BINUS UNIVERSITY

BINUS INTERNATIONAL

Red Alert

Algorithm & Programming

Final Project Report

Student Information:

Surname: Agastya **Given Name:** Randy **Student ID:** 2602119165

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

Class : L1AC **Lecturer :** Mr. Jude Joseph Lamug Martinez

Type of Assignment: Final Project Report

Submission Pattern

Due Date : 15/01/23 **Submission Date :** 15/01/23

Table of Contents

Project Specifications

Background

For my final project, I was instructed to create a python project which extends beyond my comfort level and beyond what was taught in class. The goal of the Term Project is to apply all that you have learned about the Python Programming language and problem solving to solve a relatively small but interesting problem. So, I have decided to create a platformer game in which I try my absolute best to use skills that weren't taught in this semester's syllabus while maintaining the focus of the topic we have covered in class. This report will be my report in which I will explain about my project.

Brief Description

I created Red Alert, an endless platformer game where the player can control the character to navigate through the levels, avoiding obstacles such as a moving saw blade, stationary spikes and a water area. The player would need to try to avoid these obstacles and if they interact or touch these obstacles; it would be game over. The levels are made up of blocks that the player can stand on and interact with. It is an endless-level type of game, so there wouldn't be an ending to this game, as the player's goal is to get the maximum distance possible.

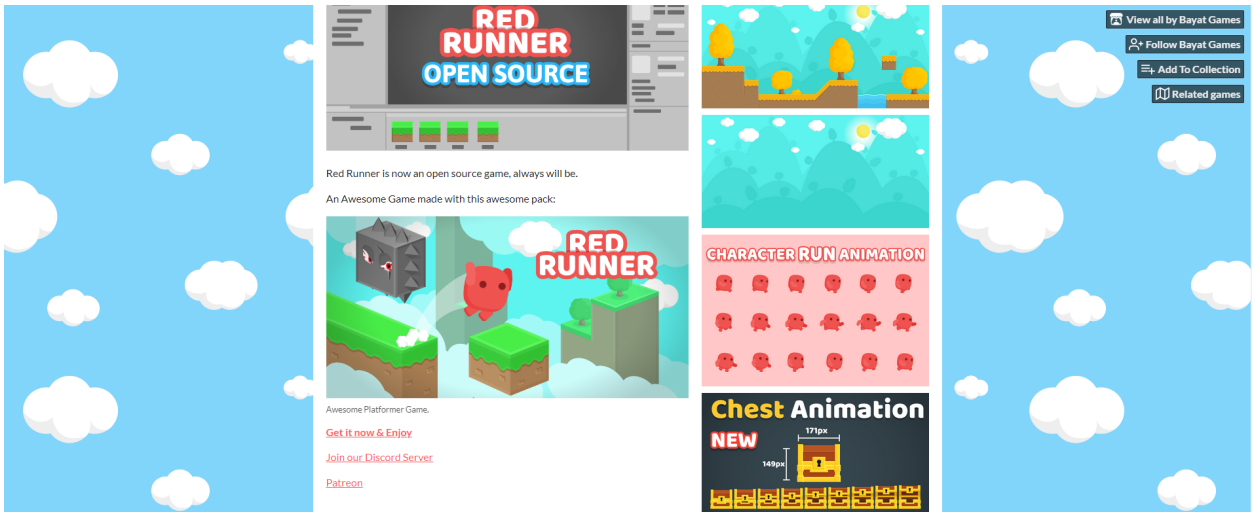
As a beginner developer, I wanted to create an endless platformer game that would keep the player coming back for more, and I believe that Red Alert achieves this goal. The objective of getting the furthest distance possible adds an element of replayability to the game, as players will want to beat their high scores. Additionally, the incorporation of coins and obstacles adds an extra layer of gameplay and strategy, as players must decide how to get through the obstacles. Overall, I created Red Alert to provide an enjoyable and challenging experience for players who love platformer games.

Solution Design

Objective of Design

During the plan, I was trying to use assets and resources that don't seem too extreme in which I would like the user to be comfortable and spoil the eyes with its bright design. The aim and requirement of the design that i was trying to achieve was for it to be family friendly as my objective is that all ages of people are able to play this game.

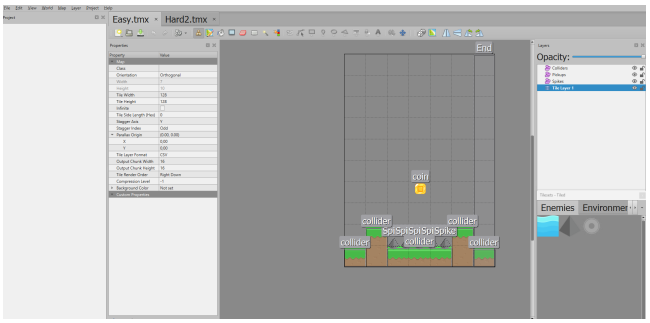
Design Choices



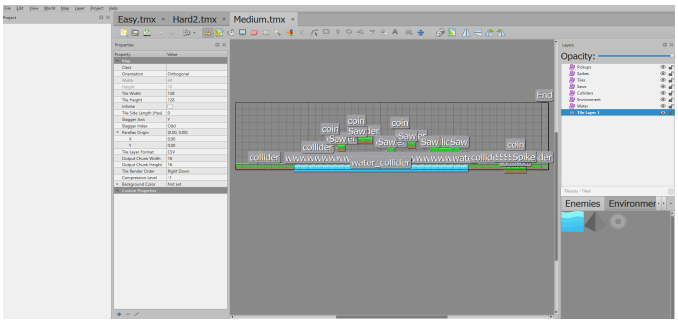
Link: https://bayat.itch.io/platform-game-assets

I decided to use this online & free asset that is easily accessible on the internet, which completes all of my objectives with its simple looks but elegant design. I feel it has a vibe where people would play any day and any time. Even Though these assets contain many resources such as tiles, characters, motion, obstacles and many more, I still need to create the map using software called TILED. I design the map from easy to hard, as you can see below:

