.move_x = move_x
        self.move_y = move_y
```

The movement class is used for enemies and platforms, it receives 6 parameters. Self.image for each image, x and y for coordinates, move_x and move_y for adding movement in x and y directions and value for deciding whether to scale the image or not. Self.rect is used to get the image rectangle and self.counter and self.move_direction is used to keep track of the movement



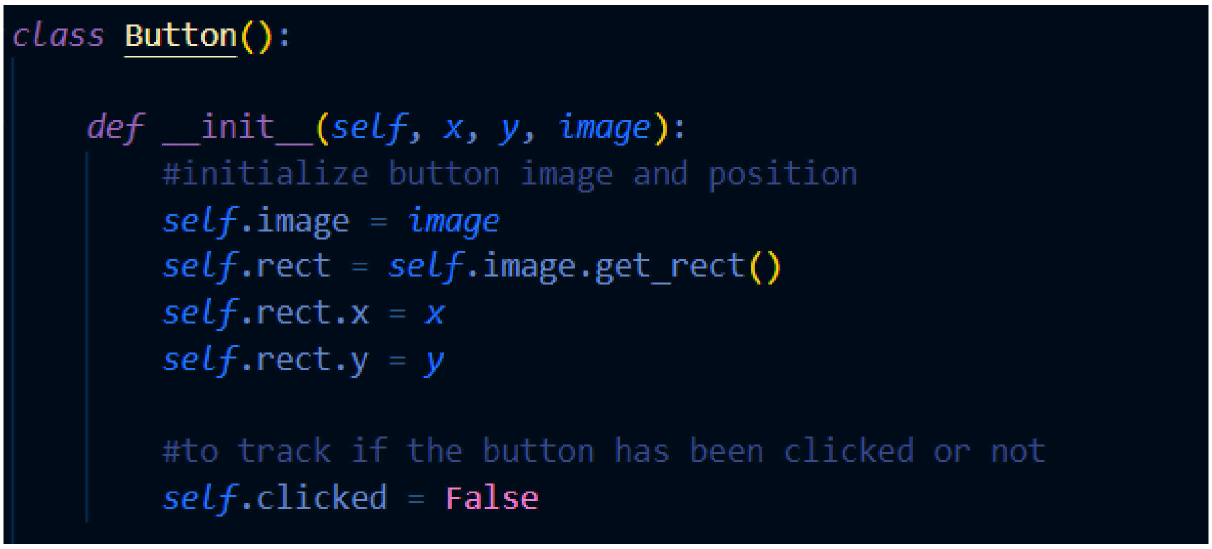
```python
def update(self):
    #movement for x and y coordinates
    self.rect.x += self.move_direction * self.move_x
    self.rect.y += self.move_direction * self.move_y
    self.move_counter += 1

    #to reverse platform movement if it reaches a certain point
    if abs(self.move_counter) > move_limit:
        self.move_direction *= -1
        self.move_counter *= -1
```

The position of x and y coordinates of the rectangle is added with the move_direction*self.move_x and self.move_y.
This makes it easier since for the horizontal platform, it just needs to fill in the move_x value and set the move_y value to 0 which will not move in the vertical direction and the same can be said for the vertical platform. If the move_counter is > move_limit, the direction and move_counter is multiplied by -1, this makes the rect.x and rect.y moves towards the opposite direction from before and the counter creating equal time for both positive and negative, this creates the enemies and platforms moves right and left within a certain area.



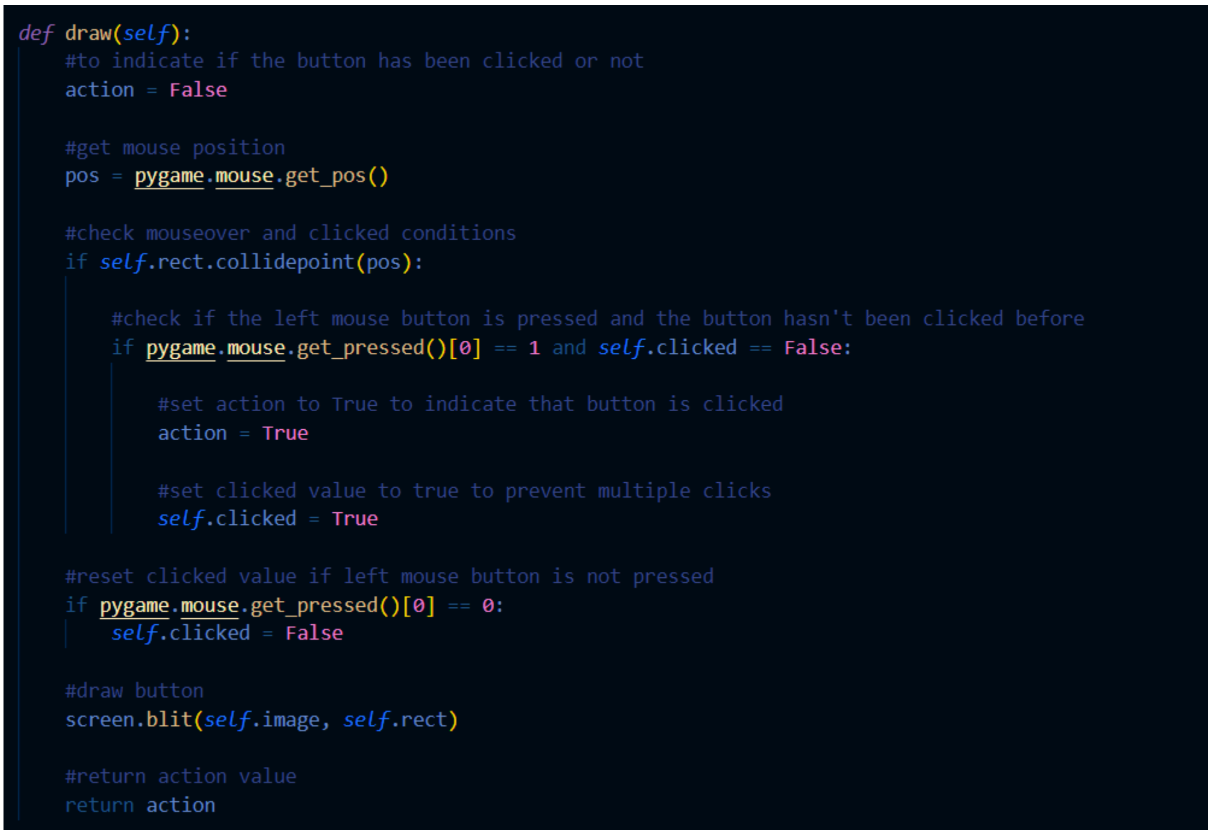
```python
class Block(pygame.sprite.Sprite):

    #initialize image, coordinates and scale value
    def __init__(self,image, x, y,scale,value):

        #inherits functionality and attributes from the Sprite class
        pygame.sprite.Sprite.__init__(self)
        img = image
        self.image = pygame.transform.scale(img, scale)

        #set rectangle of an image and initial position
        self.rect = self.image.get_rect()
        if value == 3:
            self.rect.x = x
            self.rect.y = y
        elif value == 2:
            self.rect.center = (x, y)
```

The block class is used for stationery objects such as lava, heart and exit, it receives 5 parameters. Self.image for each image, x and y for coordinates, scale for scaling the image and value for deciding whether the image rect will be in center or not



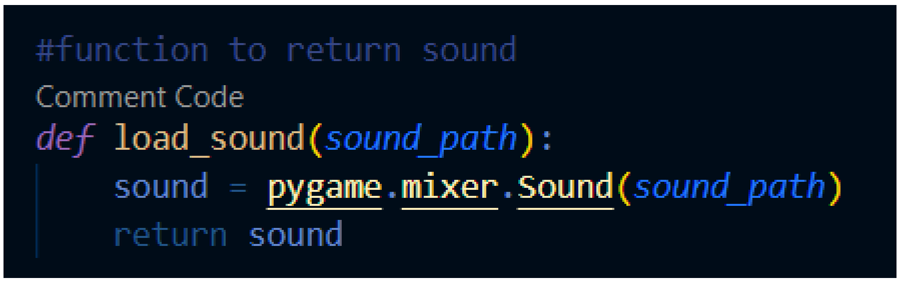
```python
class Button():

    def __init__(self, x, y, image):
        #initialize button image and position
        self.image = image
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y

        #to track if the button has been clicked or not
        self.clicked = False
```

The button class is used for buttons such as the start button, quit button and restart button, it receives 3 parameters. X and y for coordinates and image for the button image with an initial value of false for self.clicked for tracking the button if it has been clicked or not.



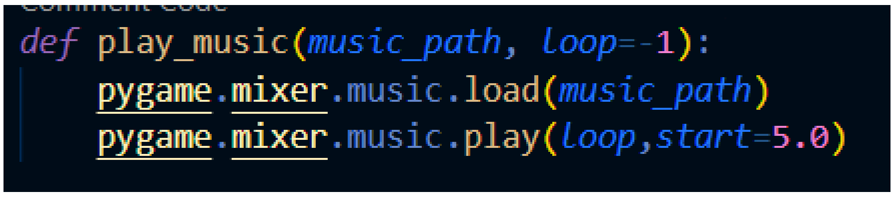
```python
def draw(self):
    #to indicate if the button has been clicked or not
    action = False

    #get mouse position
    pos = pygame.mouse.get_pos()

    #check mouseover and clicked conditions
    if self.rect.collidepoint(pos):

        #check if the left mouse button is pressed and the button hasn't been clicked before
        if pygame.mouse.get_pressed()[0] == 1 and self.clicked == False:

            #set action to True to indicate that button is clicked
            action = True

            #set clicked value to true to prevent multiple clicks
            self.clicked = True

    #reset clicked value if left mouse button is not pressed
    if pygame.mouse.get_pressed()[0] == 0:
        self.clicked = False

    #draw button
    screen.blit(self.image, self.rect)

