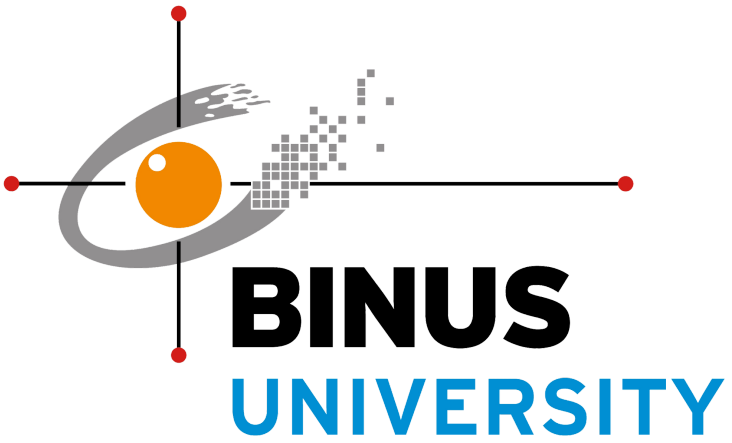
Odd Semester (2023)



BINUS UNIVERSITY

ALGORITHM AND PROGRAMMING FINAL PROJECT

PROJECT REPORT

Student Information :

Name: Rafael Anderson **NIM:** 2702255981 **Class:** L1AC

Course Information:

Course Code: COMP6047001 **Lecturer:** Jude Joseph Lamug Martinez, MCS

A. Introduction

My project is a platformer game where users need to collect coins and finish all of the levels without dying in order to complete it, and I am using pygame to create all of that.

When the program is initially executed, users are directed into the main menu with an option to play or exit. Once the users click the play button, the users can choose which character they want to play as. After that, it redirects the users to the game which starts from stage 1 up to stage 4. Every time the user dies, they are redirected to another window with an option to retry or exit. Finally, if the user wins the game, they will be redirected into another pygame window stating that they have won the game, displaying their score as well.

Background

For our algorithm and programming final project, we are asked to create a comprehensive application that solves a problem. At first I was confused on what to make, but after watching some pygame tutorials, I decided to create a game using pygame. I initially wanted to create a top down RPG game, but the tutorials were quite confusing and hard to follow. After watching some more tutorials and trying to understand some new concepts, I decided to create a parkour platformer game.

Problem Identification

One of the problems I want to solve is stress. Engaging in a game can help individuals escape reality for a while and relax for a bit. Even though it

isn't a major problem, it is one of the most common problems people suffer from. Of course, playing too much is also not good, so I decided to keep my game simple and short with a friendly user interface.

B. Project Specifications

Game Name: The Crusaders

1. Game Concept

The game consists of monsters / enemies moving left and right. The player needs to avoid them and finish all stages by reaching the door. Their score is determined by the amount of coins collected that will be displayed at the end of the game if they win.

2. Game Objective

The game's objective is to collect as many coins as possible, and reach the final stage alive.

3. Game Display

The game uses pixelated looking images and fonts. It also consists of a friendly user interface for every action that is done by the user. Such as the main menu, character choosing, character dying, and the user winning the game. There are also character animations when they are idle, falling, jumping and running. Once the game is finished (The user won) the score will be displayed as well as some buttons.

4. **Game mechanics**

The game utilizes standard gaming buttons to move, where the key A is to move left, key D to move right and key W to jump.

5. **Game Physics**

The enemies are constantly moving left and right. I use colliderect to check the collision between the character's rectangle and the enemy's rectangle. If they do collide, the user will die. I also use the colliderect for the blocks, so the character's can't move through the blocks, and are able to stand on the blocks. I also made it so that the character will always go back down after jumping like real life.

6. **Game Input**

   a. Key pressed
   b. Key released
   c. Mouse clicks

7. **Game Output**

- Picture of the character chosen along with the animations according to the player's movement (Animations available inside the character1 and character2 folders according to the character chosen)
- Necessary texts using the Minecraftia - Regular.ttf font style
- Different world structures for each level
- Enemies
- Coins
- Current Stage
- Player's score
- Background images
- Buttons

- Jump sound effect
- Dying sound effect

8. File Names

- Character1 (folder) - It is the folder that contains all assets needed such as animations for the first character.
- Character2 (folder) - It is the folder that contains all assets needed such as animations for the second character
- Minecraftia - Regular.ttf - It is the font - style that I use to write texts
- back.png - It is the image for the back button
- bg.png - It is the background image for the main menu and others
- cancel.png - It is the image for the cancel button
- caution.png - Image for a caution sign
- coin.png - Image for the coins
- death.mp3 - Is the sound played when the character dies
- door.png - Image for the door
- exit.png - Image for the exit button
- game_wallpaper.jpg - Background image in the main game
- gamelogo.png - Icon for the game
- guard.png - Image for the slime enemy
- guard2.png - Image for the ghost enemy
- guard3.png - Image for the red guard enemy
- jump.mp3 - Sound that is played when the character jumps
- lava.png - Image for the lava
- logo.png - Image for the logo of the pygame window
- main.py - Main code
- play.png - Image for the play button
- retry.png - Image for the retry button
- select.png - Image for the select button

- tile.png - Image for the blocks

9. Modules / Library Used

1. **Pygame**

   Pygame is a module that I used to create most of the game. It is normally used to write video games.

2. **Sys**

   I used this module so it will be smoother when exiting the program.

3. **Mixer**

   The mixer module is also from pygame. I use the mixer module to play audio such as jumping sound effects and death sound.

10. Resources

Character1 Resources: Warrior-Free Animation set V1.3 by Clembod (itch.io)

Character2 Resources: Animated Pixel Adventurer by rvros (itch.io)