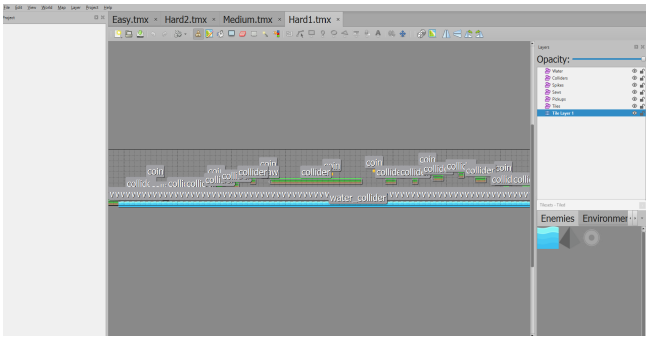
Easy



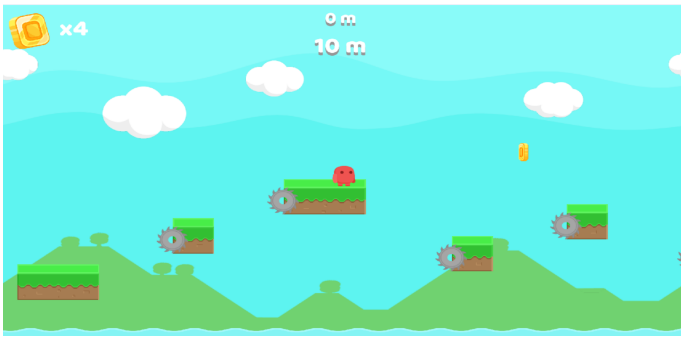
Medium



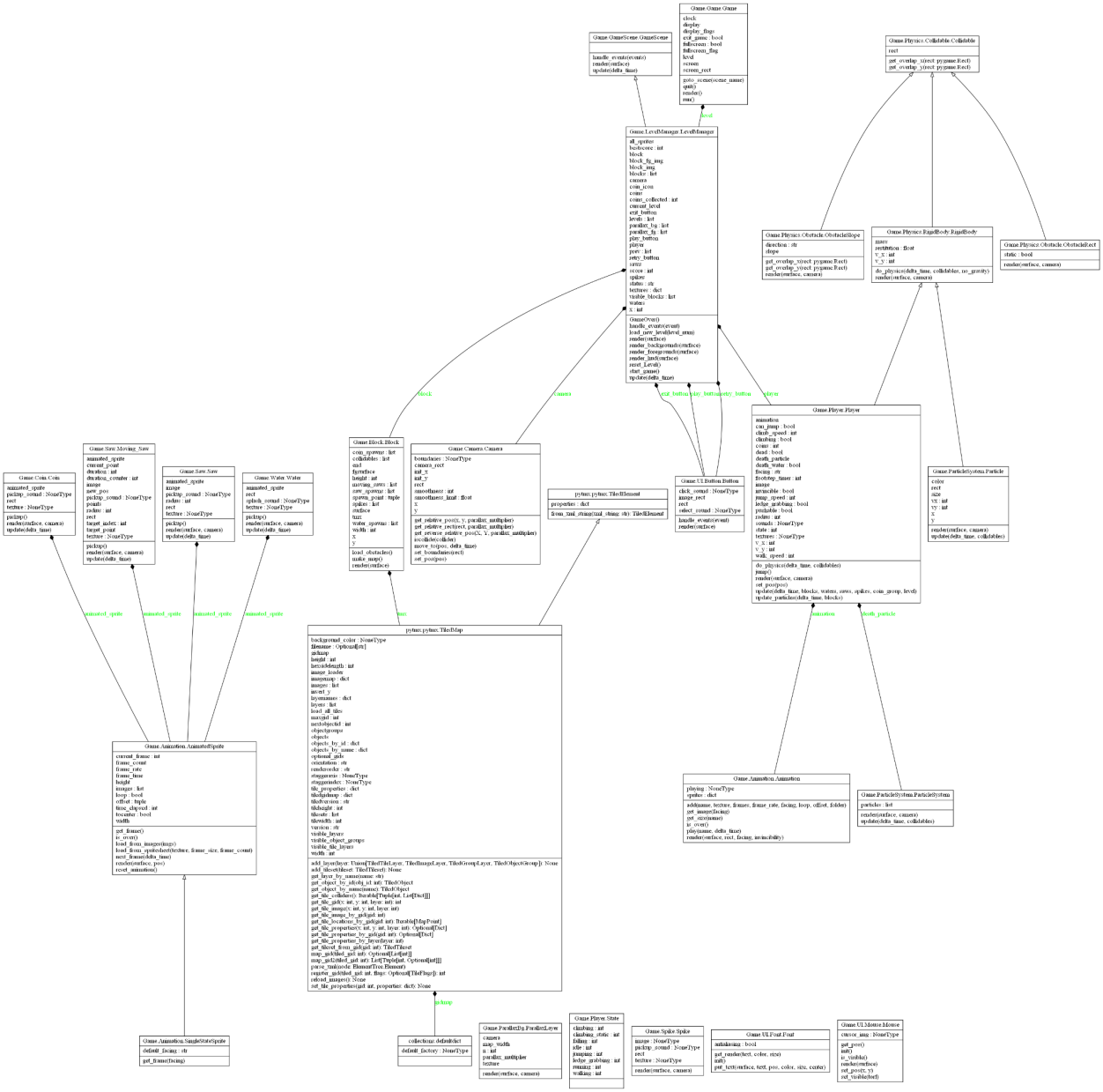
Hard



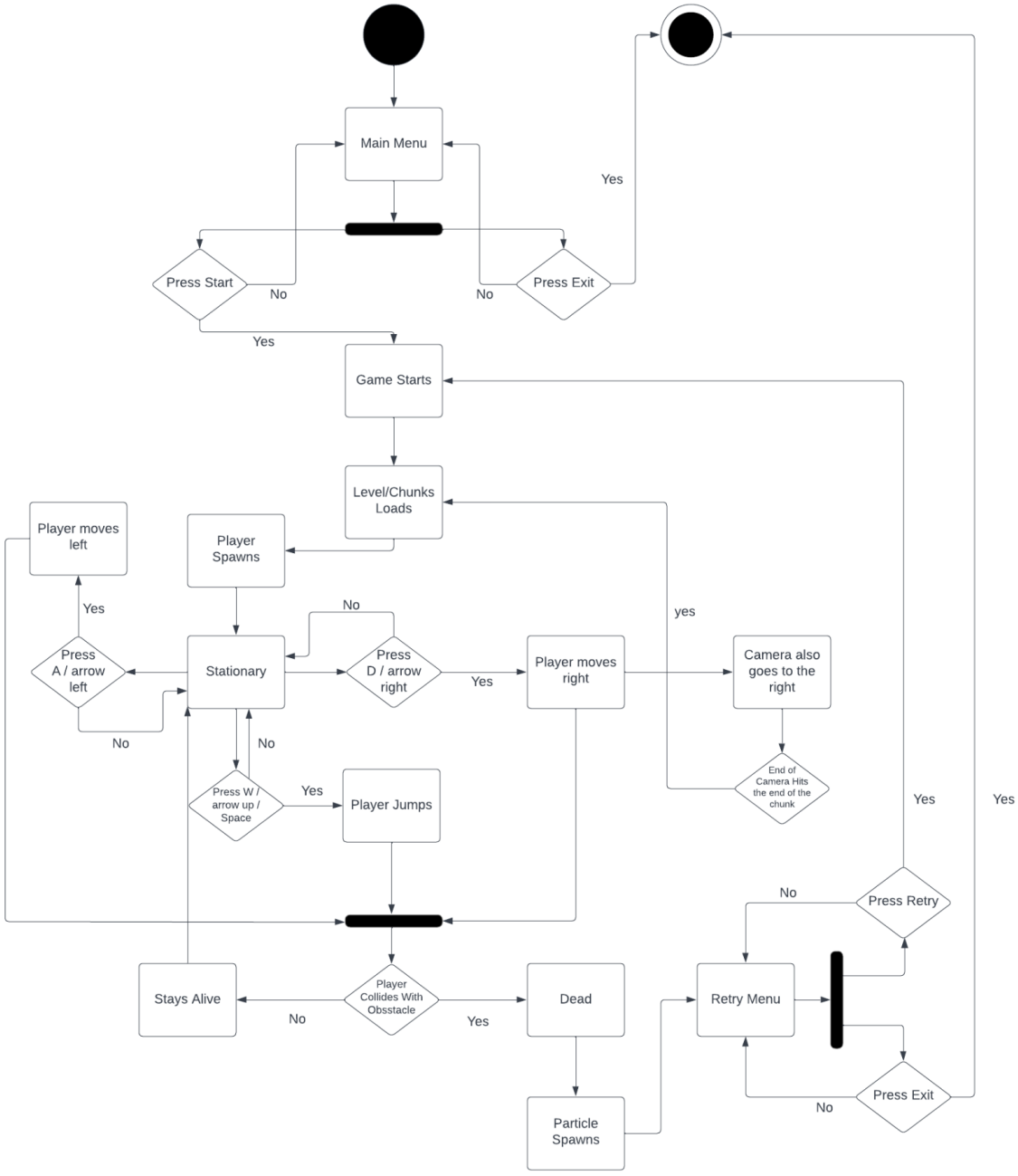
Result



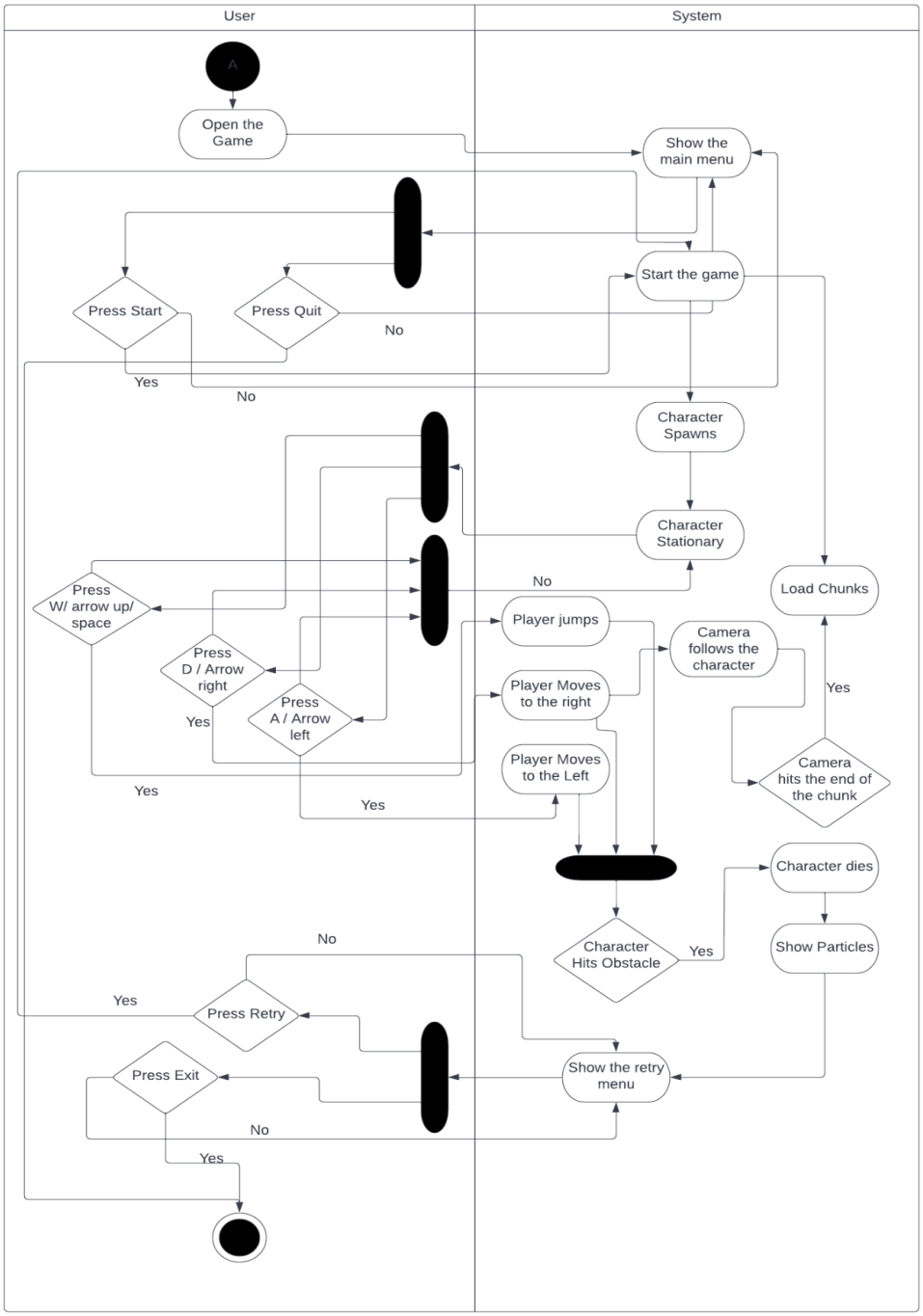
Class Diagram



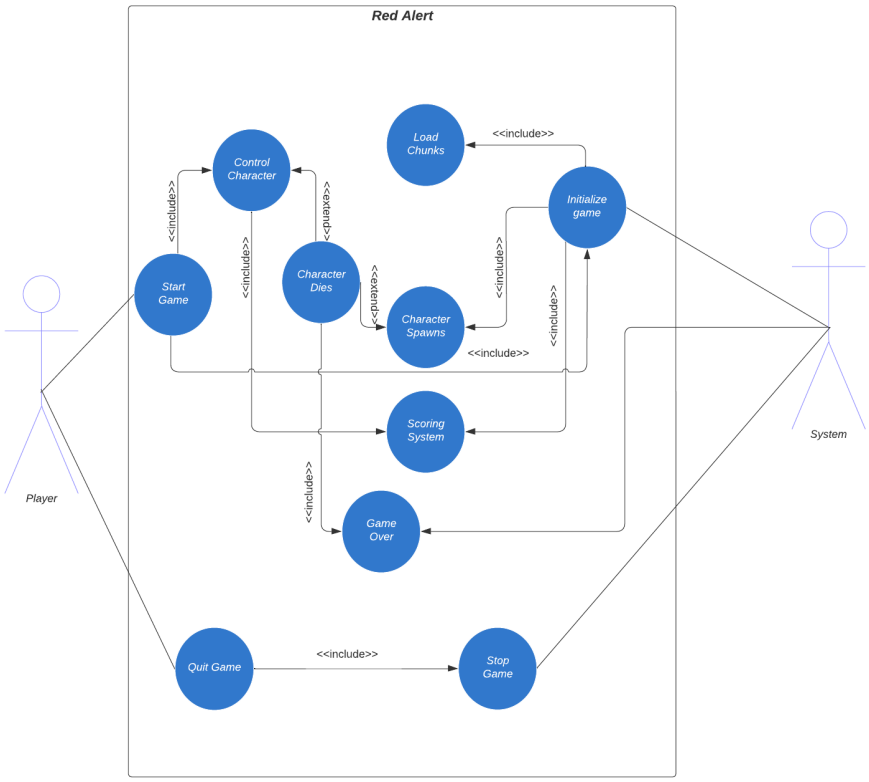
Flow Chart


Main Menu
Press Start
No
Press Exit
No
Yes
Yes
Game Starts
Level/Chunks Loads
Player moves left
Player Spawns
Yes
No
Press A / arrow left
Stationary
Press D / arrow right
Yes
Player moves right
yes
Camera also goes to the right
No
No
End of Camera Hits the end of the chunk
Press W / arrow up / Space
Yes
Player Jumps
Yes
Yes
No
Press Retry
Stays Alive
Player Collides With Obsstacle
No
Yes
Dead
Retry Menu
Particle Spawns
Press Exit
No