    #return action value
    return action
```

Action = False is used to indicate whether the button has been clicked and the current mouse position is retrieved by using pygame.mouse.get_pos() and stored in the pos variable.
The collidepoint method is used to check if the mouse is over the button (the button's rectangle, represented by self.rect). It checks if the left mouse button (pygame.mouse.get_pressed()[0]) is pressed and if the button hasn't been clicked before (self.clicked == False).
If both conditions are met, action is set to True to indicate that the button has been clicked, and self.clicked is set to True to prevent multiple clicks. If the left mouse button is not pressed, the self.clicked variable is reset to False which allows the button to be clicked again. The button is then displayed on the screen using the blit function and action value is returned.



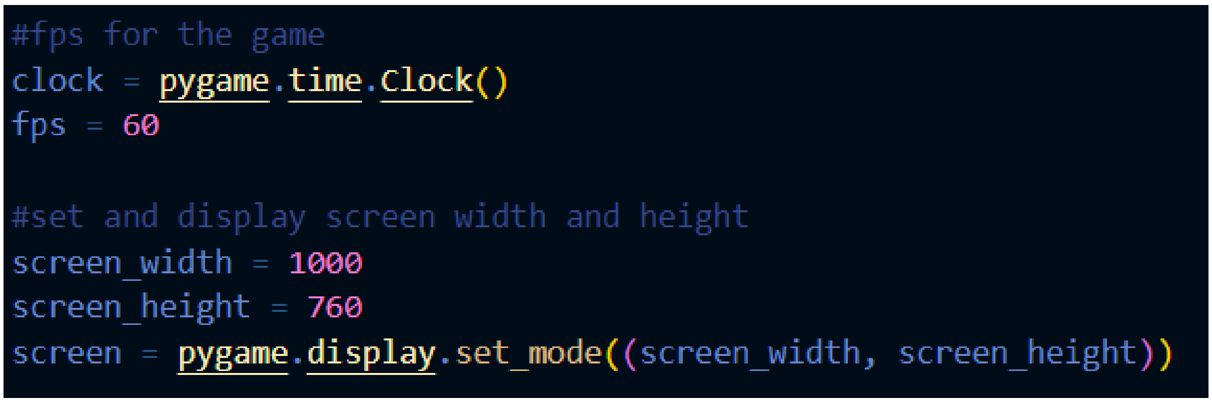
```
#function to return sound
Comment Code
def load_sound(sound_path):
    sound = pygame.mixer.Sound(sound_path)
    return sound
```

The function receives a sound parameter and creates a sound by loading the file and stores it in sound variable and returns the sound variable



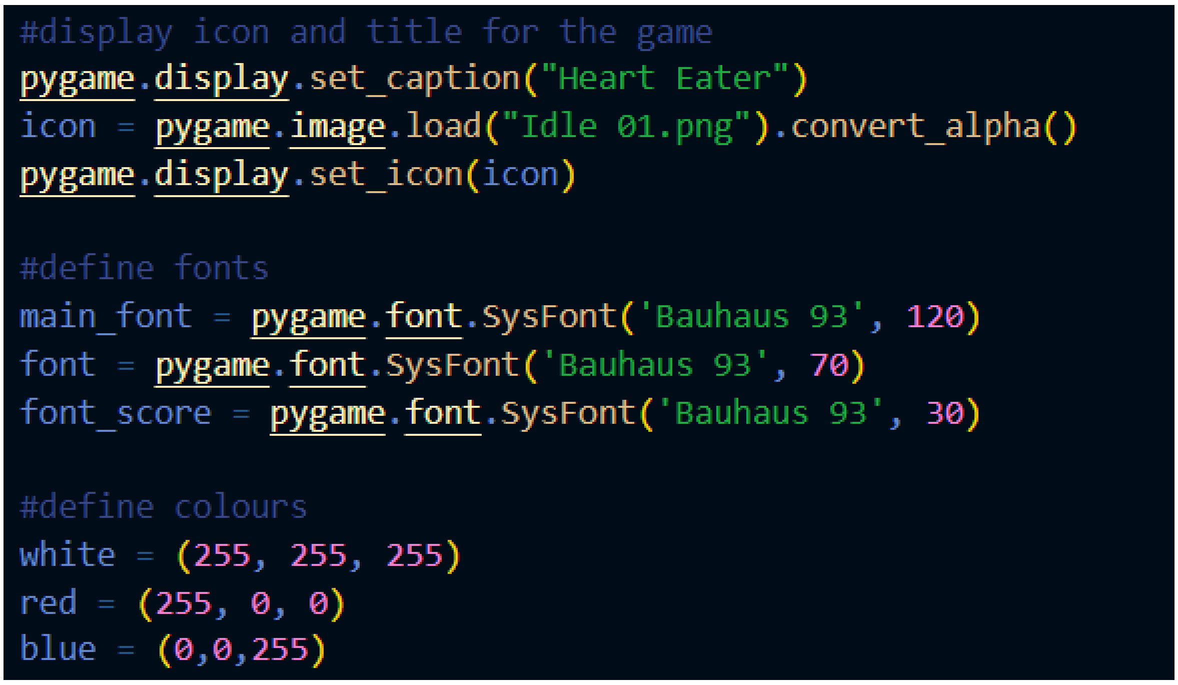
```
def play_music(music_path, loop=-1):
    pygame.mixer.music.load(music_path)
    pygame.mixer.music.play(loop,start=5.0)
```

This function is used to loop a music in the background infinitely

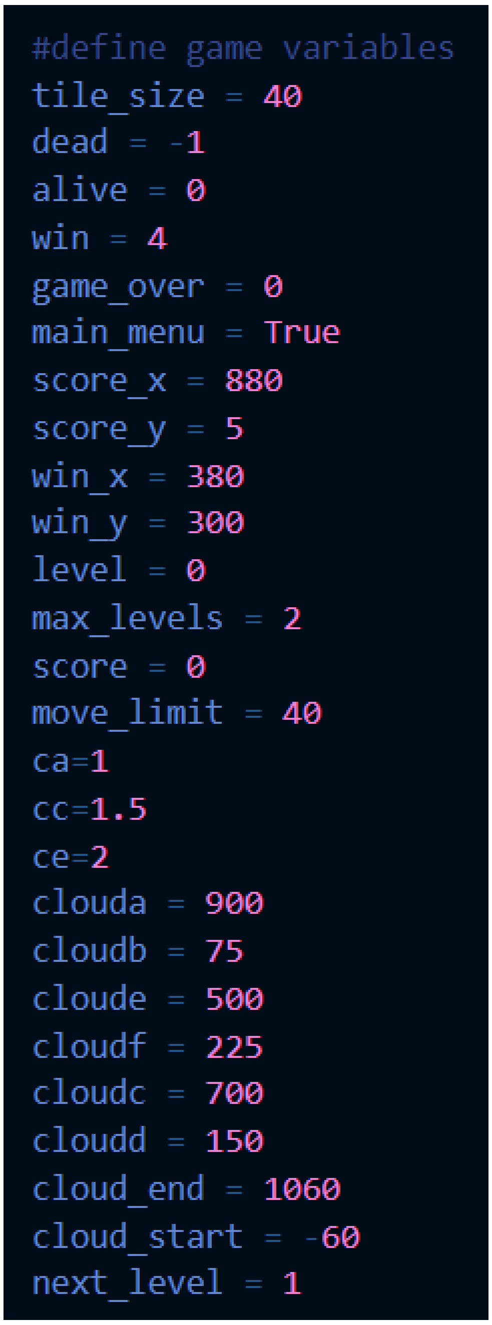


```
#fps for the game
clock = pygame.time.Clock()
fps = 60

#set and display screen width and height
screen_width = 1000
screen_height = 760
screen = pygame.display.set_mode((screen_width, screen_height))
```

This sets the fps, screen width, screen height and displays the screen for the game

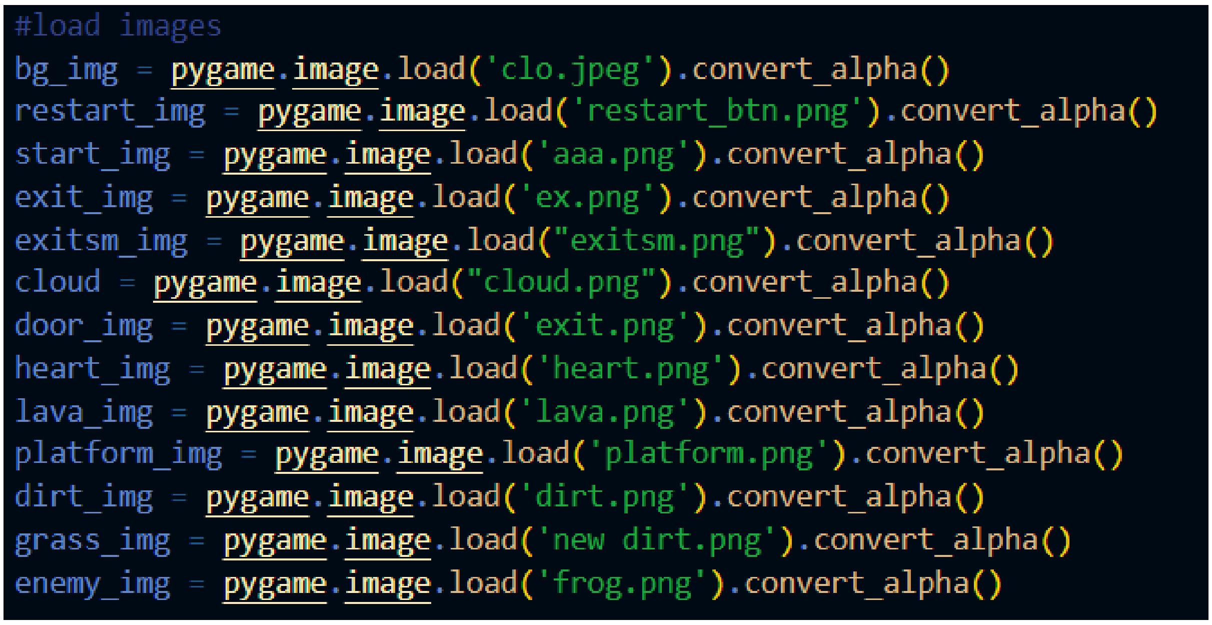

```python
#display icon and title for the game
pygame.display.set_caption("Heart Eater")
icon = pygame.image.load("Idle 01.png").convert_alpha()
pygame.display.set_icon(icon)

#define fonts
main_font = pygame.font.SysFont('Bauhaus 93', 120)
font = pygame.font.SysFont('Bauhaus 93', 70)
font_score = pygame.font.SysFont('Bauhaus 93', 30)