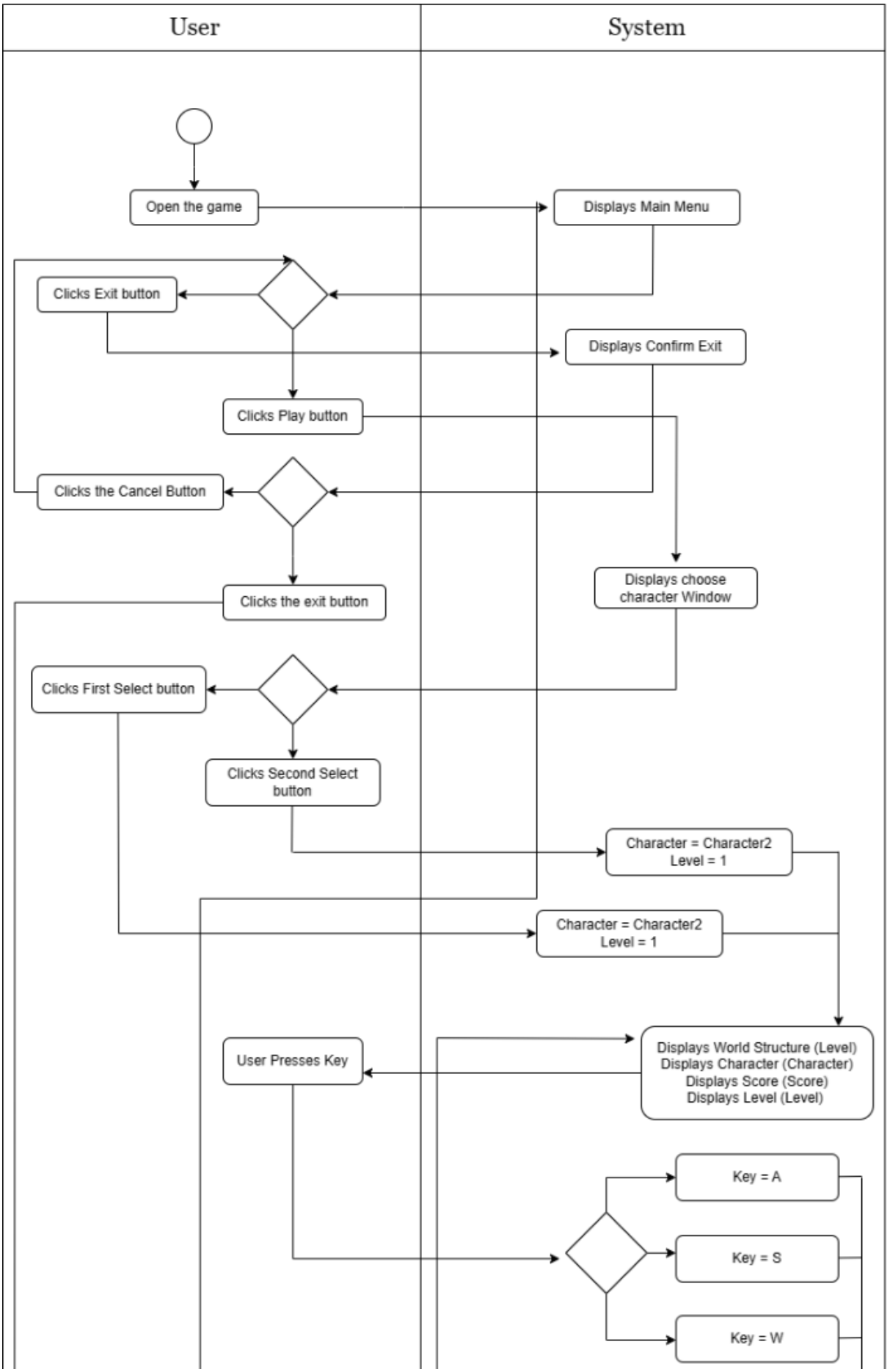
Button images are taken from google

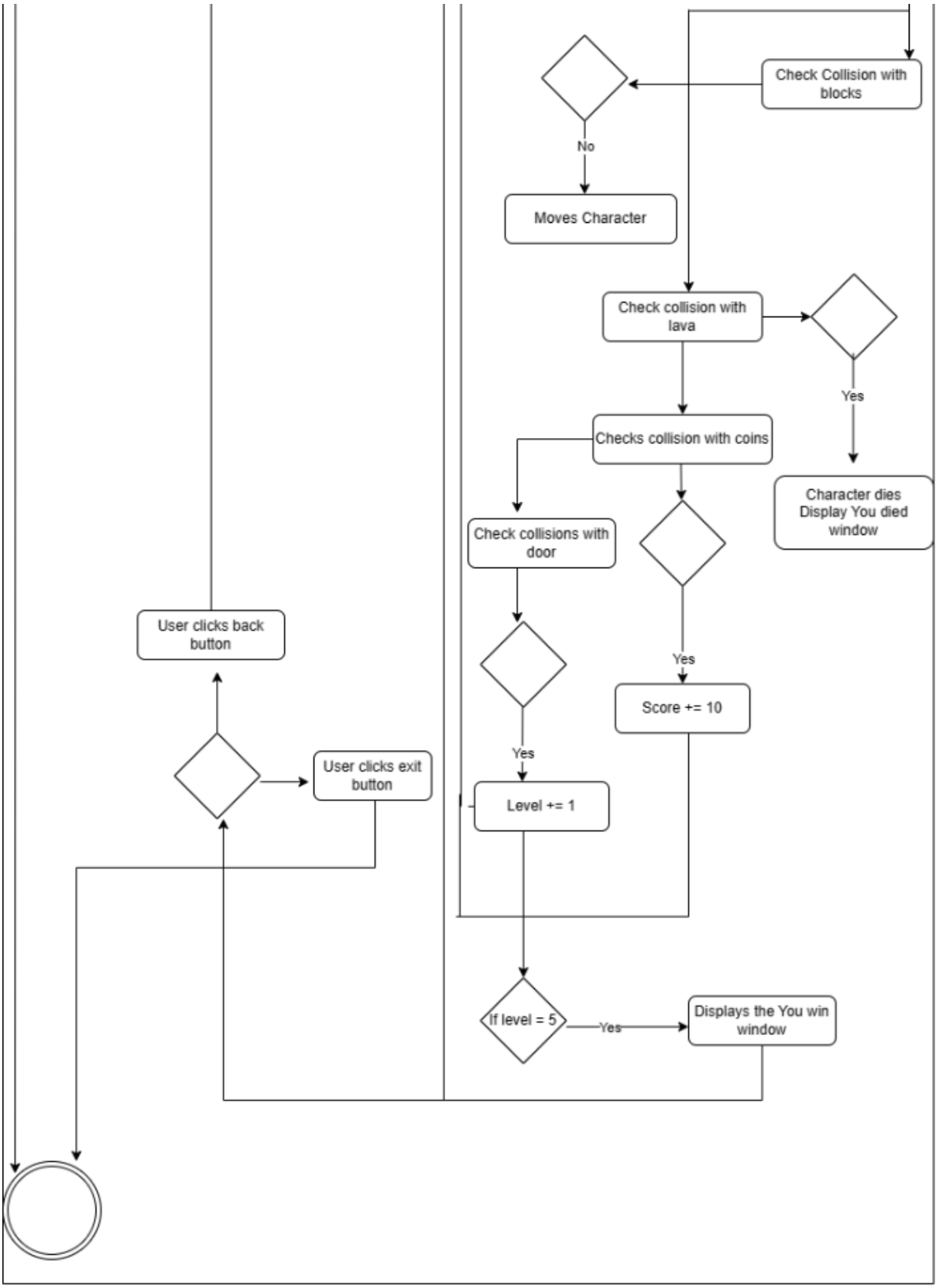
Sound effects: Storyblocks

C. Solution Design

1. **Activity diagram**

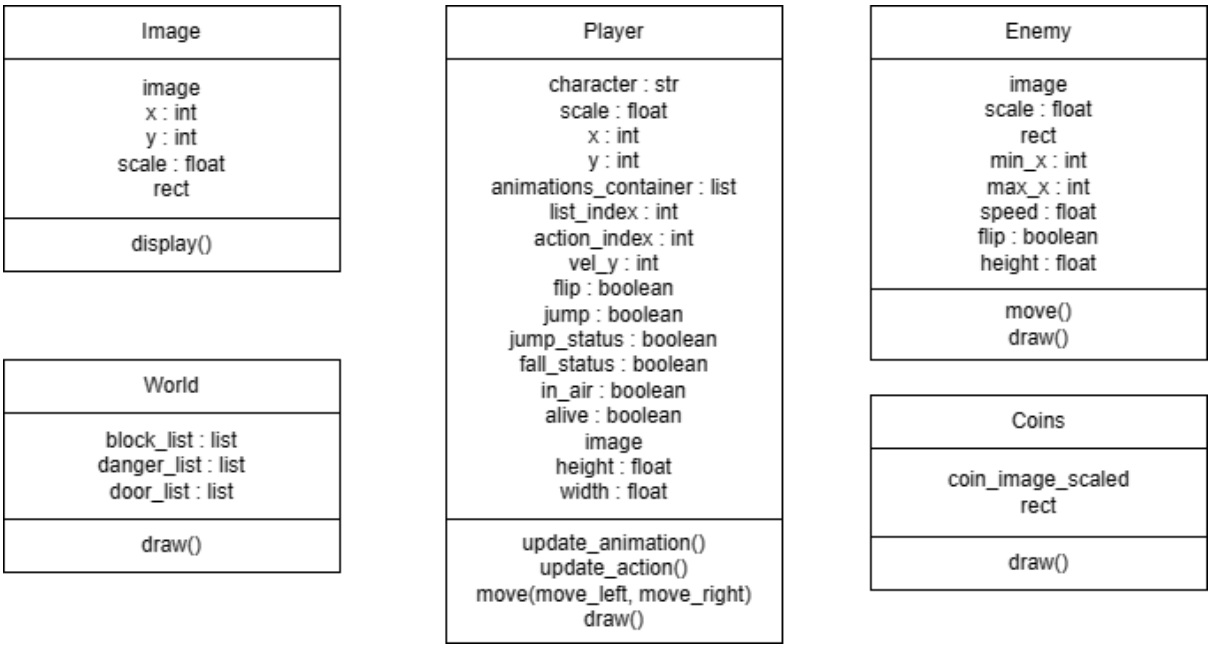
   The following diagram is the activity diagram for the application
   The full image is available in my github