Activity Diagram



Use Case diagram



Libraries/Modules

- Pygame

  a set of Python modules designed for creating games. It includes functionality for handling graphics, audio, and input, making it a useful tool for creating games of all types.
- Pytmx

  A set of Python modules used to parse TMX map files, which are commonly used in games to create tile-based levels. TMX stands for Tiled Map XML and is a format created by the software Tiled, which is a popular open-source map editor.

- os

  A set of modules in Python which provides a way to interact with the operating system. It provides a variety of functions for interacting with the file system, starting new processes, killing processes, and more
- Glob

  A set of modules in Python which provide a way to search for files and directories using wildcard characters.
- Random

  A set of modules in Python which provides a suite of functions to generate random numbers, sequences, and selections. This module uses a popular and robust algorithm called Mersenne Twister, which can generate a large number of random numbers quickly and with a high degree of randomness.

Glob, os and Random are modules that are included in the Python standard library, which means that they are part of the Python programming language and are available to use without the need to download or install them.

However, pygame and pytmx are third-party modules, which means that they are not part of the Python standard library and must be installed separately.

To install these libraries:

```
C:\Users\randy>pip install pygame pytmx
```

Essential Algorithms

Player Motion

So the player motion works using the Player.py class which contains the player class. First, it works by loading the resources through the load_resources method which loads all the resources in the Assets folder for the player motion such as idle, run, jump and falling down and appends them to the texture dictionary.

```
Player.textures = {
    'player-idle': idle_texture,
    'player-run': run_texture,
    'player-jump': pygame.image.load(os.path.join("Assets/Player/Jump", 'Jump.png')).convert_alpha(),
    'player-fall': pygame.image.load(os.path.join("Assets/Player/Jump", 'Fall.png')).convert_alpha(),

}
```

After that, we use the dictionary to create the player animation which uses the state of the player's movement and we can use the animation class that can be found in animation.py like this:

```
self.animation.add(self.State.idle, Player.textures['player-idle'], 12, 10, 'right', offset=(-4, -0),folder=True)
self.animation.add(self.State.walking, Player.textures['player-run'], 18, 40, 'right', offset=(-10, -0),folder=True)
self.animation.add(self.State.jumping, Player.textures['player-jump'], 1, 1, 'right', offset=(-10, -0))
self.animation.add(self.State.falling, Player.textures['player-fall'], 1, 1, 'right', offset=(-10, -0))
```

So for example, for the state of idle, it has a frame rate of 12 frames per second, it loops indefinitely, it faces right and its image is offset by (-4,0) from the top-left corner of the sprite. The folder is set to true as idle is a list of image paths rather than image frames.

In animation.py, the class Singlestatesprites is used in which it helps render the sprite with a given number of frames at a given frame rate. It contains a method to load frames from either a sprite sheet or a folder of images and render the frame at a position. The Animation class has methods to add new states, play a given state, and update the current state. It also has a method to check if the current state is over, which can be used to determine when to switch to a different state.

The purpose of the render method is to draw the current frame of the animation on the given surface at the position specified by the rect. It first checks if a state is currently being played, and if so, it retrieves the image for the current frame and stores it in the img variable.

```
def render(self, surface, rect, facing, invincibility=0):
    if self.playing is not None:
        img = self.get_image(facing)
        img_rect = img.get_rect()
        y_offset = self.sprites[self.playing].offset[1]
        x_offset = self.sprites[self.playing].offset[0] * [-1, 1][facing == self.sprites[self.playing].default_facing]
        y = rect.y + y_offset
        if facing == self.sprites[self.playing].default_facing:
            x = rect.x
        else:
            x = rect.right - img_rect.width
        x = x + x_offset
```

For the user to control the movement of the character, we can use pygame.key.pressed(). I created the algorithm using if and elif to for the keys used in character movement. The user can click space, w or arrow up to jump, a or arrow left to go left and d and arrow right to go right.

```
    keys = pygame.key.get_pressed()
    #checking event for jump
    if keys[pygame.K_SPACE] or keys[pygame.K_UP] or keys[pygame.K_w]:
        self.jump()
    #checking event for running to left
    if (keys[pygame.K_LEFT] or keys[pygame.K_a]) :
        self.v_x += -1
    #checking event for running to right
    elif (keys[pygame.K_RIGHT] or keys[pygame.K_d]) :
        self.v_x += 1
    #setting player facing by its velocity
    if self.v_x < 0 and self.facing != 'left':
        self.facing = 'left'
    if self.v_x > 0 and self.facing != 'right':
        self.facing = 'right'

    self.v_x *= self.walk_speed
```

Map Management

The map management in this game is handled by the LevelManager class. But first, we would like to talk about the block.py.

The block class is used to load the subsection of the level in Red Alert. It is responsible for rendering the terrain and static sprites of the level, as well as loading and storing information about various objects such as spawn points, coins, water, saw and spikes.

So, the TILED file or the pytmx is first loaded through the __init__ method.

```
self.tmx = pytmx.load_pygame(filename, pixelalpha=True)
```

Then, we use the render method to render the terrain and sprites to the surface and loop each layer of the “TileMap” object.

```
def render(self, surface):
    for layer in self.tmx.layers:
        #rendering level image
        if isinstance(layer, pytmx.TiledTileLayer):
            for x, y, image in layer.tiles():
                if image:

                    surface.blit(image, ( x * self.tmx.tilewidth,
                                          (y + 1) * self.tmx.tileheight - image.get_rect().height))
```

Finally in block, we put a make_map method to create the map which the player will navigate through. It creates not only the terrain, but also the static sprites and all of the obstacles and other objects that are present, but, we will talk more about that later. In make_mape it uses render to draw the terrain into the surface.

```
def make_map(self):

    self.render(self.surface)
    self.load_obstacles()
    return self.surface.convert_alpha(),self.fgsurface.convert_alpha()
```