#define colours
white = (255, 255, 255)
red = (255, 0, 0)
blue = (0,0,255)
```

This sets the title and icon for the game and it defines some fonts and colours



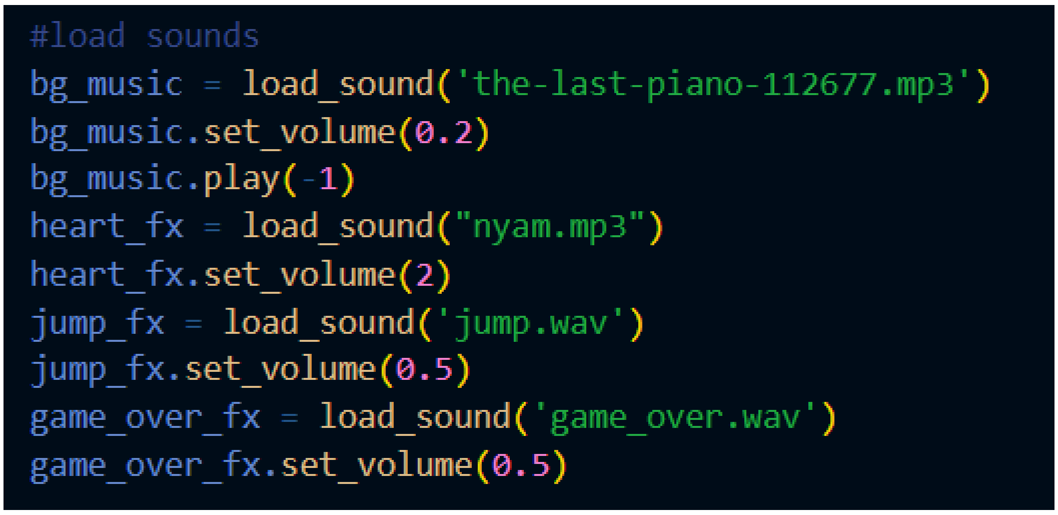
```
#define game variables
tile_size = 40
dead = -1
alive = 0
win = 4
game_over = 0
main_menu = True
score_x = 880
score_y = 5
win_x = 380
win_y = 300
level = 0
max_levels = 2
score = 0
move_limit = 40
ca=1
cc=1.5
ce=2
clouda = 900
cloudb = 75
cloude = 500
cloudf = 225
cloudc = 700
cloudd = 150
cloud_end = 1060
cloud_start = -60
next_level = 1
```

This defines essential game variables which is used for the size of a tile, dead alive and win state, game over state, main menu, score position, text position, initial level, max level, initial score, move limit as shown before, clouds speeds, clouds location and adding the next level value to go to the next level



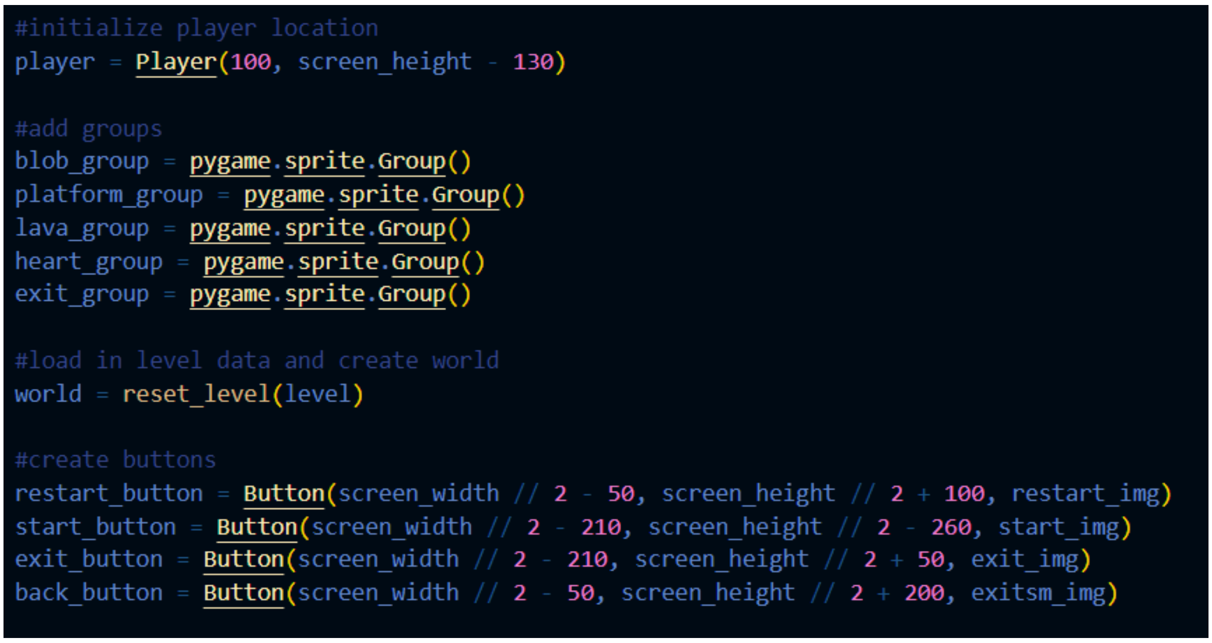
```
#load images
bg_img = pygame.image.load('clo.jpeg').convert_alpha()
restart_img = pygame.image.load('restart_btn.png').convert_alpha()
start_img = pygame.image.load('aaa.png').convert_alpha()
exit_img = pygame.image.load('ex.png').convert_alpha()
exitsm_img = pygame.image.load("exitsm.png").convert_alpha()
cloud = pygame.image.load("cloud.png").convert_alpha()
door_img = pygame.image.load('exit.png').convert_alpha()
heart_img = pygame.image.load('heart.png').convert_alpha()
lava_img = pygame.image.load('lava.png').convert_alpha()
platform_img = pygame.image.load('platform.png').convert_alpha()
dirt_img = pygame.image.load('dirt.png').convert_alpha()
grass_img = pygame.image.load('new dirt.png').convert_alpha()
enemy_img = pygame.image.load('frog.png').convert_alpha()
```

This loads images used for the game and convert_alpha() change the pixel format of an image including per pixel alphas method after loading so that the image has per pixel transparency.



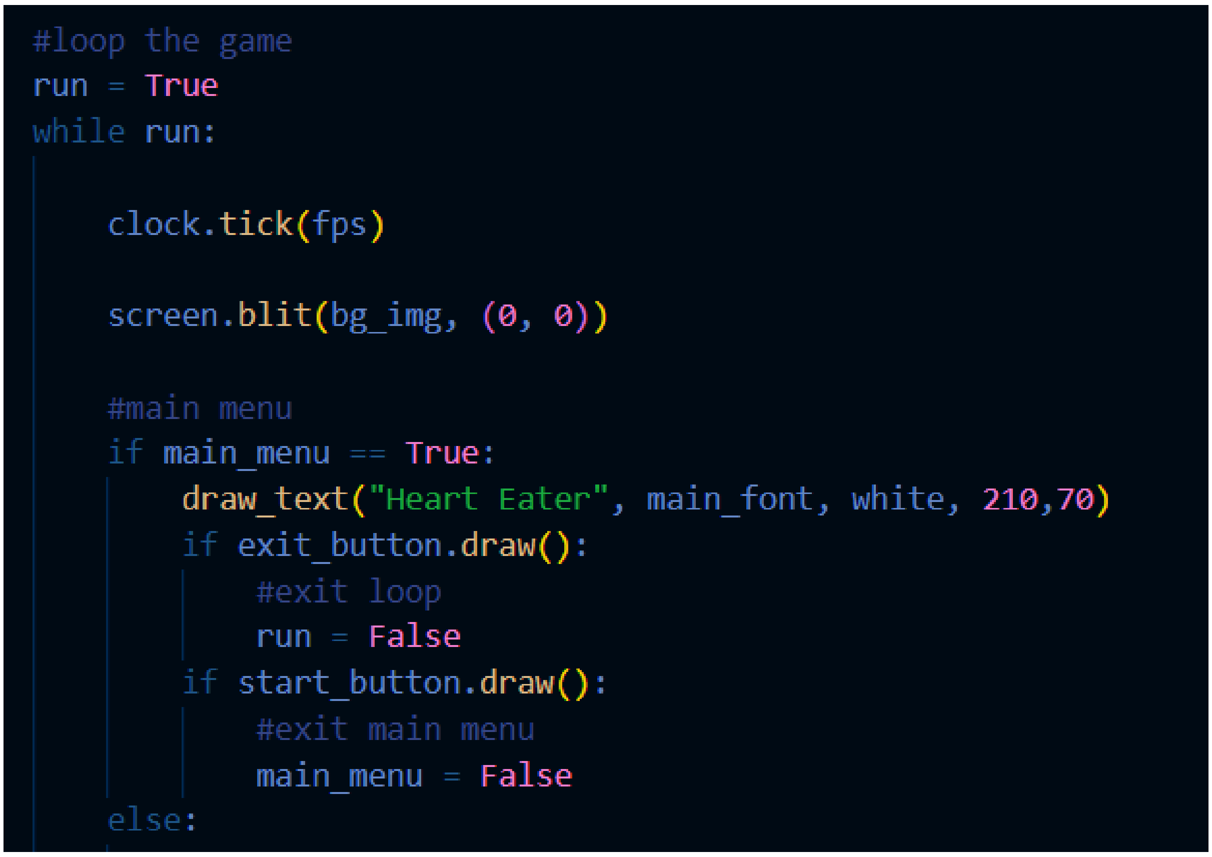
```
#load sounds
bg_music = load_sound('the-last-piano-112677.mp3')
bg_music.set_volume(0.2)
bg_music.play(-1)
heart_fx = load_sound("nyam.mp3")
heart_fx.set_volume(2)
jump_fx = load_sound('jump.wav')
jump_fx.set_volume(0.5)
game_over_fx = load_sound('game_over.wav')
game_over_fx.set_volume(0.5)
```

This loads the sounds used in the game



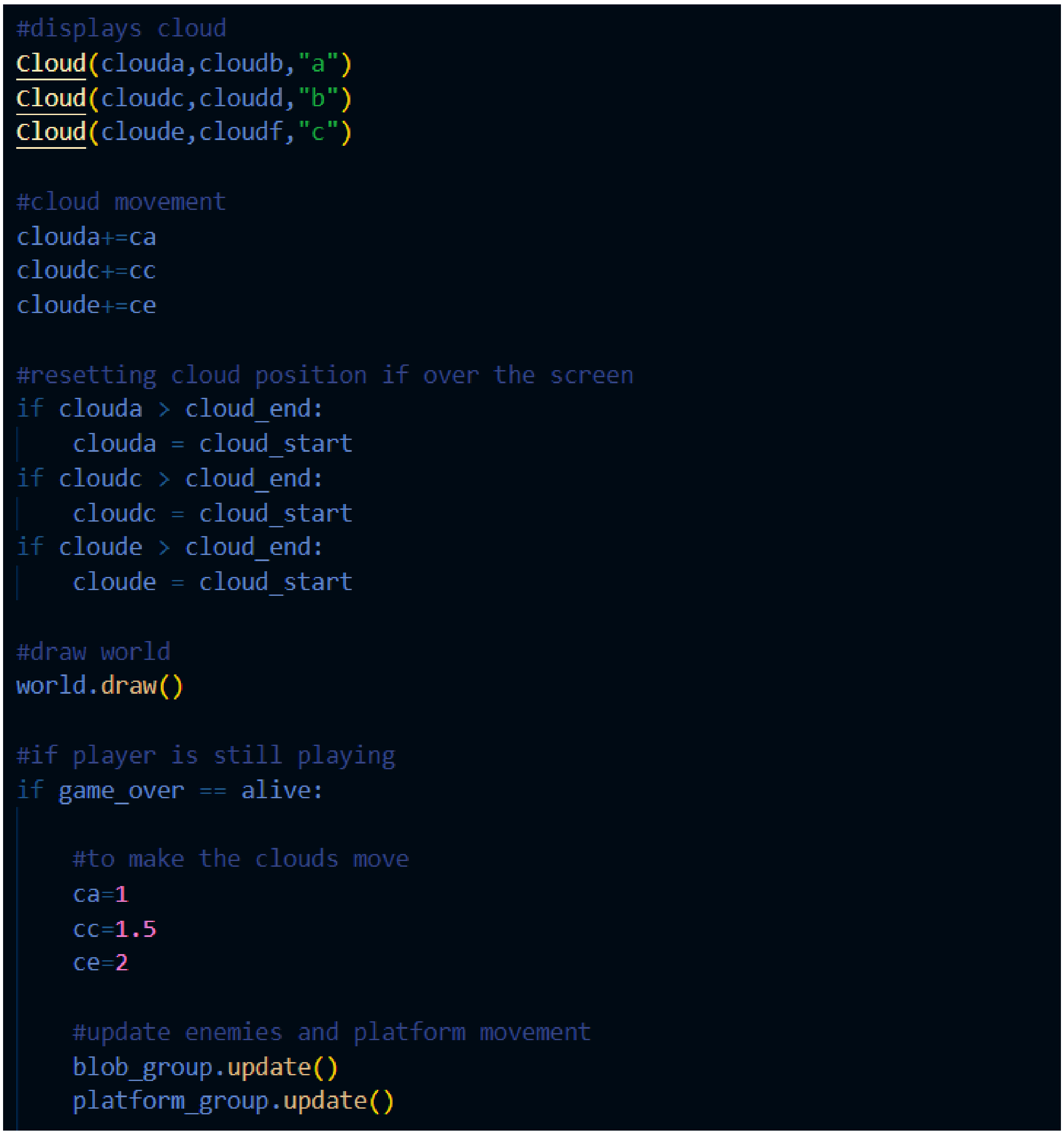
```python
#initialize player location
player = Player(100, screen_height - 130)