User
System
Open the game
Displays Main Menu
Clicks Exit button
Displays Confirm Exit
Clicks Play button
Clicks the Cancel Button
Displays choose character Window
Clicks the exit button
Clicks First Select button
Clicks Second Select button
Character = Character2
Level = 1
Character = Character2
Level = 1
User Presses Key
Displays World Structure (Level)
Displays Character (Character)
Displays Score (Score)
Displays Level (Level)
Key = A
Key = S
Key = W


Check Collision with blocks
No
Moves Character
Check collision with lava
Yes
Character dies Display You died window
Checks collision with coins
Check collisions with door
Yes
Score += 10
Yes
Level += 1
If level = 5
Yes
Displays the You win window
User clicks back button
User clicks exit button

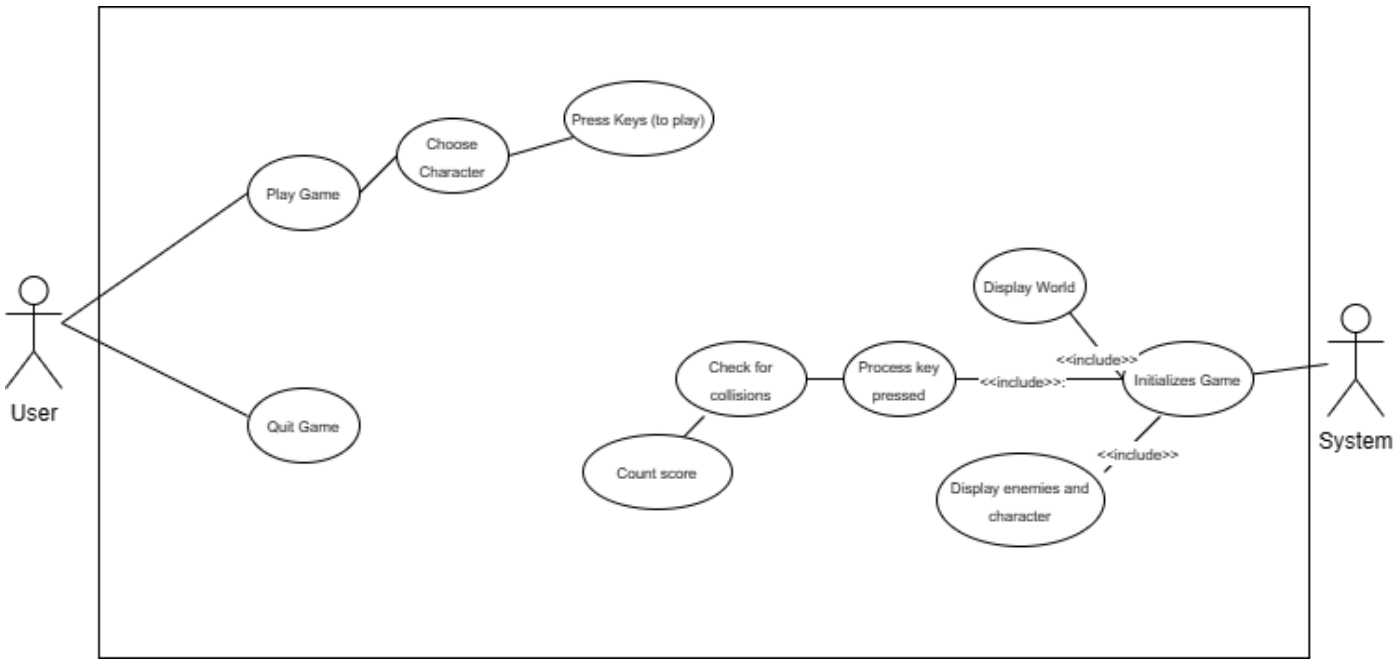
2. Class diagram

The following image is the class diagram for my game



3. Use case diagram

The image below is the use case diagram for the game



4. Algorithms

This section is where I will be explaining my code

```
import pygame
import sys
from pygame import mixer
```

Imports all of the necessary modules / libraries needed for the program

```
pygame.init()
mixer.init()
```

Initializing the modules and library

```
# The size for the pygame window, and setting the window
screen_width = 850
screen_height = 500
screen = pygame.display.set_mode((screen_width,screen_height))
```

This sets up the screen size for the pygame window.

```
# Importing buttons and scaling it
exit = pygame.image.load("exit.png").convert_alpha()
exit_scaled = pygame.transform.scale(exit,(150,70))

# Loading in sound effect
jump_sound = pygame.mixer.Sound("jump.mp3")
jump_sound.set_volume(0.3)
death_sound = pygame.mixer.Sound("death.mp3")
death_sound.set_volume(0.6)
```

This section just loads in the necessary images and sound effects. Also scaling it in terms of size and the volume.

```
FPS = 60
font_size = 60

clock = pygame.time.Clock()

# It is the size for each block in the world (blocks, lava and doors)
tile_width = 30
tile_height = 30

left_mouse_clicks = 1
```

The FPS is used later in Clock.tick, so that the number of frames per second stays constant, the clock is necessary to make this happen. The tile_width and tile_height refers to the size of each block in the world (such as tiles, lava and doors). We initialize left_mouse_clicks to avoid using magic numbers later on.

```
# The Image class helps displaying images and scaling it when necessary
class Image:

    def __init__(self, image, x, y, scale):
        self.image = image
        self.x = x
        self.y = y
        self.scale = scale
        self.image = pygame.transform.scale(self.image,(int(self.image.get_width()*self.scale),int(self.ima
        self.rect = self.image.get_rect(topleft = (self.x, self.y))
```

As seen here, we have an Image class and we're initializing all of the necessary data needed. I use this class to display images (Mostly buttons), scaling them, and check for mouse clicks by using the self.rect. Self.rect represents the rectangle shape of the image. Later on, I used the colliderect to help check collisions of the mouse.

```
    def display(self):
        screen.blit(self.image,(self.x,self.y))
```

Still in the Image class, this draw function displays the image on the screen with its respective x and y coordinates.

```python
# The Player class is the class that animates the character, displaying the character, moving the character
class Player:

    # Initializing
    def __init__(self,character, scale, x, y):
        self.character = character
        self.scale = scale
        self.x = x
        self.y = y
        self.animations_container = []
        self.list_index = 0
        num_of_idle_images = 5
        self.action_index = 0
        self.vel_y = 0
        self.flip = False
        self.jump = False
        num_of_run_images = 0
        num_of_jump_images = 0
        num_of_fall_images = 0
        self.gravity = 0.75
        self.jump_status = False
        self.fall_status = False
        self.in_air = False
        self.alive = True
```

Moving on to the next class, we have the main class that displays the player , animating it and moving it. As shown, we initialize it first, taking in character, scale, x and y. The character is according to the character chosen in the choosing character section. The scale, x, and y coordinates are dependent on the character and the level they're in.

```python
    # The action_list acts as a temporary list to append itself containing different actions into the
    action_list = []

    # Storing scaled idle images according to the character chosen
    for i in range(num_of_idle_images):

        # My file names are numbers so it will make it easy storing it.
        img = pygame.image.load(f"{character}/idle/{i}.png")
        img = pygame.transform.scale(img, (int(img.get_width() * self.scale), int(img.get_height() *
        action_list.append(img)
    self.animations_container.append(action_list)

    # Both characters have different number of images for each action
    action_list = []
    if self.character == "character1":
        num_of_run_images = 8
        num_of_jump_images = 3
        num_of_fall_images = 3

    elif self.character == "character2":
        num_of_run_images = 6
        num_of_jump_images = 4
        num_of_fall_images = 2
```

```
# The same concept as before but for run images.
for i in range(num_of_run_images):

    img = pygame.image.load(f"{self.character}/run/{i}.png")
    img = pygame.transform.scale(img, (int(img.get_width() * self.scale), int(img.get_height() * se
    action_list.append(img)
self.animations_container.append(action_list)

# For jump images.
action_list = []
for i in range(num_of_jump_images):
    img = pygame.image.load(f"{self.character}/jump/{i}.png")
    img = pygame.transform.scale(img, (int(img.get_width() * self.scale), int(img.get_height() * se
    action_list.append(img)
self.animations_container.append(action_list)

# For fall images
action_list = []
for i in range(num_of_fall_images):
    img = pygame.image.load(f"{self.character}/fall/{i}.png")
    img = pygame.transform.scale(img, (int(img.get_width() * self.scale), int(img.get_height() * se
    action_list.append(img)
self.animations_container.append(action_list)
```

The action list is a temporary list which stores every action for each character. That's why it is emptied after each loop. The for loop takes in the number of idle images which varies for each character (My file names are labeled from 0 to …. so we can take in the value of i). It loads the images, scales it and appends it to the action list, which is then appended to the self.animations_container. As both characters have different numbers of images for each action, the if section is necessary.