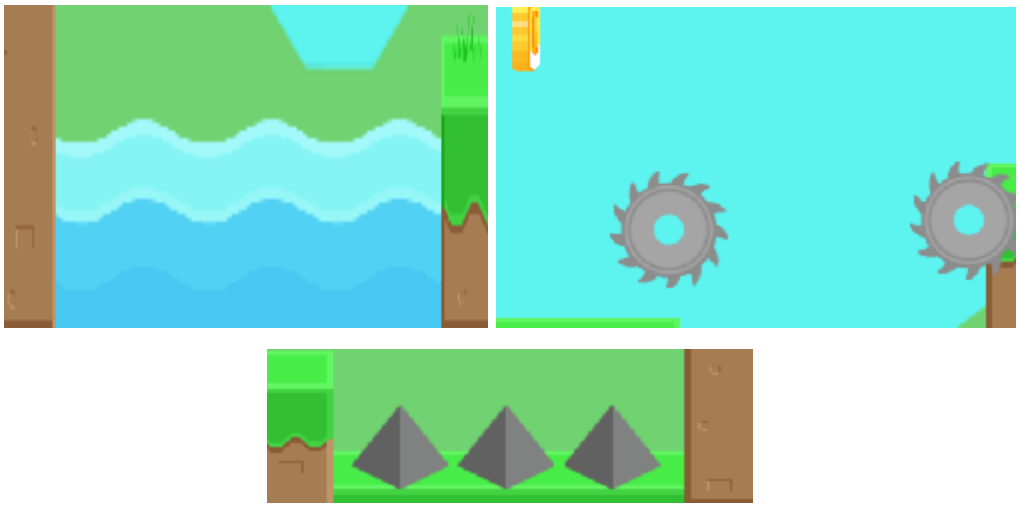
From the levelmanager, we are able to create the endless level system through the ‘update’ method. If the camera’s rectangle collides with the last block’s end collider, it randomly selects a new block to load except for the block that has already been loaded. To load a new block, the function generates a random index between 1 and the number

of levels minus 1 (since the first level is the starting block and we don't want to load it again), and checks if this index has already been used to load a block before. If it has, it generates a new index until it finds one that hasn't been used before.

```
if self.camera.get_relative_rect(self.blocks[-1][0].end).colliderect(self.camera.rect):
    while 1:
        blockindex=randint(1,len(self.levels)-1)
        if not blockindex in self.prev:
            break
    print("loading new Chunk",blockindex)
    self.load_new_level(blockindex)
```

To load a new block to the game, we use the method of “load_new_leve” in which it generates the game objects such as saw, spike, coins etc for the new block. It does this by initiating a block object for the block specified by “level-num”. It creates the game objects for the new block by looping through the list of each object and creating each spawn point. FInally, it appends the new block to the list of blocks and updates the x position for the next block to be generated.

```
#initializing Block
self.block = Block(self.x,0,os.path.join( self.levels[level_num]))
self.block_img,self.block_fg_img = self.block.make_map()
#offsetting new block collider x by current block x
for collider in self.block.collidables:
    collider.rect.x+=self.x

self.block.end.x+=self.x

# Initializing Water at water spawns
for water_loc in self.block.water_spawns:
    Water(self.x+water_loc[0], water_loc[1], groups=(self.all_sprites, self.waters))

# Initializing Saw at saw spawns

for saw_loc in self.block.saw_spawns:
    Saw(self.x+saw_loc[0], saw_loc[1], groups=(self.all_sprites, self.saws))

# Initializing Moving Saw at moving saw spawns
for moving_saw in self.block.moving_saws:
    points=[]
    for point in moving_saw:
        pos=point
        pos[0]+=self.x
        points.append(pygame.Vector2(*pos))

    Moving_Saw(points,groups=(self.all_sprites,self.saws))

# Initializing Spike at spikes spawns
for spike_loc in self.block.spikes:
    Spike(self.x+spike_loc[0], spike_loc[1], groups=(self.all_sprites, self.spikes))

# Initializing Coin at coins spawns
for coin_loc in self.block.coin_spawns:
    Coin(self.x+coin_loc[0], coin_loc[1], groups=(self.all_sprites, self.coins))

self.x+=self.block.width
self.blocks.append([self.block,self.block_img,self.block_fg_img])
```



The obstacles were loaded using load_resources, then The __init__() method initializes a new obstacle object and takes its position as arguments. It also takes a groups argument, which is a sequence of groups to add the obstacle object to. The obstacle object is initialized as a sprite and its rect attribute is set to a rectangle representing its position and size on the screen. The update() method updates the frame of the obstacle animation. The render() method draws the current frame of the obstacle animation on the given surface, taking into account the position of the camera. The obstacles are spawned in the Load_new_level method in levelmanager as described above. It will be explained further below.

Collidables

This is the algorithm which allows collisions between objects from the game. We can see from the collidable class in the Collidable.py which has a rect attribute to represent the position and size of collidable objects.

This is then linked to the obstacle.py file in which the Collidable class acts as the parent class of the ‘ObstacleRect’ and ‘ObstacleSlope’. ObstacleRect represents a rectangular obstacle that can be placed in the game world. The static attributes help determine if the obstacle is stationary or movable. Then the ObstacleSlope represents a sloped obstacle where it uses get_overlap_x and get_overlap_y t and return the x and y dimensions.

In the update method in levelmanager, there is a block of code that handles collision detection between the player character and the visible blocks, waters, saws, spikes, and coins in the level. This is done by calling the get_overlap_x and get_overlap_y methods of the collidables and using the returned overlap values to adjust the character’s position

Obstacles

To start, I want to explain the implementation on how the particles were implemented when the user touches the obstacles.

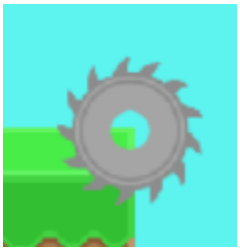
So, the particle system works in a way that The ParticleSystem class creates a number of particles (specified by the "n" parameter) with random properties such as position, velocity, color, and size. The Particle class, which is used by the ParticleSystem class, has an update method that updates the position of each particle based on its velocity and handles collisions with other collidable objects. The velocity of each particle also decreases over time to simulate the effect of drag. The render method of the Particle class is used to draw each particle on the screen. The ParticleSystem class also has update and render methods that call the corresponding methods for each particle. So, when there is a collision, the code will check it’s collision point in

To implement it when colliding with an obstacle, we can see it in the player.py:

I will put one example, when the player collide with water:

First, we use the update_particles method in player.py in which blocks are used to check for collision between the particles and the obstacles in the game level and delta_time is used to update the position of the particles based on the time that has passed since the last frame. If self.death_water is true, it sets the color of the particles to be the water color, it sets the position of the particle to be the center of the bottom of the rect. And sets the range of the speed of the particle to be (-10,10) for x and (-90,0) for y, and sets the number of particles to be 40.

Saw



It was first implemented by loading its resources using the __load_resources method and then the update method is used to update the animation and render is to draw the current frame of the saw at a given surface.