#add groups
blob_group = pygame.sprite.Group()
platform_group = pygame.sprite.Group()
lava_group = pygame.sprite.Group()
heart_group = pygame.sprite.Group()
exit_group = pygame.sprite.Group()

#load in level data and create world
world = reset_level(level)

#create buttons
restart_button = Button(screen_width // 2 - 50, screen_height // 2 + 100, restart_img)
start_button = Button(screen_width // 2 - 210, screen_height // 2 - 260, start_img)
exit_button = Button(screen_width // 2 - 210, screen_height // 2 + 50, exit_img)
back_button = Button(screen_width // 2 - 50, screen_height // 2 + 200, exitsm_img)
```

This declares the initial player location and add groups using pygame.sprite.Group() function which stores tiles and world variable loads in the level data stored in reset_level function and buttons are created using the button class with parameters of coordinates and images.



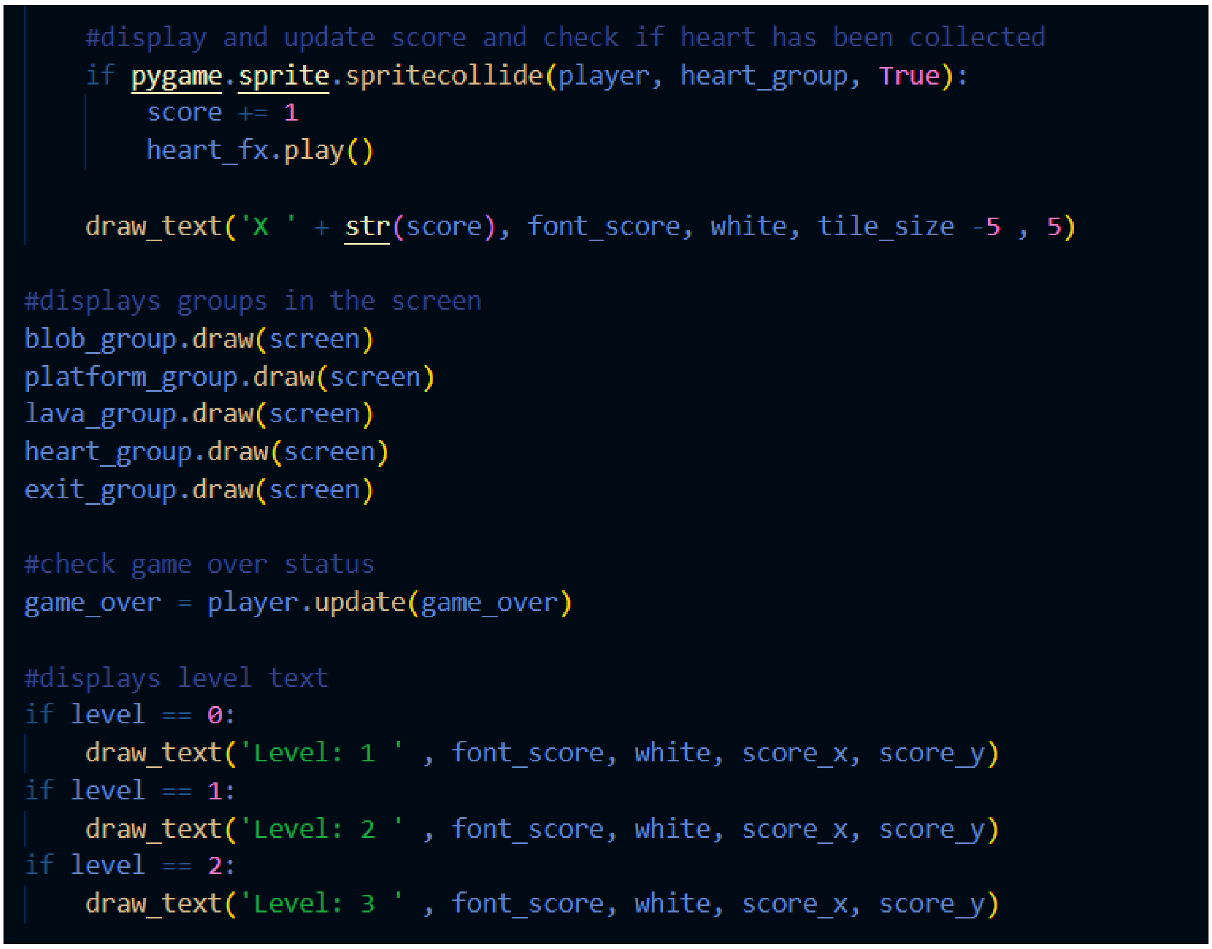
```python
#loop the game
run = True
while run:

    clock.tick(fps)

    screen.blit(bg_img, (0, 0))

    #main menu
    if main_menu == True:
        draw_text("Heart Eater", main_font, white, 210,70)
        if exit_button.draw():
            #exit loop
            run = False
        if start_button.draw():
            #exit main menu
            main_menu = False
    else:
```

Run = True to loop the game and inside the loop fps and background is declared and it displays the main menu, inside the main menu there is a Heart Eater title drawn by the draw_text function and exit button and start button drawn, if the exit button is clicked then the program will stop and if the start button is clicked then the game will start

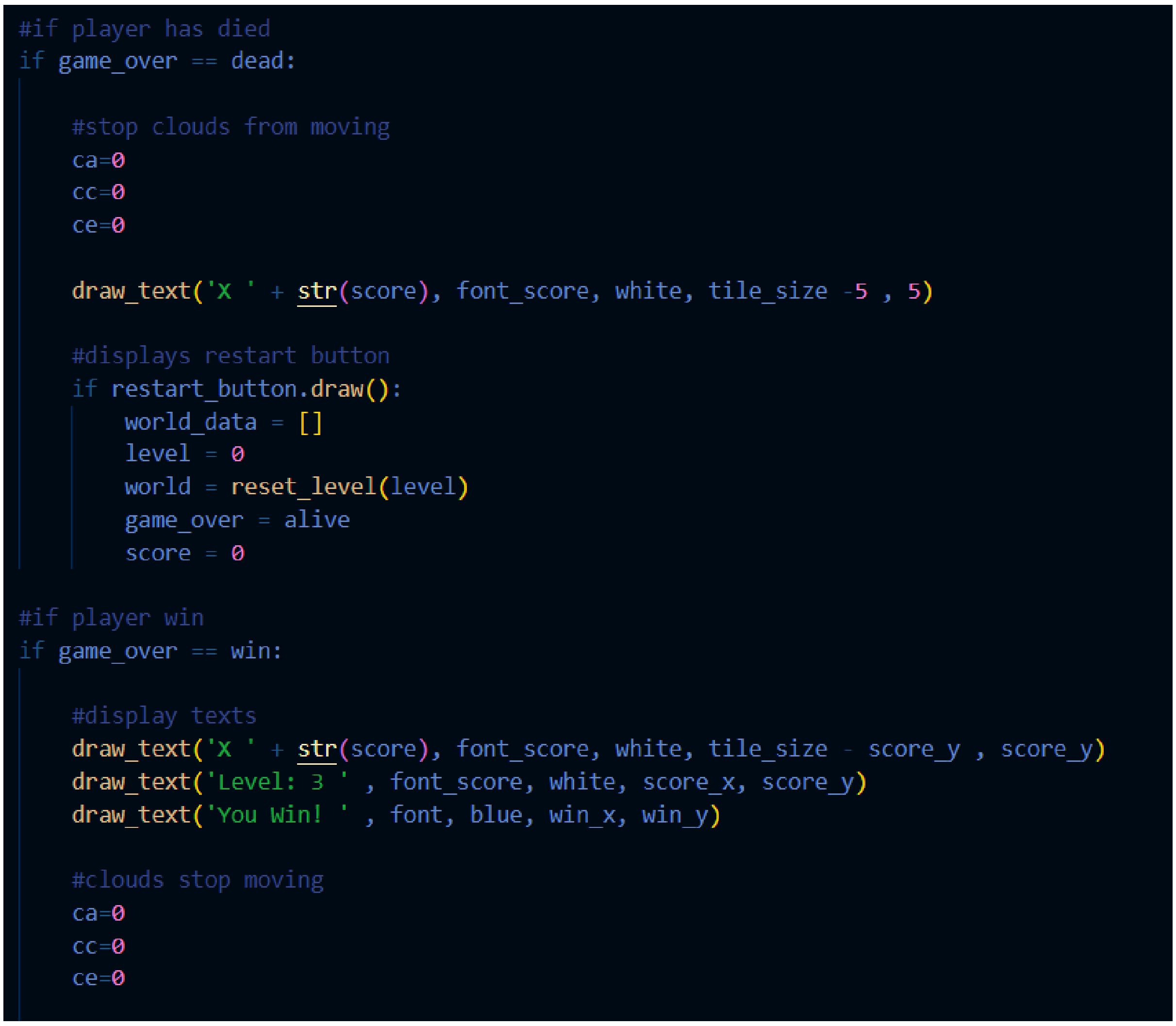