The first for loop is to store idle images, the second to store run images, the third to store jump images and the fourth to store fall images. Each action is later accessed by their index in the self.animations_container. All of these are according to the character type.

```
# Self.update_time helps to set a delay between images
self.update_time = pygame.time.get_ticks()
self.rect = img.get_rect(topleft = (x,y))

# Self.image is according to the self.action_index and self.list_index.
# Self.action_index represents the action, and the list_index represent the image in the list
self.image = self.animations_container[self.action_index][self.list_index]
self.height = self.image.get_height()
self.width = self.image.get_width()
```

The pygame.time.get_ticks() is to get the current time. The self.update_time is used to provide a delay between transition of images (in the self.animations_container). The self.image is defined, so that we can just change the self.action_index (Refers to the lists in the

self.animations_container) and the self.list_index (Refers to individual images in the temporary list inside the self.animations_container) to display the animation. The self.width and height will help with collisions between objects.

```
# This method switches between images in a list inside the self.animations_list.
def update_animation(self):

    # Time delay between images
    picture_transition_time = 100
    self.image = self.animations_container[self.action_index][self.list_index]

    # Self.update_time is continuously updated whenever the list_index is changed so there will be cons
    if pygame.time.get_ticks() - self.update_time > picture_transition_time:
        self.update_time = pygame.time.get_ticks()
        self.list_index += 1

    # If the list_index is equal or more to the length of an action in the self.animations_list, the se
    if self.list_index >= len(self.animations_container[self.action_index]):
        self.list_index = 0
```

We have the update_animation method, which updates the image for the current action. The picture_transition_time is the delay between the switch between each image. The first if condition, it checks if the current time subtracted by the self.update_time is more than the transition_time. If it satisfies the condition, it will update the self.update_time and switch the image to the next one.

The second if conditions checks if the image is the last one, if it is, it converts the self.list_index to 0. So the animation starts from the beginning once it ends.

```
# This method switches actions when needed. (When the upcoming action is different from the previous on
def update_action(self, action):

    # Change the action
    if self.action_index != action:
        self.action_index = action

        # Reset the image index
        self.list_index = 0

        #Update the self.update_time so the image transition delay still exists
        self.update_time = pygame.time.get_ticks()
```

We have the update_action method next. It is used to change the action according to the action it receives. The method is used later in the move method, it is used to switch actions, like from staying idle to moving, or from moving to jumping, etc. The self.list_index and self.update_time is resetted so the new animation also starts from the beginning.

```
def move(self,move_left,move_right):

    # At the end of the method, the values will be added to the player.rect. So it is set to 0.
    x_change = 0
    y_change = 0

    # for some reason the characters  move faster when jumping than running, so I added this extra code
    if self.jump_status or self.fall_status:
        player_speed = 5
    elif self.jump_status == False or self.fall_status == False:
        player_speed = 8
    else:
        player_speed = 5

    # Edit the x_change if the player moves left / right
    if move_left:
        x_change = -player_speed
        self.flip = True
    elif move_right:
        x_change = player_speed
        self.flip = False
    else:
        player_speed = 0
```

The move method is used to move the player and update the action when moving. The x_change and y_change are always reset to 0, because at the end of the method they're added to the character's coordinates. There's a bug that the characters move faster when jumping than running so I added this extra code, where player_speed represents the change in x coordinates.

The move_left and move_right values are taken from the main loop, where it checks for the keys pressed by the user. Here, it checks if those values are true, if they are, the x coordinates and self.flip are updated. Self.flip when true makes the character face to the right and left otherwise. If none are true, the speed is 0.

```
# Playing the animation for running
if move_left or move_right:
    self.update_action(1)
else:
    self.update_action(0)


if self.jump and self.in_air == False:

    # A negative number so there will be a transition when the character is jumping
    self.vel_y = -11
    self.jump = False

    # To prevent the player from jumping twice
    self.in_air = True

# Make the character look like they have a transition when jumping, and causes them to fall when th
self.vel_y += self.gravity

# Updates the y coordinates
y_change += self.vel_y
```

Here, the update_action is used. When the character is moving, it self.update_action(1), because in the previous codes, the running images are appended in second place (after the idle images). It will self.update_action(0) otherwise. The update for jumping and falling will be later.

Next, we have an if statement that ensures the character cannot double jump. The self.vel_y is set to a negative number and added with self.gravity to make it look slowly go up and go back down once the self.vel_y is positive. The y_change is added with the self.vel_y value.

```
# There is a ground in the stage 1 from the wallpaper, I made this code so that the player can stan
wallpaper_ground_x = 475
if self.rect.bottom + y_change >= wallpaper_ground_x:
    y_change = 0
    self.fall_status = False
    self.in_air = False

# Helps in updating the action
if self.in_air and self.vel_y > 0:
    self.fall_status = True
    self.jump_status = False

elif self.vel_y + self.gravity < 0:
    self.fall_status = False
    self.jump_status = True

# Updates jumping and falling action
if self.fall_status:
    self.update_action(3)
if self.jump_status:
    self.update_action(2)
```

The first section of code prevents the users from falling through the wallpaper ground. (There is a ground in the wallpaper that the users can stand on in stage 1) The if section after that checks if the player is in air and moving downwards the screen. If it is, it makes the jump value false and the fall value true. The next section does the opposite of this.

The last section updates the action of the character for jumping and falling.

```
# So we can access the world even though in different classes
global world

# Prevent the player from moving through the tiles
for tile in world.block_list:
    if tile[1].colliderect(self.rect.x + x_change, self.rect.y, self.width, self.height):
        x_change = 0
    if tile[1].colliderect(self.rect.x, self.rect.y + y_change, self.width, self.height):

        if self.vel_y > 0:
            self.on_top_of_tile = True
            y_change = tile[1].top - self.rect.bottom
            self.vel_y = 0
            self.in_air = False
            self.fall_status = False
            player_speed = 4

        if self.vel_y < 0:
            y_change = tile[1].bottom - self.rect.top
            self.vel_y = 0
            player_speed = 4
```

```
# Same as above but for lavas, but it will cause the player to die when colliding
for tile in world.danger_list:

    if tile[1].colliderect(self.rect.x + x_change, self.rect.y, self.width, self.height):
        x_change = 0
        self.alive = False

    if tile[1].colliderect(self.rect.x, self.rect.y + y_change , self.width, self.height):

        if self.vel_y >= 0:

            y_change = tile[1].top - self.rect.bottom
            self.vel_y = 0
            self.in_air = False
            self.fall_status = False
            player_speed = 4
            self.alive = False

        if self.vel_y < 0:
            y_change = tile[1].bottom - self.rect.top
            self.vel_y = 0
            player_speed = 4
            self.alive = False
```

The world variable is defined later on in each level method. I made it global so it can be accessed. The next section of the code produces effects when the character collides with blocks (tiles). The lava and door are stored in separate lists.