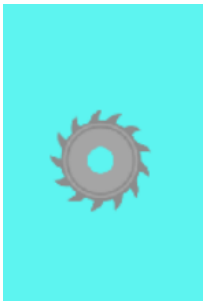
To load the saw animation, i inherited the code from animation.py called AnimatedSprite in which was used in the update and render method where in update, it uses the next_frame function in which It increments the current frame counter by the amount of time passed since the last frame divided by the frame duration. This allows smooth frames without any stutters or frame lag.

In render, the render method of the animated_sprite object is called with the surface and the camera's relative position of the saw's rect as the parameters, to render the current frame of the animation on the game screen.

To actually render it to it's spawn, we use this code in the levelmanager.py:

```
for saw_loc in self.block.saw_spawns:
    Saw(self.x+saw_loc[0], saw_loc[1], groups=(self.all_sprites, self.saws))
```

Moving Saw



The moving saw basically inherits the class of the saw however, there are the codes to allow it to move the saw to a targeted point. The difference of the moving saw is that the update method takes in the delta_time as a parameter, which is the time passed since the last update. The method then uses this delta_time to update the position of the saw. It starts by checking if the duration counter has reached 1, which means that the saw has reached its target point. Then, it will go to another target point and reset the counter. Then it calculates the time and distance to the next target and goes to a loop.

This sets the moving saw to surface and in position 0:

```
for moving_saw in self.block.moving_saws:
    points=[]
    for point in moving_saw:
        pos=point
        pos[0]+=self.x
        points.append(pygame.Vector2(*pos))

    Moving_Saw(points,groups=(self.all_sprites,self.saws))
```

Spikes

So the spikes uses the pygame sprite class and overrides the init and render method. So, once again, it needs to load texture using the __load_resources method and then we can define how big the spike is using get_rect which has the width 70, and height 70. The layer is set to 3 which is the same as the other collider.

Then, the render is used to draw the obstacle in the surface and the

```
for spike_loc in self.block.spikes:
    Spike(self.x+spike_loc[0], spike_loc[1], groups=(self.all_sprites, self.spikes))
```

Is used to load the spikes in the game

Water

Finally, we have the water obstacle. As usual, we load the resource in the _load_resources method. The init method then initializes the self.rect variable with a pygame.Rect object, which stores the position and size of the water sprite. It also calls the pygame.sprite.Sprite.__init__ method to initialize the sprite, and calls self._load_resources() to load the resources

The update method is used to change the water to the next frame and the render is to put

down the water at the surface.

Then, render method to put it in a surface and finally, to put it in the map, we also use the data from the block.py:

```
for water_loc in self.block.water_spawns:
    Water(self.x+water_loc[0], water_loc[1], groups=(self.all_sprites, self.waters))
```

Scoring System

First, the best score is achieved through this code:

```
if self.score>self.bestscore:
    self.bestscore=self.score
```

As the player moves forward, the score will increase by 1 for every block the player passes. The player's x position is checked on the x position of the last block that was added to the level. If the player's x position is greater than the x position of the last block, it means the player has moved past that block and a new block should be added to the level. The distance traveled by the player is calculated by taking the x position of the player's rect, relative to the camera, and dividing it by 500. So, it increases the player's score every time they move 500 pixels to the right.

```
#updating the score
if level.score<int((level.camera.get_relative_rect(self.rect).x+self.rect.x)/500)-1:
    level.score=int((level.camera.get_relative_rect(self.rect).x+self.rect.x)/500)-1
```

Other than distance, the scoring could also be done by the amount of coins collected. The hitbox of the coin is defined through the rect attribute. If the collision between the user and the coin occurs, the coin is removed from the all_sprites group and the score is incremented by 1.


```
#Handle the collision of Coins with Player
def _coin_collision(self, coin_group):
    coins = pygame.sprite.spritecollide(self, coin_group, False)
    #checking whether any coin collided with player or not
    for coin in coins:
        coin.pickup()
        self.coins += 1
```

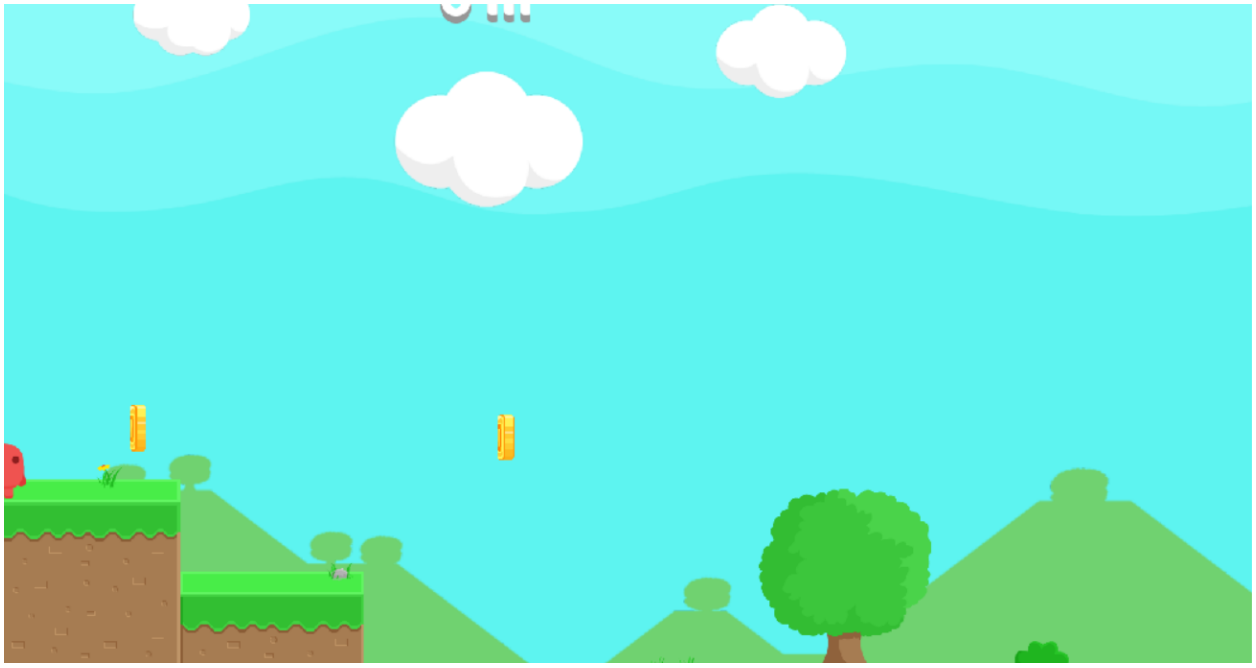
Camera

It tracks the player's movement and it has a height of 1200 and 2133 of width. In the camera class, there are two similar codes which are the get_reverse_relative_pos and the get_relative_pos. The get_relative_pos is to check the current position and relative_pos to check the position before. The move_to method that allows the camera to move as the player moves to the right.

The camera also works in such ways that if the camera is passing the end of the chunk, it is going to load another chunk.