```
#displays cloud
Cloud(clouda,cloudb,"a")
Cloud(cloudc,cloudd,"b")
Cloud(cloude,cloudf,"c")

#cloud movement
clouda+=ca
cloudc+=cc
cloude+=ce

#resetting cloud position if over the screen
if clouda > cloud_end:
    clouda = cloud_start
if cloudc > cloud_end:
    cloudc = cloud_start
if cloude > cloud_end:
    cloude = cloud_start

#draw world
world.draw()

#if player is still playing
if game_over == alive:

    #to make the clouds move
    ca=1
    cc=1.5
    ce=2

    #update enemies and platform movement
    blob_group.update()
    platform_group.update()
```

Since the start button is clicked, the main menu value is False which will run the game, It displays clouds and the movement of the cloud, the world is drawn and if the player is alive, clouds are moving and enemies and platform are updated.

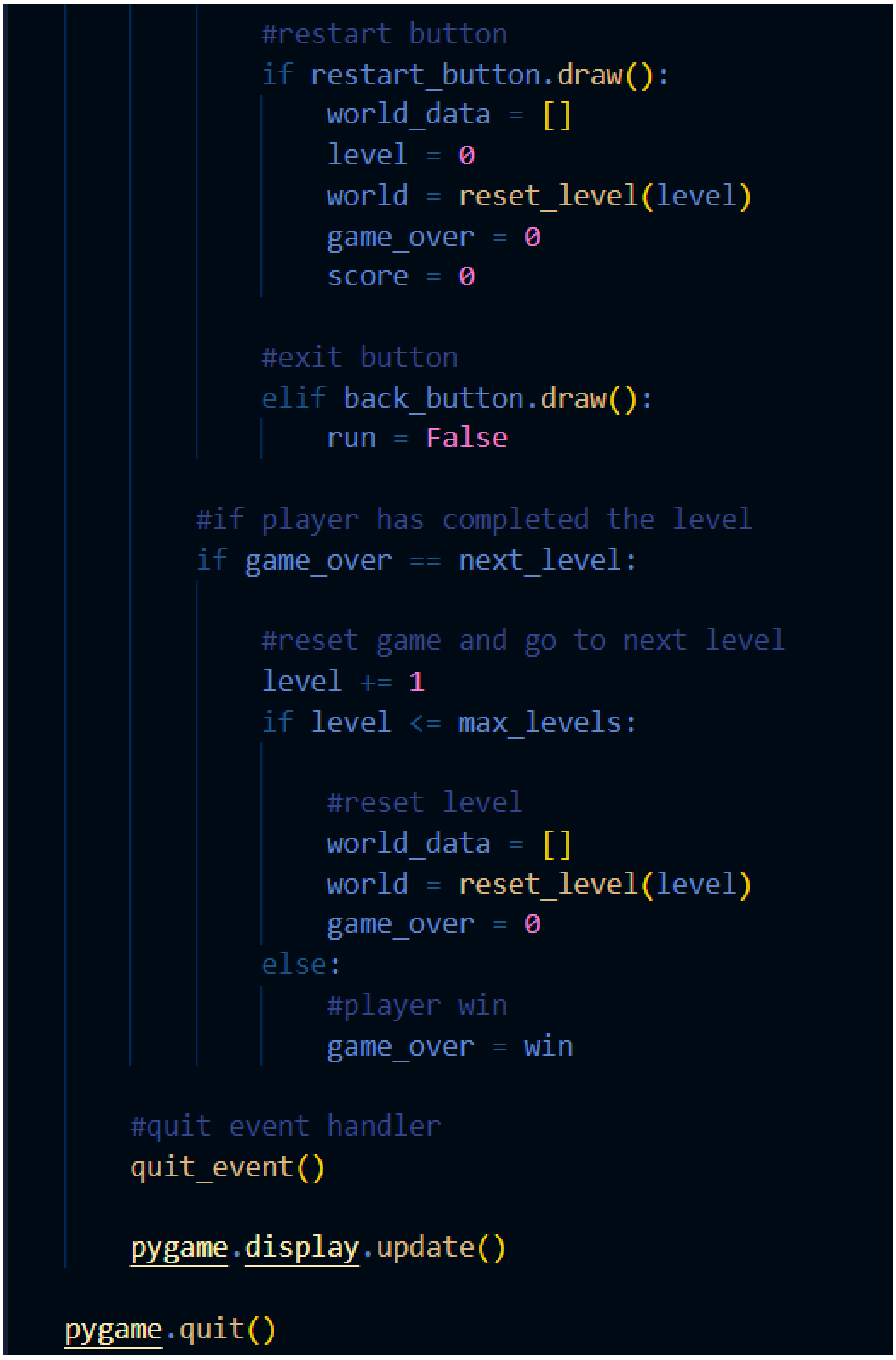

```
    #display and update score and check if heart has been collected
    if pygame.sprite.spritecollide(player, heart_group, True):
        score += 1
        heart_fx.play()

    draw_text('X ' + str(score), font_score, white, tile_size -5 , 5)

#displays groups in the screen
blob_group.draw(screen)
platform_group.draw(screen)
lava_group.draw(screen)
heart_group.draw(screen)
exit_group.draw(screen)

#check game over status
game_over = player.update(game_over)

#displays level text
if level == 0:
    draw_text('Level: 1 ' , font_score, white, score_x, score_y)
if level == 1:
    draw_text('Level: 2 ' , font_score, white, score_x, score_y)
if level == 2:
    draw_text('Level: 3 ' , font_score, white, score_x, score_y)
```

If player collides with a heart, a sound effect will be played. There is a score displayed on top of the screen by using the draw_text function and tile groups are drawn to the screen which covers the world, game_over status are updated and the level text are displayed depending on the level.



```
#if player has died
if game_over == dead:

    #stop clouds from moving
    ca=0
    cc=0
    ce=0

    draw_text('X ' + str(score), font_score, white, tile_size -5 , 5)

    #displays restart button
    if restart_button.draw():
        world_data = []
        level = 0
        world = reset_level(level)
        game_over = alive
        score = 0

#if player win
if game_over == win:

    #display texts
    draw_text('X ' + str(score), font_score, white, tile_size - score_y , score_y)
    draw_text('Level: 3 ' , font_score, white, score_x, score_y)
    draw_text('You Win! ' , font, blue, win_x, win_y)

    #clouds stop moving
    ca=0
    cc=0
    ce=0
```

(If the player lost)
If player has died, clouds movement are stopped by setting the movement variable to 0 and the total score and a restart button will be displayed with the restart button resetting all the variables.
(If the player win)
If the player has completed all the level by turning the game_over == win, a “YOU WIN!” text is displayed and the clouds stop moving.



```
            #restart button
            if restart_button.draw():
                world_data = []
                level = 0
                world = reset_level(level)
                game_over = 0
                score = 0

            #exit button
            elif back_button.draw():
                run = False

        #if player has completed the level
        if game_over == next_level:

            #reset game and go to next level
            level += 1
            if level <= max_levels:

                #reset level
                world_data = []
                world = reset_level(level)
                game_over = 0
            else:
                #player win
                game_over = win

    #quit event handler
    quit_event()

    pygame.display.update()