This section of code prevents the users from passing through the tiles vertically and horizontally by changing the values of y_change and x_change. (The self.on_top_of_tile is a mistake and not used) . If the vel_y is larger than 0, the character is falling, the character is jumping if the number is less than 0.

The second picture is pretty much the same but it is for lava blocks, where colliding with it will kill the character.

```
        if self.alive == False:
            y_change = 0
            x_change = 0

        # Finally updating the coordinates
        self.rect.x += x_change
        self.rect.y += y_change

    # This method displays the player
    def draw(self):
        screen.blit(pygame.transform.flip(self.image,self.flip,False),self.rect)
```

Finally, if the character is dead, the character won't move at all. The coordinates of the character are also updated. The final method of the class draw() will display the character with the updated coordinates along with the self.flip value. The value after it is false because we don't want to flip it vertically.

```
# The enemy class displays the enemy character and moves it.
class Enemy:

    # Initializing
    def __init__(self, scale, min_x, max_x, x, y, image):
        self.image = image
        enemy = pygame.image.load(self.image).convert_alpha()
        self.scale = scale
        self.image = pygame.transform.scale(enemy,(int(enemy.get_width()*self.scale),int(enemy.get_height()
        self.rect = self.image.get_rect(topleft = (x, y))
        self.min_x = min_x
        self.max_x = max_x
        self.speed = 0.5
        self.flip = False
        self.height = self.image.get_height()
```

Moving on to the next class, we have the Enemy class. Where it takes in scale, min_x, max_x, x, y and image. The min_x and max_x are borders that restrict the movement of the enemies. The x and y represent the initial placement.

```
    def move(self):

        # Once the enemy reaches the max point (x coordinate) set, the enemy will move the oppsite way
        if self.rect.x >= self.max_x:
            self.speed = -0.6
            self.flip = True

        if self.rect.x <= self.min_x:
            self.speed = 0.6
            self.flip = False

        # Updating coordinates
        self.rect.x += self.speed
```

The move method moves the enemy, and changes values once the enemy has reached the borders. So it is able to move from left to right. The coordinates are then updated.

```
    # Displaying the enemy
    def draw(self):
        screen.blit(pygame.transform.flip(self.image, self.flip, False),self.rect)
```

The draw method displays the enemy.

```
class World:

    # Initializing
    def __init__(self, data):

        # stores the image and the rect(coordinates) according to the number in the list
        # I want to give different effects when the user collides with different blocks
        self.block_list = []
        self.danger_list = []
        self.door_list = []

        row_count = 0

        block_img = pygame.image.load("tile.png").convert_alpha()
        lava_img = pygame.image.load("lava.png").convert_alpha()
        door_img = pygame.image.load("door.png").convert_alpha()
```

We have the world class next. It takes in data which is the world structure for each level. The self.block_list, danger_list and door_list that were used previously store the images. The block_list storing the tiles, danger_list storing the lava, and the door_list storing the doors. They are stored in different lists, because they have different effects when colliding with them.

The data that is taken in is in the form of a list containing many lists. So we set the row count to 0, and load other necessary images.

```
        # goes through each list
        for row in data:
            col_count = 0
            # goes through each element in each list
            for tile in row:
                # if statements according to the numbers
                if tile == 1:
                    imagee = pygame.transform.scale(block_img, (tile_width, tile_height))
                    imagee_rect = imagee.get_rect()

                    # So there will be constant gap length between tiles
                    imagee_rect.x = col_count * tile_width
                    imagee_rect.y = row_count * tile_height
                    tile = (imagee, imagee_rect)
                    self.block_list.append(tile)
                if tile == 2:
                    imagee = pygame.transform.scale(lava_img,(tile_width,tile_height))
                    imagee_rect = imagee.get_rect()
                    imagee_rect.x = col_count * tile_width
                    imagee_rect.y = row_count * tile_height
                    tile = (imagee, imagee_rect)
                    self.danger_list.append(tile)
                if tile == 3:
                    imagee = pygame.transform.scale(door_img, (tile_width, tile_height))
                    imagee_rect = imagee.get_rect()
                    imagee_rect.x = col_count * tile_width
                    imagee_rect.y = row_count * tile_height
                    tile = (imagee, imagee_rect)
```

```
                # So the gap is continuously updated
                col_count += 1
            row_count += 1
```

Here the elements that are in the list are in the form of numbers. Where each number represents a block. Here, we append the scaled images with its coordinates to each respective list. The row_count and col_count are

constantly updated no matter what the number is, so we can use the row_count and col_count to help us find the coordinates for the block.

```
    # Displaying the world structure (The blocks, lava, door.)
    def draw(self):

        # prints each tuple containing the image and the coordinates
        for tile in self.block_list:
            screen.blit(tile[0], tile[1])
        for tile in self.danger_list:
            screen.blit(tile[0],tile[1])
        for tile in self.door_list:
            screen.blit(tile[0], tile[1])
```

Using the draw method, we can draw the world. (tile[0] is the image and tile[1] is the coordinates)

```
# This class displays coins and responses if the player collides with the coinse
class Coins:

    # Initializing
    def __init__(self, x, y):
        coin_image = pygame.image.load("coin.png").convert_alpha()
        self.coin_image_scaled = pygame.transform.scale(coin_image,(int(coin_image.get_width()*0.08),int(
        self.rect = self.coin_image_scaled.get_rect(topleft = (x, y))

    def draw(self):

        screen.blit(self.coin_image_scaled,(self.rect.x, self.rect.y))

        # So the score value can be accessed from other methods
        global score

        if player_rectangle.colliderect(self.rect):

            # Move the coin out of screen when colliding and adding the score
            self.rect.x += 1000
            score += 10
```

For the last class, we have the coin class. This class takes in the x and y coordinates of the coins. It loads the coin images and scales it, and also gets the rect to check for collisions.

The draw method displays the coins, we have a global score variable so it can be accessed from other methods (So we can continuously update the score value). The player rectangle is defined later, but it is just the rectangle of the character. So if the player collides with a coin, the coin is moved out of frame

to prevent further increase of the score from colliding with a coin, and the score increases.

```
# This function is displayed when the program is executed. It displays the main menu.
def main_menu():

    yellow = (255,255,0)

    run = True

    # Inserting necessary images, caption and logo
    pygame.display.set_caption("THE CRUSADERS")
    logo = pygame.image.load("logo.png").convert_alpha()
    pygame.display.set_icon(logo)

    background = pygame.image.load("bg.png").convert_alpha()
    background_scaled = pygame.transform.scale(background,(850,500))

    play = pygame.image.load("play.png").convert_alpha()
    play_scaled = pygame.transform.scale(play,(150,70))

    game_logo = pygame.image.load("gamelogo.png").convert_alpha()
    game_logo_scaled = pygame.transform.scale(game_logo,(60,90))

    font = pygame.font.Font("Minecraftia-Regular.ttf",font_size)
    display_text1 = font.render("THE", True, yellow)
    display_text2 = font.render("CRUSADERS", True, yellow)
```

```
    while run:

        screen.fill((0,0,0))

        # Displaying Images
        background_image = Image(background_scaled,0,0,1)
        background_image.display()

        play_image = Image(play_scaled,195,320,1)
        play_image.display()

        exit_image = Image(exit_scaled,485,320,1)
        exit_image.display()

        game_logo_image = Image(game_logo_scaled, 385,220,1)
        game_logo_image.display()

        screen.blit(display_text1,(345,40))
        screen.blit(display_text2,(221,120))

        mouse_position = pygame.mouse.get_pos()
```

```
    # Provide responses when the mouse is clicked on the buttons.
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == left_mouse_clicks:
                if exit_image.rect.collidepoint(mouse_position):
                    comfirm_quit()
                if play_image.rect.collidepoint(mouse_position):
                    choose_character()

    pygame.display.update()
    clock.tick(FPS)
pygame.quit()
sys.exit()
```