Background

Illusion of depth in the background of the game, through the use of multiple layers of background images that move at different speeds. The class uses the parallax_multiplier attribute to determine the speed at which each layer should move and the camera object to ensure that each layer is rendered in the correct position.



```
#background textures folder path
parallax_folder = os.path.join("Assets","Backgrounds")
self.textures = {
   }
#loading background layers textures
for i in [0,9,10]:
    self.textures[str(i)]= pygame.image.load(os.path.join(parallax_folder, f"{i}.png")).convert_alpha()

# make parallax layers
self.parallax_bg = []

for i in [0,9,10]:
   self.parallax_bg.append(ParallaxLayer(self.textures[str(i)], i*0.08, self.camera, self.block.width, self.block.height))
pygame.mixer.music.play(1)
```

The parallax folder sets up a folder path to where the background images are located. Then it creates an empty dictionary called textures. It then loads the background images by iterating through a list containing the numbers 0, 9, and 10. Each image is loaded and added to the textures dictionary with the key being the number it was loaded from.

After that, it creates an empty list called parallax_bg. Then it iterates through the same

list of numbers and creates a new instance of the ParallaxLayer class for each number, passing in the corresponding texture from the textures dictionary, a parallax multiplier that is calculated by multiplying the number by 0.08, the camera object, the width and height of the block. Finally, it starts playing the background music.

HUD

This is the code to render the HUD into the game:

```
def render_hud(self, surface):
    x = 0
    y = surface.get_rect().height -118


    surface.blit(self.coin_icon,(30,30))
    Font.put_text(surface, "x"+str(self.player.coins), (188,25), (255, 255, 255),"u")

    Font.put_text(surface, str(self.bestscore)+ " m", (settings.SCREEN_WIDTH/2,60), (150, 150, 150),"huge",True)
    Font.put_text(surface, str(self.bestscore)+ " m", (settings.SCREEN_WIDTH/2,50), (255, 255, 255),"huge",True)
    Font.put_text(surface, str(self.score)+ " m", (settings.SCREEN_WIDTH/2,160), (150, 150, 150),"u",True)
    Font.put_text(surface, str(self.score)+ " m", (settings.SCREEN_WIDTH/2,150), (255, 255, 255),"u",True)
```

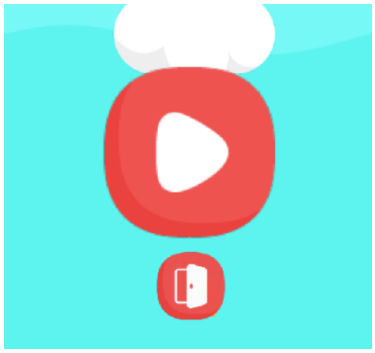
As the score, the coins and score system has been explained above. However, for the design, the placement of the coins for example is in the coordinate of 188,25 with the color code of (255,255,255). The score is also implemented the same where for example for the self.bestscore where it is placed in the screen_width/2.60 with the color code of (150,150,150). I created a shadow effect in which it has the placement of screend_width/2.50 and colored black. The font is implemented through the font.py in this code:

```
Font._font["normal"] = pygame.font.Font(os.path.join(settings.font_folder, 'Baloo-Regular.ttf'), 24)
Font._font["small"] = pygame.font.Font(os.path.join(settings.font_folder, 'Baloo-Regular.ttf'), 14)
Font._font["big"] = pygame.font.Font(os.path.join(settings.font_folder, 'Baloo-Regular.ttf'), 40)
Font._font["huge"] = pygame.font.Font(os.path.join(settings.font_folder, 'Baloo-Regular.ttf'), 60)
Font._font["u"] = pygame.font.Font(os.path.join(settings.font_folder, 'Baloo-Regular.ttf'), 80)
```

This code determines the font that is being used for the writing in the HUD.

Button & Cursor

The play, restart and exit buttons are implemented using the Button class in the script. Each button is created with a position, size, an image, and a callback function. The callback function is called when the button is clicked on.



The play button is created with the image "Assets/UI/Play.png" and the callback function "self.start_game" which starts the game and plays the background music. The restart button is created with the image "Assets/UI/Restart.png" and the callback function "self.reset_level" which resets the level of the game. The exit button is created with the image "Assets/UI/Exit.png" and the callback function "lambda: goto_scene("quit")" which quits the game.

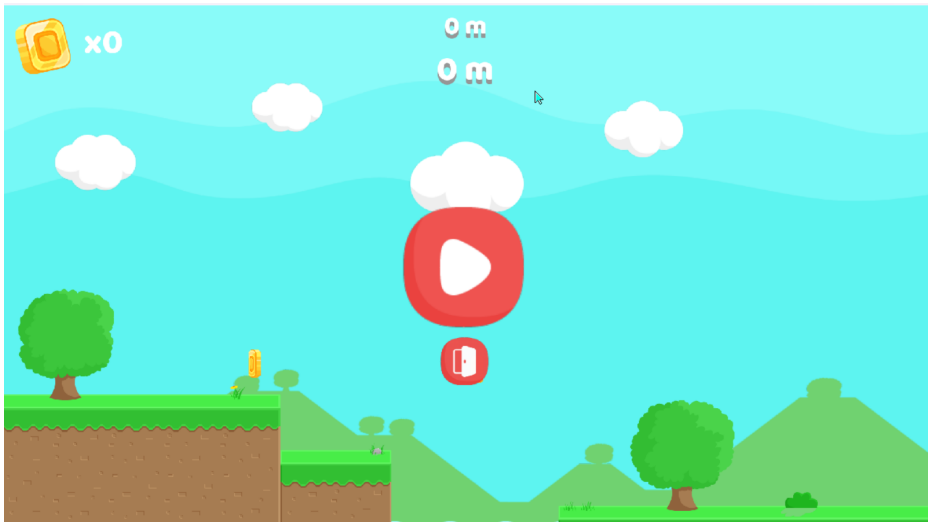
```
#loading UI icons
self.parallax_fg = []
play_icon=pygame.transform.smoothscale(pygame.image.load("Assets/UI/Play.png").convert_alpha(),(300,300))
self.play_button=Button((settings.SCREEN_WIDTH/2,settings.SCREEN_HEIGHT/2),(300,300),play_icon,self.start_game)

retry_icon=pygame.transform.smoothscale(pygame.image.load("Assets/UI/Restart.png").convert_alpha(),(300,300))
self.retry_button=Button((settings.SCREEN_WIDTH/2,settings.SCREEN_HEIGHT/2),(300,300),retry_icon,self.reset_Level)

exit_icon=pygame.transform.smoothscale(pygame.image.load("Assets/UI/Exit.png").convert_alpha(),(120,120))
self.exit_button=Button((settings.SCREEN_WIDTH/2,(settings.SCREEN_HEIGHT/3)*2 +20),(120,120),exit_icon,lambda: goto_scene("quit"))
```