pygame.quit()
```

(If the player win)
A restart button and an exit button is displayed.

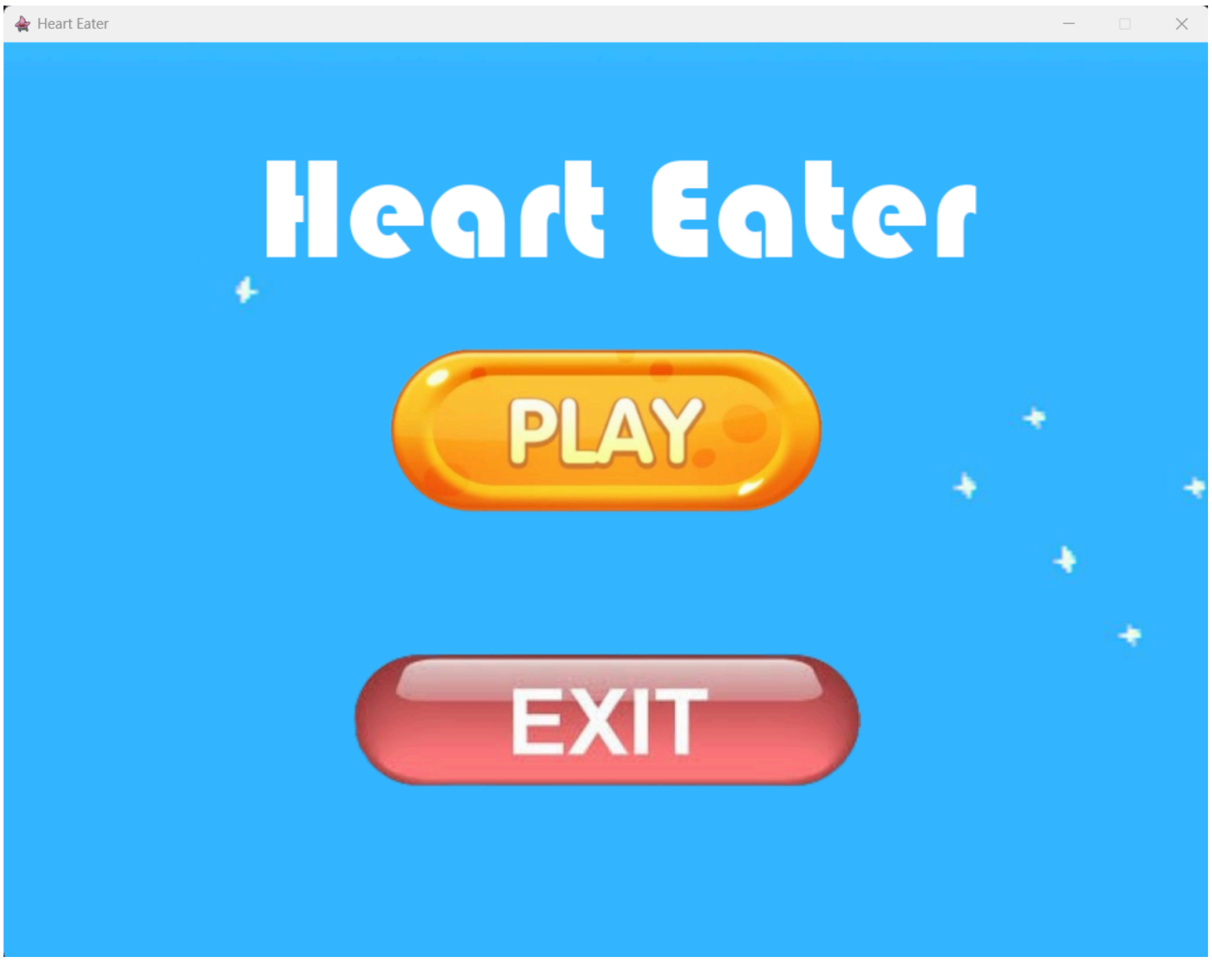
If the player has completed a level and the next level is still valid, the level will increase by 1 and this resets the world data by emptying the

list and add a new data caused by different level value, else if the next level is invalid (player has completed all levels) player has win

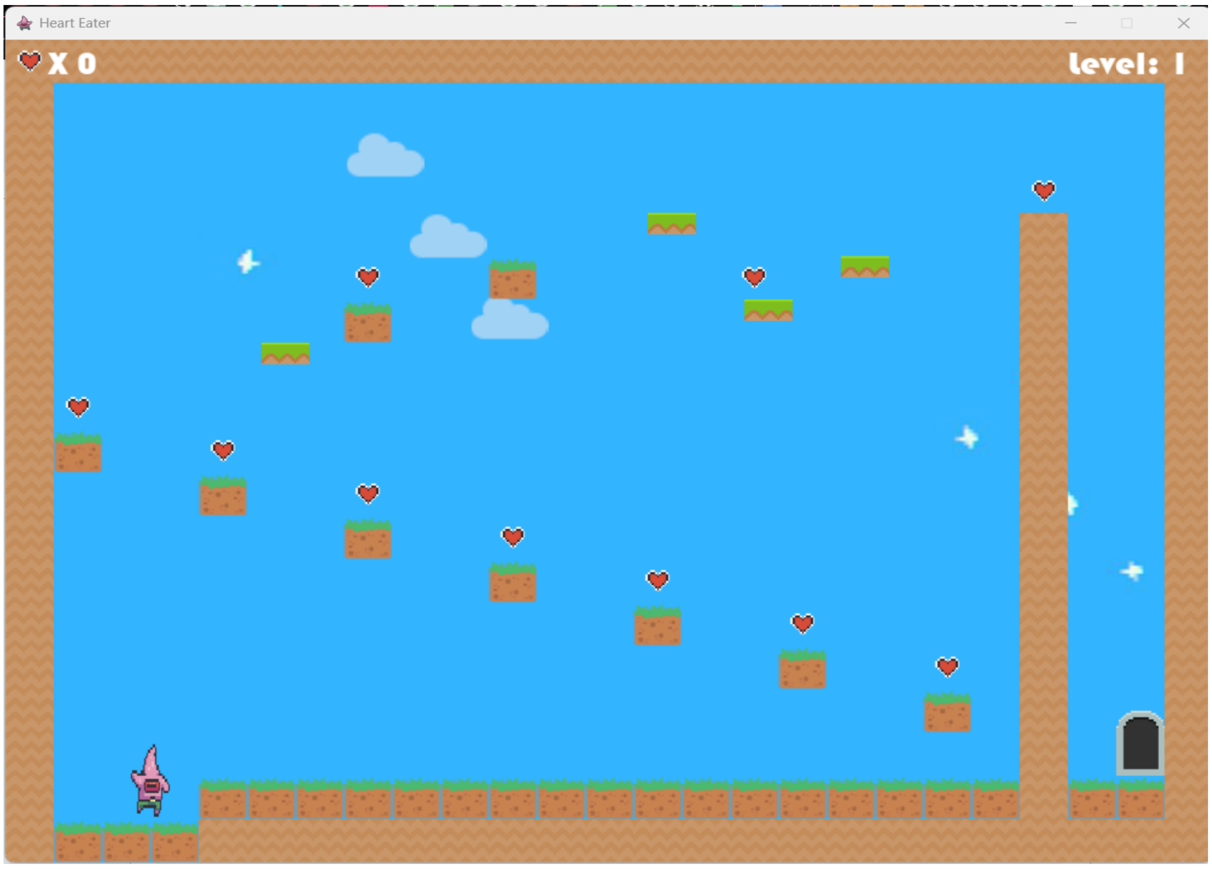