We have the main_menu function that is displayed when the program is initially executed. It sets the caption, logo, background wallpaper and buttons that are already scaled. The pygame.mouse.get_pos() gets the position of the mouse which is updated every iteration. It checks for mouse clicks by using the collidepoint, the event.button represents the mouse buttons where 1 represents left clicks of the mouse. The pygame.display.update() updates the changes done.

```
# This function is displayed when the user clicks the exit button in the main menu. It re-confirms with t
def comfirm_quit():

    run = True

    yellow = (255,255,0)

    # Loading in images and font styles
    background = pygame.image.load("bg.png").convert_alpha()
    background_scaled = pygame.transform.scale(background,(850,500))
    font = pygame.font.Font("Minecraftia-Regular.ttf",font_size)
    confirm_text = font.render("CONFIRM EXIT", True, yellow)

    while run:

        screen.fill((0,0,0))

        # Displaying objects
        background_image = Image(background_scaled,0,0,1)
        background_image.display()

        exit_image = Image(exit_scaled,485,320,1)
        exit_image.display()

        cancel_image = pygame.image.load("cancel.png").convert_alpha()
        cancel_image_scaled = pygame.transform.scale(cancel_image,(150,70))
```

```
        cancel_button = Image(cancel_image_scaled, 195,320, 1)
        cancel_button.display()

        screen.blit(confirm_text,(180,170))

        mouse_position = pygame.mouse.get_pos()

        # Providing responses when the mouse button is clicked
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == left_mouse_clicks:
                    if exit_image.rect.collidepoint(mouse_position):
                        run = False
                    if cancel_button.rect.collidepoint(mouse_position):
                        main_menu()

        pygame.display.update()
        clock.tick(60)
    pygame.quit()
    sys.exit()
```

This is the confirm_quit function. When the user clicks the exit button in the main menu, it redirects the user here. The concept in the code is the same, displaying scaled images and checking for left mouse clicks.

```
# This function is where the user chooses their character
def choose_character():

    yellow = (255,255,0)

    run = True

    # Loading in images, font styles and captions
    pygame.display.set_caption("CHOOSE YOUR CHARACTER")
    font = pygame.font.Font("Minecraftia-Regular.ttf", font_size)
    display_text3 = font.render("CHOOSE CHARACTER", True, yellow)

    background = pygame.image.load("bg.png").convert_alpha()
    background_scaled = pygame.transform.scale(background, (850, 500))

    character_1_image = pygame.image.load("character1/idle/0.png").convert_alpha()
    character_1 = Image(character_1_image, 225, 150, 5)

    character_2_image = pygame.image.load("character2/idle/0.png").convert_alpha()
    character_2 = Image(character_2_image, 520, 160, 5.5)

    select_button = pygame.image.load("select.png").convert_alpha()
    select_button_scaled = pygame.transform.scale(select_button, (int(select_button.get_width() * 0.

    select_button1_x, select_button1_y = 210,390
    select_button2_x, select_button2_y = 525,390

    select_button1 = Image(select_button_scaled, select_button1_x, select_button1_y, 1)
    select_button2 = Image(select_button_scaled, select_button2_x, select_button2_y, 1)
```

The user is redirected here when clicking the play button in the main menu. It utilizes the Image class to display the buttons, so it will be simpler and more efficient.

```
while run:

    # Displaying background, characters and buttons
    screen.blit(background_scaled, (0, 0))
    screen.blit(display_text3, (80, 50))

    character_1.display()
    character_2.display()

    select_button1.display()
    select_button2.display()

    mouse_position = pygame.mouse.get_pos()

    # Responds according to the button clicked
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == left_mouse_clicks:
                if select_button1.rect.collidepoint(mouse_position):
                    level_1("character1")
                if select_button2.rect.collidepoint(mouse_position):
                    level_1("character2")

    pygame.display.update()
```

```
    pygame.quit()
    sys.exit()
```

It displays the images and detects left mouse clicks on the select button.

```
# Function that displays the first level
def level_1(character):

    pygame.display.set_caption("MAIN GAME")
    font = pygame.font.Font("Minecraftia-Regular.ttf", 20)

    # Loading in the world structure that will run in the World class
    world_structure = [
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1],
    [1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1],
    [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
    [1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
    [1,0,0,0,0,0,0,0,0,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
    [1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,0,0,0,0,0,0,0,0,0,1,1,1],
    [1,0,0,0,0,0,1,1,1,1,2,2,1,1,0,0,0,0,1,1,0,0,0,0,0,0,1,1],
    [1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,0,0,0,0,0,1,1],
    [1,3,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1],
    [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1],
    [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
    [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
    [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],
    [1,0,0,0,0,0,0,0,0,1,2,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,0,0,0,1,1,1,1,1,1,2,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    ]
```

The user is redirected here once a character is selected. As stated before the world data is in the form of numbers where 0 represents nothing, 1 represents blocks, 2 represents lava and 3 represents doors.

```
global world

# World_structure is went through
world = World(world_structure)

# Both characters have different sizes, so this is necessary
player_x = 50
if character == "character1":
    scale = 2
    player_y = 403
if character == "character2":
    scale = 2.2
    player_y = 407

# Setting up initial coordinates and sizes
player = Player(character, scale, player_x, player_y)

enemy_scale = 0.15
enemy1_minimum_x, enemy1_maximum_x, enemy1_x, enemy1_y = 200, 210, 200, 195
enemy2_minimum_x, enemy2_maximum_x, enemy2_x, enemy2_y = 480, 530, 480, 405

enemy1 = Enemy(enemy_scale, enemy1_minimum_x, enemy1_maximum_x, enemy1_x, enemy1_y, "guard.png")
enemy2 = Enemy(enemy_scale, enemy2_minimum_x, enemy2_maximum_x, enemy2_x, enemy2_y,"guard.png")

coin1, coin2, coin3, coin4, coin5, coin6, coin7 = Coins(314,320), Coins(500,320), Coins(779,323), Coins

moving_left = False
moving_right = False
```

Defining the enemies and the coins (such as coordinates, minimum x, and others) . It also utilizes the world function by giving it the world data, which is then later drawn. Both characters have different sizes so the scaling is different.

```
background = pygame.transform.scale(pygame.image.load("game_wallpaper.jpg").convert_alpha(),(850,500))
caution_img = pygame.image.load("caution.png").convert_alpha()
caution_img_scaled = pygame.transform.scale(caution_img,(int(caution_img.get_width()*0.07),int(caution_

run = True

# so it can be accessed from the coins class to add the score when collisions occur
global score
score = 0

# The stage number (level)
stage = 1
stage_text = font.render(f"Stage: {stage}", True, (255,255,0))

while run:

    # To remove trails
    screen.fill((0, 0, 0))

    # Displaying images
    screen.blit(background,(0,0))
    screen.blit(caution_img_scaled,(160,416))

    # Update player action
    player.update_action(player.action_index)
```

Loading in necessary images. Set the score to be 0 and the stage to be 1. Displaying the stage, the display of the score is inside the loop, as the score is constantly updated in stage 1 while the stage number isn't. The screen.fill is used to remove trails from objects moving. Displays images and updates the action of the player.

```
# Draw the world_structure
world.draw()

# Displaying texts
score_text = font.render(f"Score: {score}", True, (255,255,0))
screen.blit(score_text,(10,10))
screen.blit(stage_text, (750,10))

# Update player animation
player.update_animation()
```

Draws the structure of the world (according to the world data previously). Displaying the score text and updating the player animation.

```
# Moves the player according to the key pressed
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        run = False
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_a:
            moving_left = True
        if event.key == pygame.K_d:
            moving_right = True
        if event.key == pygame.K_w and player.in_air == False:
            player.jump = True
            jump_sound.play()
    if event.type == pygame.KEYUP:
        if event.key == pygame.K_a:
            moving_left = False
        if event.key == pygame.K_d:
            moving_right = False
        if event.key == pygame.K_w:
            player.jump = False

# Moving and drawing the player and enemies
player.move(moving_left, moving_right), player.draw()
enemy1.move(), enemy1.draw()
enemy2.move(), enemy2.draw()
```

Responds to the keys pressed by the user with some conditions. Like the player must not be in the air in order to jump. The values are updated when the keys are released. There is also a sound effect played when the character jumps.

The player and enemies are moved and are then displayed.

```
    # Make a rectangle to check collisions more accurately
    global player_rectangle
    player_rectangle_width, enemy_rectangle_width = 50,23

    # Setting up rectangles (If not, player may collide with enemies even though it does not)
    player_rectangle = pygame.Rect(player.rect.x , player.rect.y, player_rectangle_width , player.heig
    enemy1_rectangle = pygame.Rect(enemy1.rect.x, enemy1.rect.y, enemy_rectangle_width, enemy1.height)
    enemy2_rectangle = pygame.Rect(enemy2.rect.x , enemy2.rect.y, enemy_rectangle_width , enemy2.heigh

    # Display coins
    coin1.draw(), coin2.draw(), coin3.draw(), coin4.draw(), coin5.draw(), coin6.draw(), coin7.draw()

    # Kill player if colliding with enemies
    if player_rectangle.colliderect(enemy2_rectangle) or player_rectangle.colliderect(enemy1_rectangle
        player.alive = False

    # To give a short delay after dying
    time_delay_after_death = 300

    # Redirecting to another function after dying
    if player.alive == False:
        pygame.time.delay(time_delay_after_death)
        you_died(player.character)
```

The player_rectangle is the rectangle shape of the player, but it is more accurate to check for collisions as the width is smaller than the previous one. I used it on the enemies too so the collisions will be more accurate. The coins are drawn, and killing the character when colliding with the enemies. I put a delay after the player dying so it won't be instant by using the pygame.time.delay(), before moving to the you_died function.

```
        # if the player reaches the door, it redirects to the next level
        for tile in world.door_list:
            if player_rectangle.colliderect(tile[1]):
                level_2(character)

        clock.tick(FPS)
        pygame.display.update()

    pygame.quit()
    sys.exit()
```

If the player collides with the door, the user will be redirected to the next stage, indicating that the user has successfully cleared stage 1. The stage 2 function will also receive the character value, so the next stage will also have the same character.

```
  # Function that displays the second level
> def level_2(character):···

  # Function that displays the third level
> def level_3(character):···

  # Function thst displays the final level
> def level_4(character):···
```

The other levels function similarly to level 1. They just vary in the world structure, number and placement of enemies, placement and number of coins.

```
# Function when the player dies
def you_died(character):

    run = True

    # Setting images and texts
    background = pygame.image.load("bg.png").convert_alpha()
    background_scaled = pygame.transform.scale(background,(850,500))
    font = pygame.font.Font("Minecraftia-Regular.ttf",font_size)
    confirm_text = font.render("YOU DIED", True, (255,255,0))

    # Will be used to make sure the audio is played once
    death_sound_played = False

    while run:

        screen.fill((0,0,0))

        # Displaying images and texts
        background_image = Image(background_scaled,0,0,1)
        background_image.display()

        exit_image = Image(exit_scaled,485,320,1)
        exit_image.display()

        retry_image = pygame.image.load("retry.png").convert_alpha()
        retry_image_scaled = pygame.transform.scale(retry_image,(150,70))
```

```
        screen.blit(confirm_text,(250,170))

        mouse_position = pygame.mouse.get_pos()

        # Provides a choice to the player to exit or retry
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                run = False
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:
                    if exit_image.rect.collidepoint(mouse_position):

                        # Exit
                        run = False
                    if retry_button.rect.collidepoint(mouse_position):

                        # Retry
                        level_1(character)

        # Makes sure the audio is played once
        if death_sound_played == False:
            death_sound.play()
            death_sound_played = True

        pygame.display.update()
        clock.tick(60)
    pygame.quit()
    sys.exit()
```

The you_died method is displayed when the character dies. Most of the code is just loading images and displaying them. There is a sound effect played once the function is displayed. The death_sound_played is set to false, and after the death sound is played, the death_sound_played is set to True to make sure that the sound is played only once. If the retry button is clicked, it redirects the user to level_1 function and exits the program if the exit button is clicked.

```
# Function if the player wins, it either goes back to the main menu or exit
def game_finished():

    yellow = (255,255,0)

    run = True

    # Loading in images and texts
    pygame.display.set_caption("YOU WON!")
    font = pygame.font.Font("Minecraftia-Regular.ttf", font_size)
    display_text3 = font.render("YOU WON!", True, yellow)

    background = pygame.image.load("bg.png").convert_alpha()
    background_scaled = pygame.transform.scale(background, (850, 500))

    back_button_x, back_button_y = 210,340
    exit_button_x, exit_button_y = 525,340

    back_button = pygame.image.load("back.png").convert_alpha()
    back_button_scaled = pygame.transform.scale(back_button,(160,80))
    back_button_rect = back_button_scaled.get_rect(topleft = (back_button_x, back_button_y))

    exit_button = pygame.image.load("exit.png").convert_alpha()
    exit_button_scaled = pygame.transform.scale(exit_button,(160,80))
    exit_button_rect = exit_button_scaled.get_rect(topleft = (exit_button_x, exit_button_y))

    select_button1 = Image(back_button_scaled, back_button_x, back_button_y, 1)
    select_button2 = Image(exit_button_scaled, exit_button_x, exit_button_y, 1)
```

```
  score_text = font.render(f"Score: {score}", True, (255,255,0))

  while run:

      # Displaying images
      screen.blit(background_scaled, (0, 0))
      screen.blit(display_text3, (280, 80))
      screen.blit(score_text, (255, 190))

      select_button1.display()
      select_button2.display()

      mouse_position = pygame.mouse.get_pos()

      for event in pygame.event.get():
          if event.type == pygame.QUIT:
              run = False
          if event.type == pygame.MOUSEBUTTONDOWN:
              if event.button == left_mouse_clicks:
                  if exit_button_rect.collidepoint(mouse_position):

                      # Exits
                      run = False
```

```
                    run = False
                if back_button_rect.collidepoint(mouse_position):

                    # Back to main menu
                    main_menu()

        pygame.display.update()

    pygame.quit()
    sys.exit()

# Executing the code
if __name__ == "__main__":
    main_menu()
```

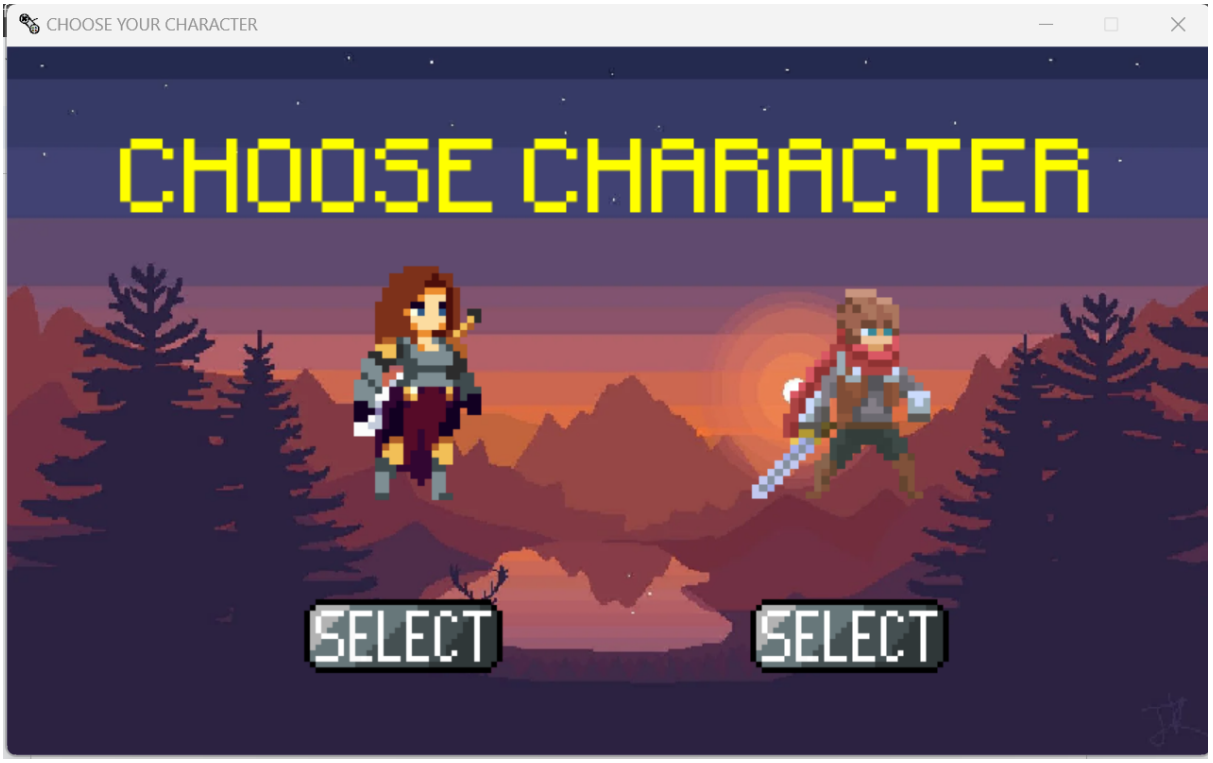
Most of the code is loading the images, scaling and displaying them.