Quit_event() function is called to exit the program properly, pygame.display.update() updates what is going on in the program and pygame.quit() uninitialize all pygame module that is called

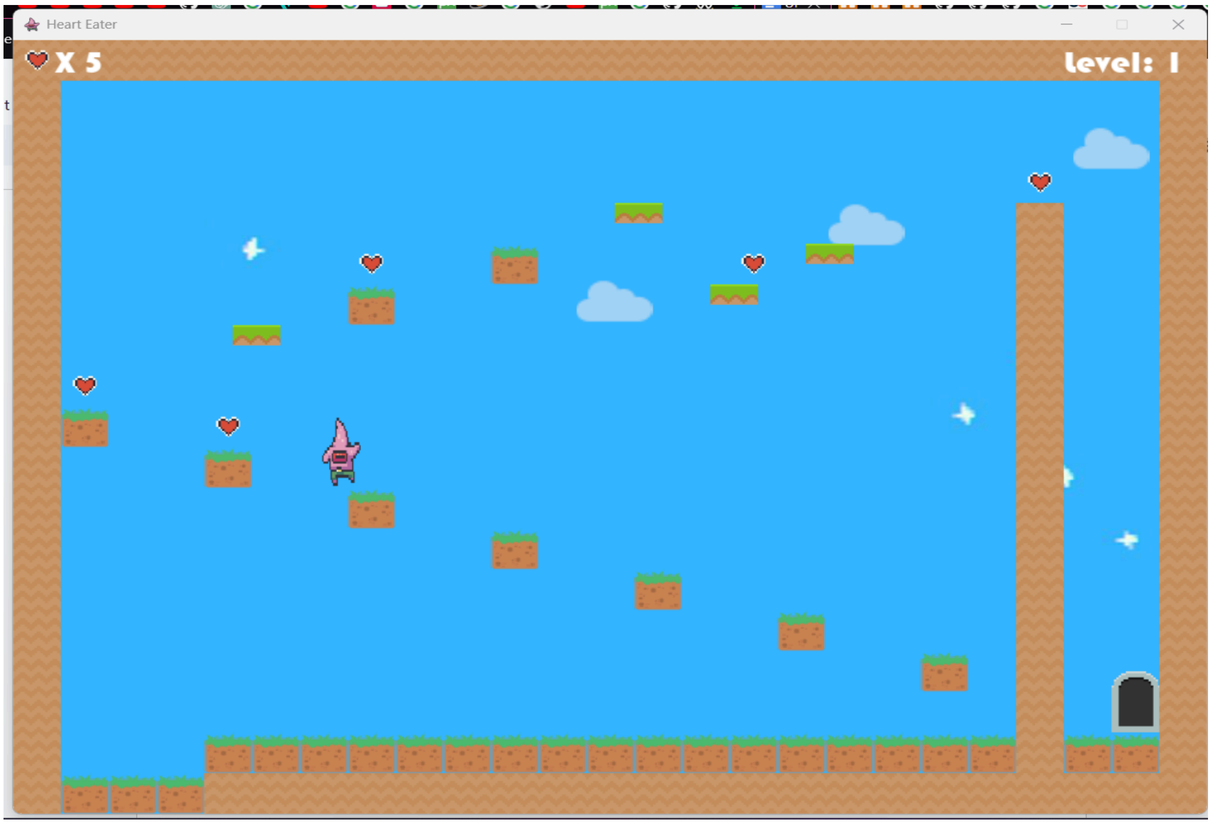
E. Evidence of working program

Menu

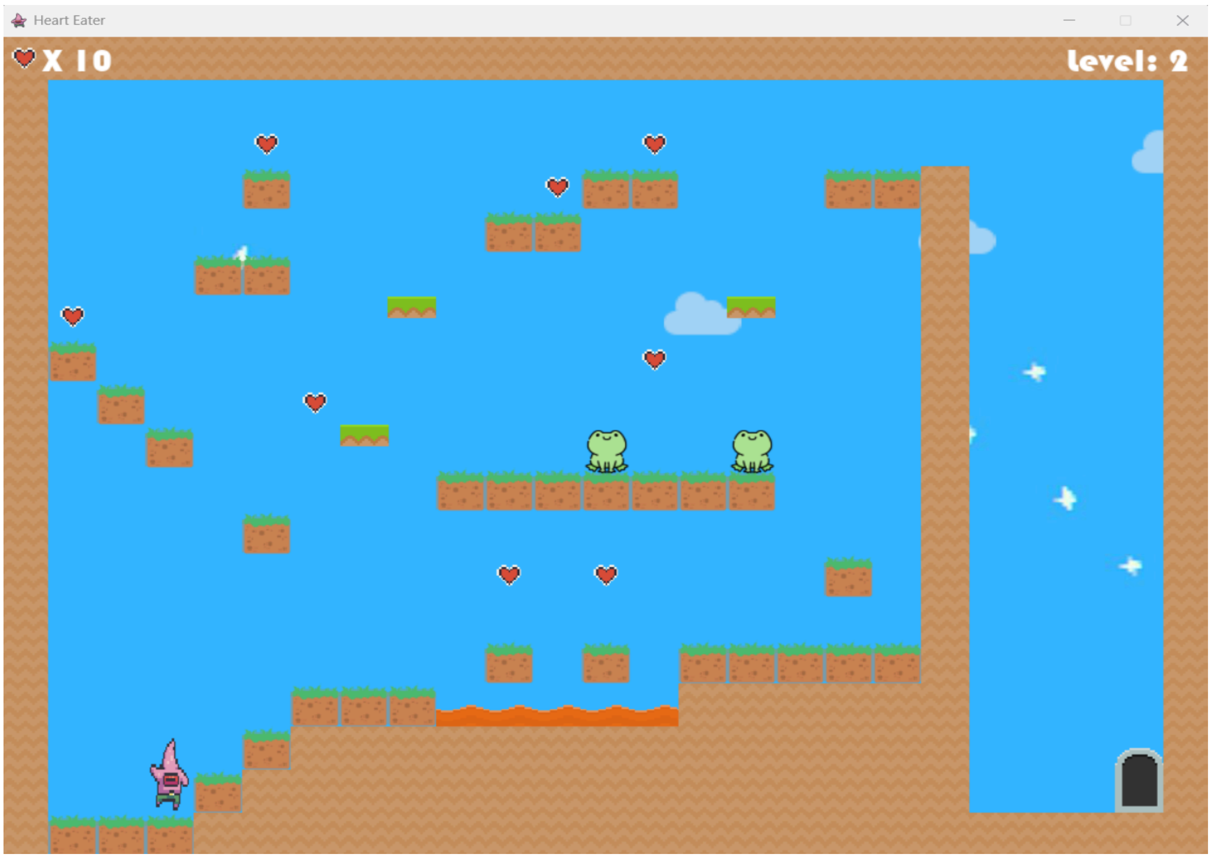


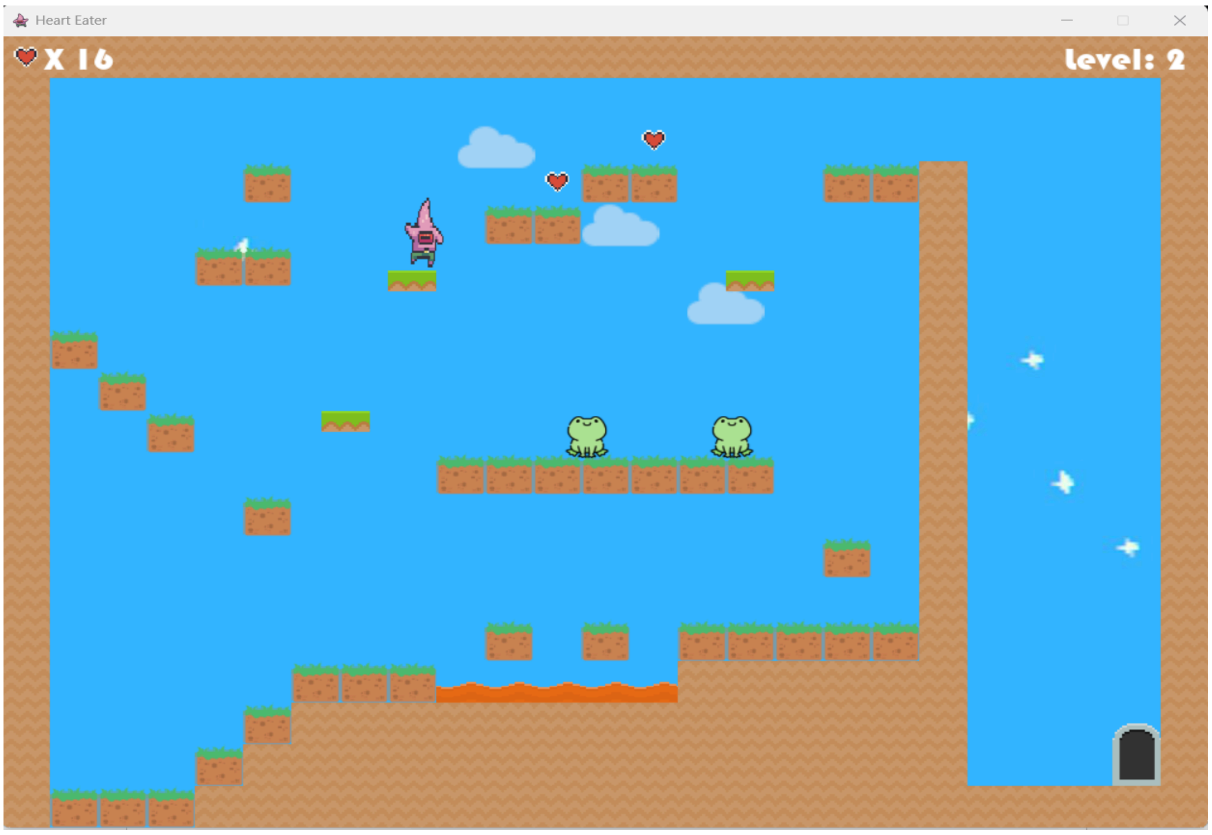
Level 1



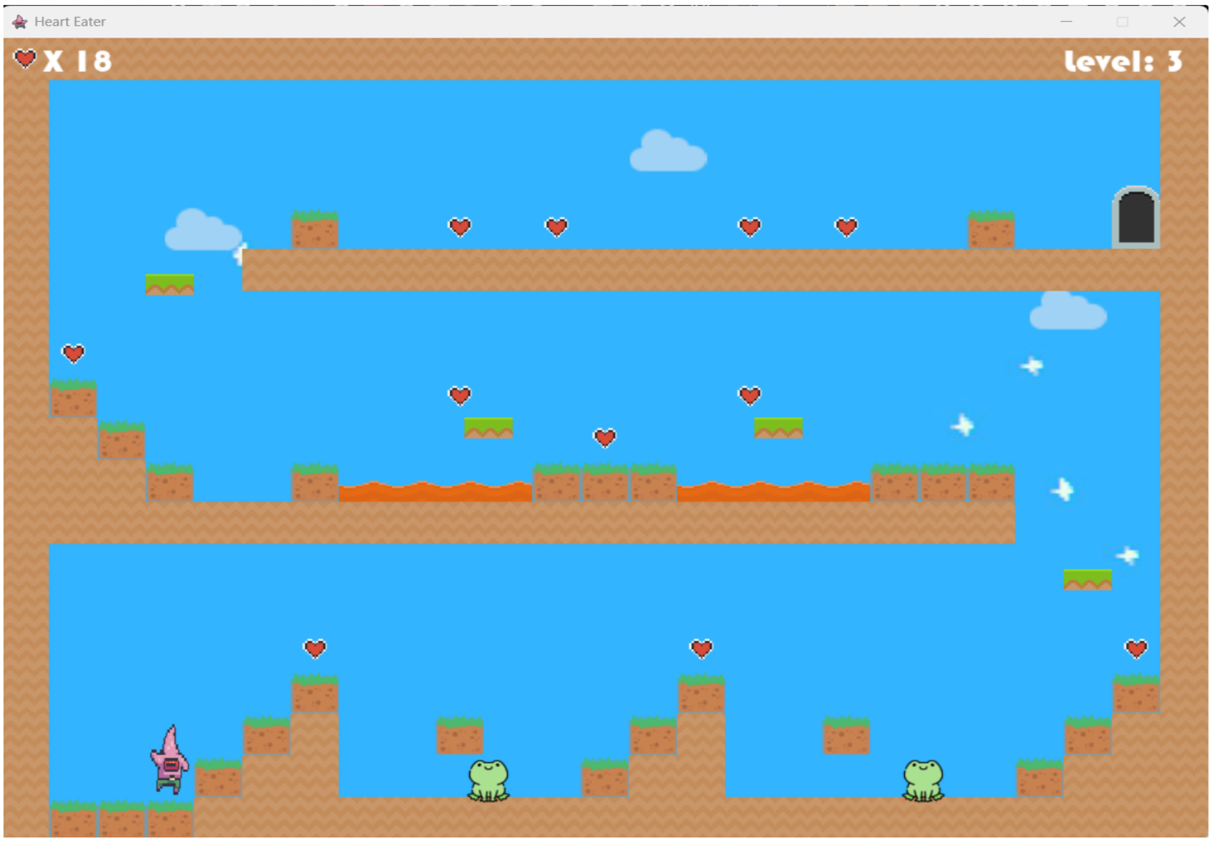


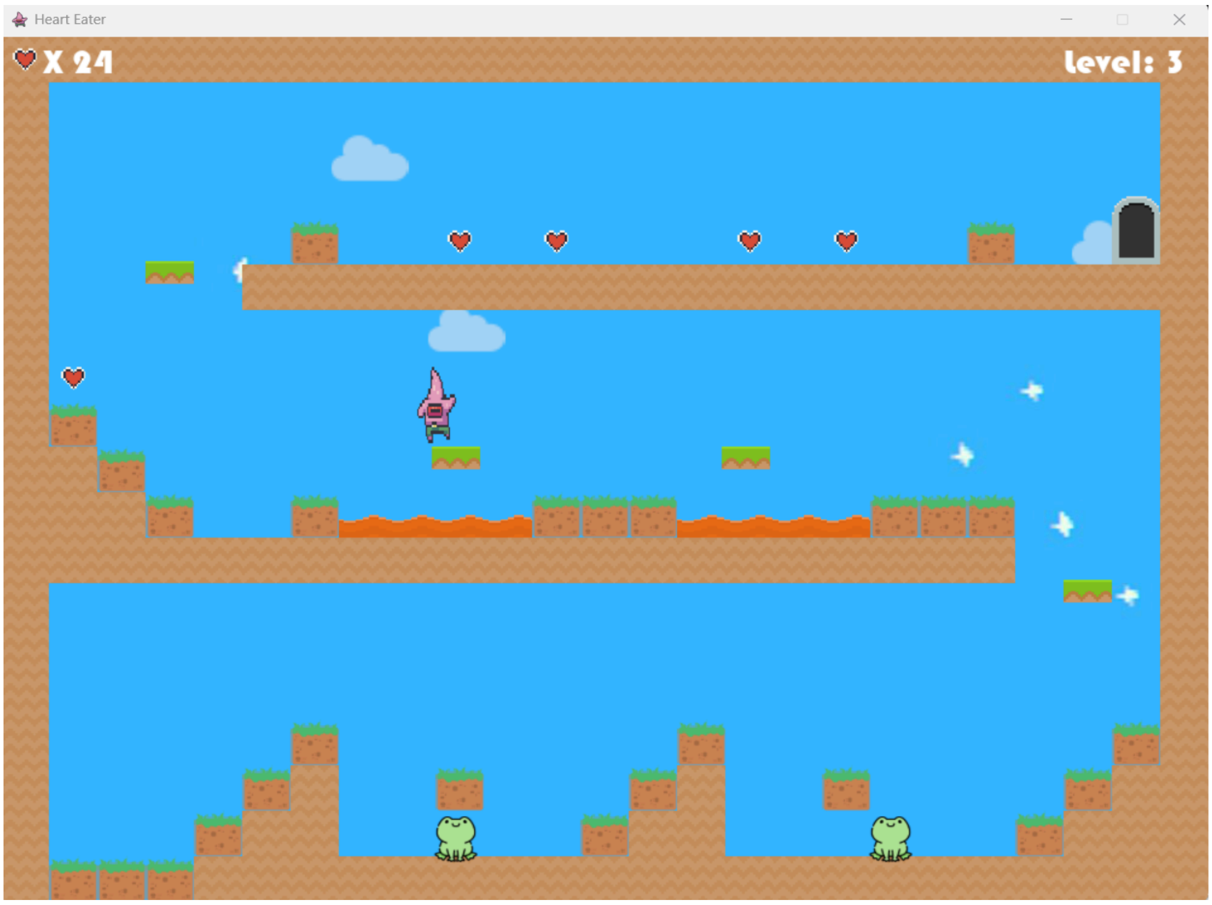
Level 2



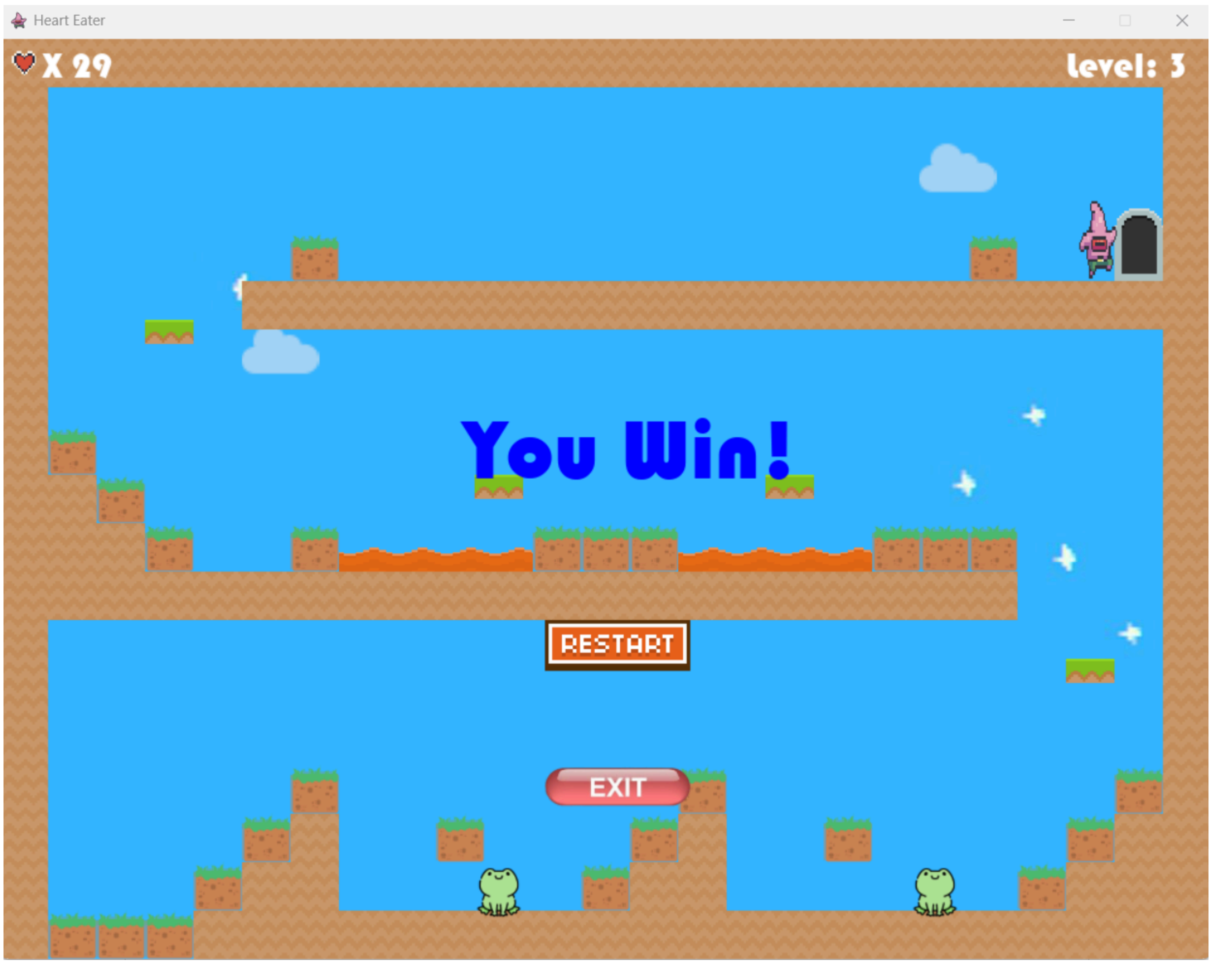


Level 3

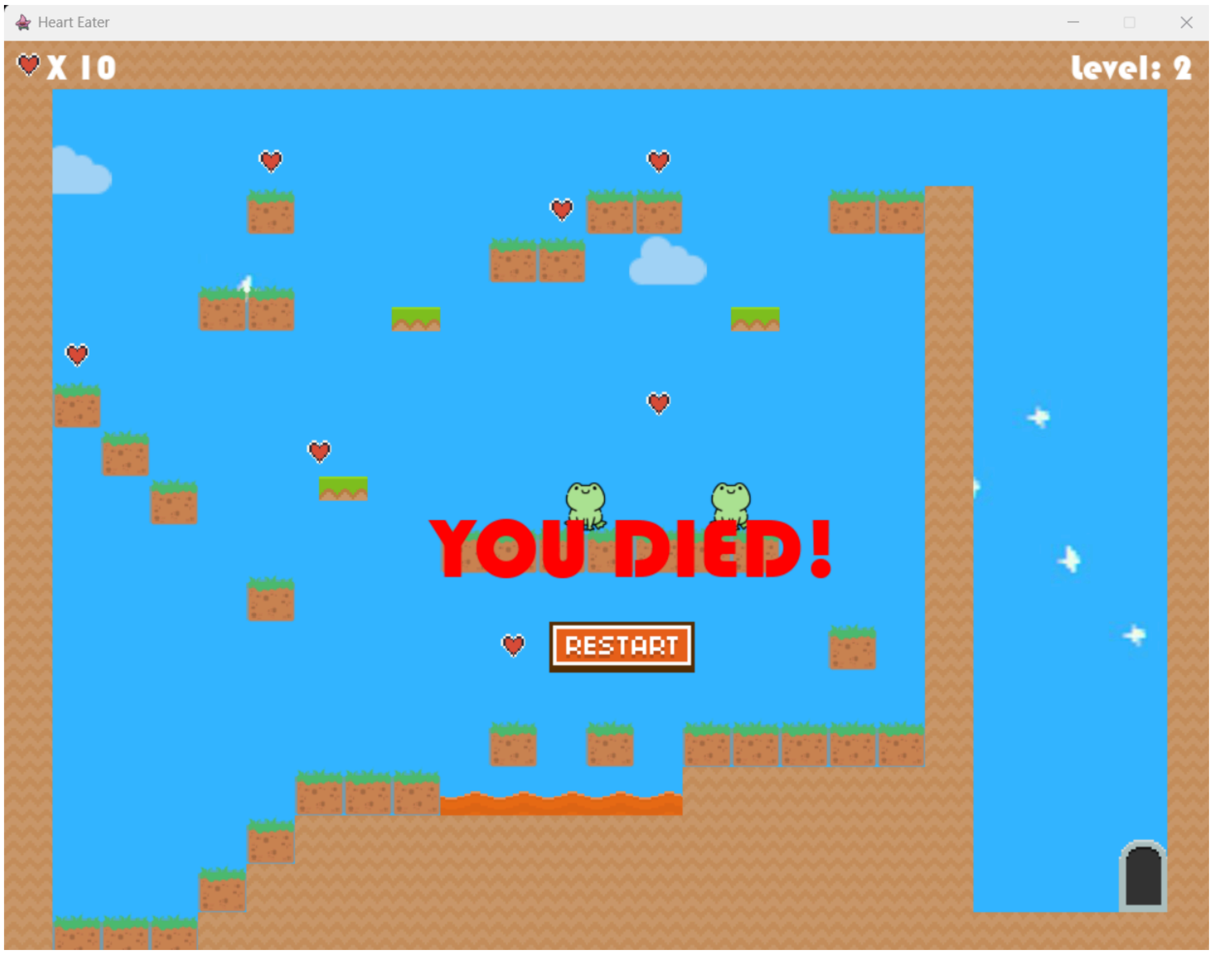




Win



Lose



F. Lesson Learned

I learned a lot by doing this final project, i learned to create games using pygame library which i was unfamiliar with and i learned that making a game was harder than i expected. I also developed some logics which i was not aware of by encountering some setbacks during developing the game and i learned on about how to manage time better.

G. Resources

Project program files : https://github.com/gamakagami/Algo-Prog-FP

Images : https://itch.io/game-assets/free/genre-platformer

demo:https://drive.google.com/file/d/1RZRDoZ5iO1TU-QKcSo_GTIMWvfNtS8WO/view?usp=sharing