The game finished function is displayed once the user has completed all 4 stages. It gives the user a choice to exit or go back to the main menu.
At the bottom of the code the main menu function is executed, the if function is used to check if the python script is run directly.

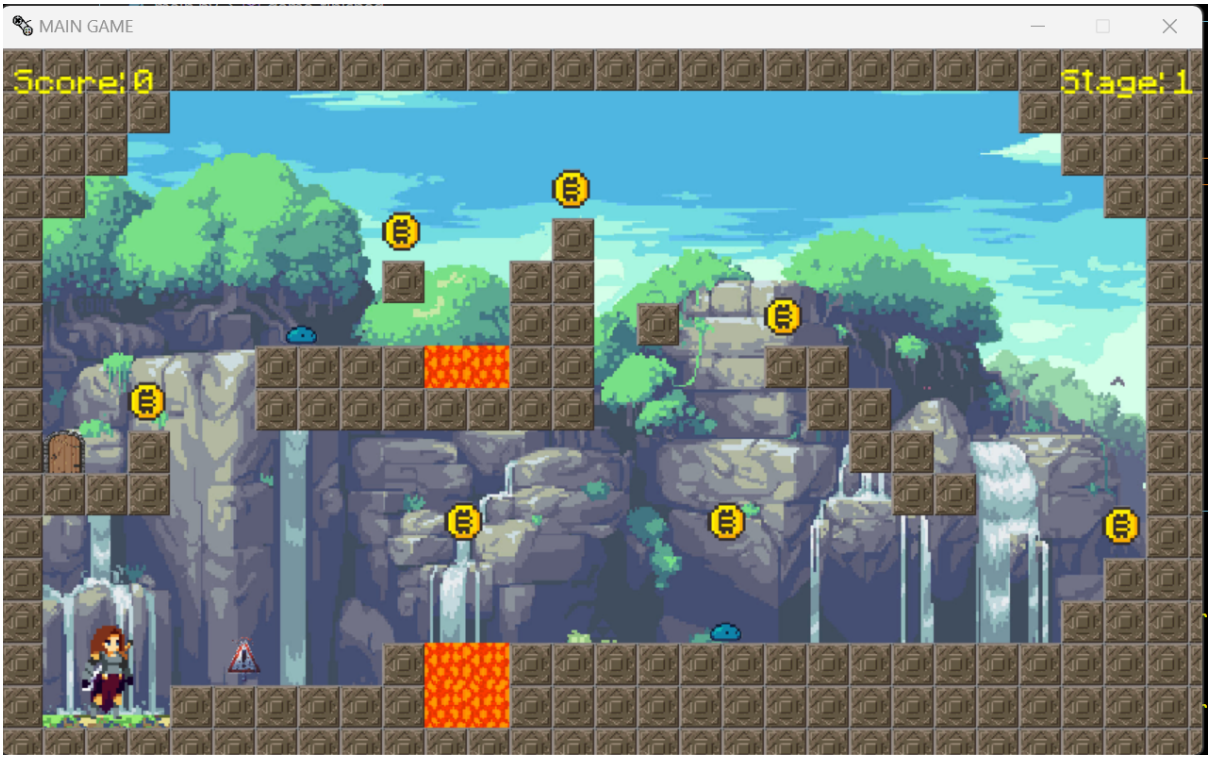
5. Screenshots of the Game

Main Menu

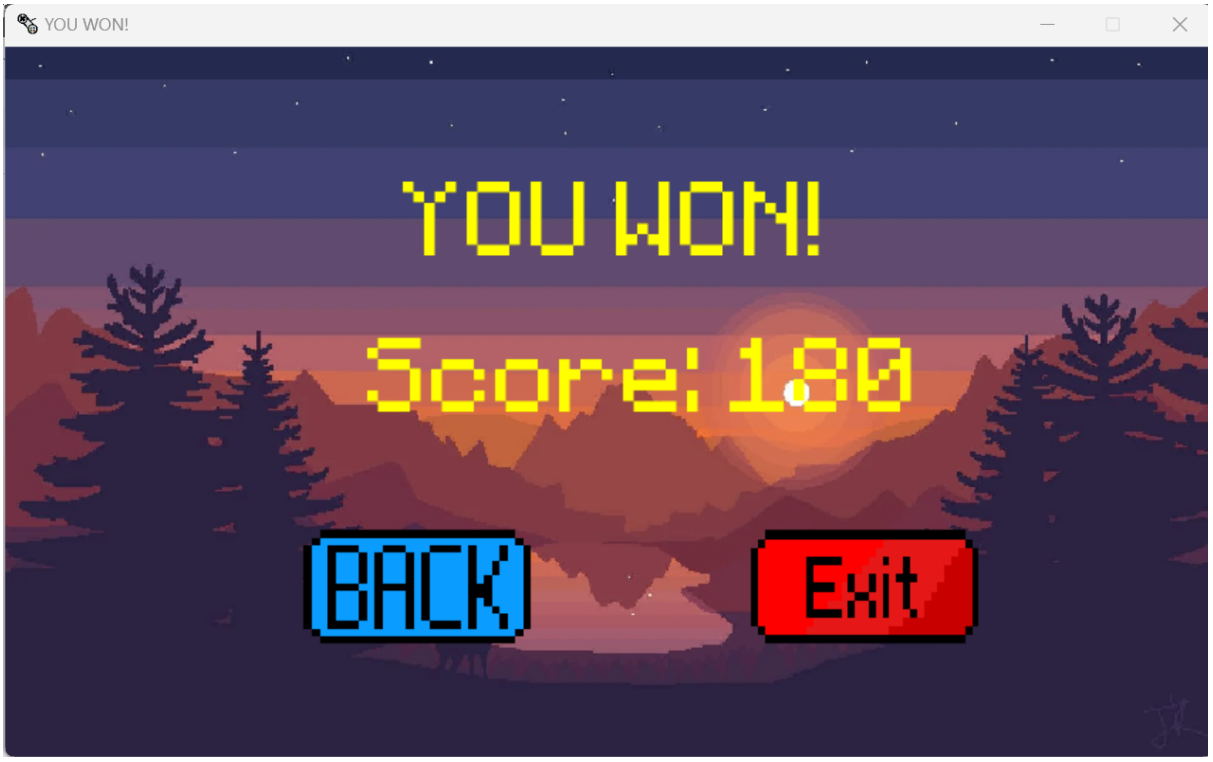
Choosing the character



First Stage



Winning the game



6. Reflection

I have learned a lot of new concepts, and feel a lot more comfortable working with pygame as I understand the simple concepts. At first, I was struggling to understand the concepts. But with the help of youtube videos, I was able to understand them clearly.

Maybe one of the things I need to improve is to avoid the repetition of similar codes. Like using similar codes in the Level_1 through 4 function.

At the end of the day, I have learned about new concepts and understand other concepts better. So I see this project as very benefitting.