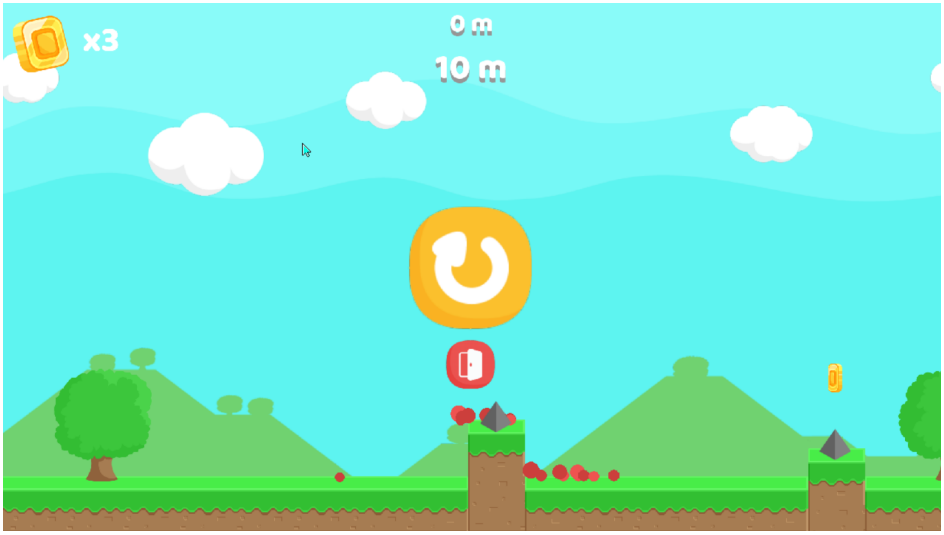
On the other hand, The cursor is implemented using the Mouse class in the script. It has several methods such as init, set_visible, is_visible, render, get_pos and set_pos. The init method is used to load the cursor image "Assets/UI/Cursor/cursor.png" and set the mouse cursor to be invisible. The set_visible method is used to set the visibility of the cursor. The is_visible method is used to check whether the cursor is visible or not. The render method is used to render the cursor image on the screen. The get_pos method is used to get the position of the cursor on the screen. The set_pos method is used to set the position

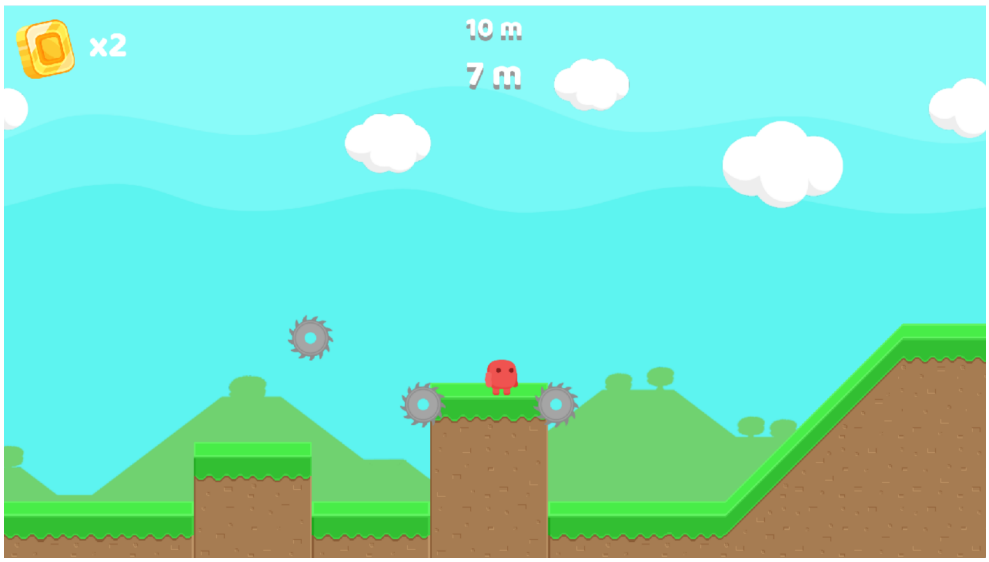
of the cursor on the screen. The cursor is rendered on the screen by calling the render method and its position is updated by calling the get_pos method. In order for the cursor to only show on the menu and not during the game, we can use this code:

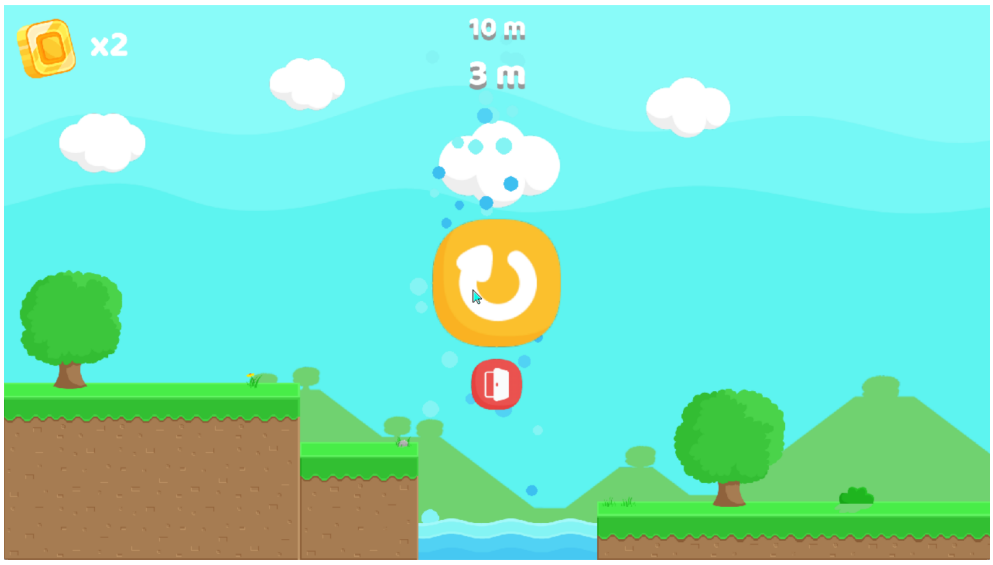
```
def render(self, surface):
    
    if Mouse.is_visible() and self.status=="Game":
        Mouse.set_visible(False)
    if not Mouse.is_visible() and self.status=="GameOver":
        Mouse.set_visible(True)
```

Evidence of a Working Program






x2
10 m
7 m


x2
10 m
3 m

Reflection

In this final project, There is a lot that I have learned. We can start with the use of pytmx, which before this final project, I had never even heard of pytmx. This project taught me to use pytmx and the "Tiled" software to create my map. I also learn about the basics of an endless platformer through many youtube video demonstrations, which I will link to the sources below. Furthermore, this project helped me understand why we need math in programming, which I used to track my character and other algorithms. With python, I realized how many things could be made, starting from a game, a website and many, many more. After this project, I will study much more about python, and I want to master the ways of its logic.

There are many things that I have not understood or mastered yet, as there are still many things to be explored in a platformer game. I couldn't create my design as I didn't have the skill and creativity to draw my sprites. My code is not as neat as other people's, as I use a lot of inheritance, so I have difficulties organizing my code without creating an error. I wouldn't be able to do my projects without the internet as they give me considerable knowledge and tutorials to create my game. It gave me pieces of information and helped me to debug my errors. Finally, thank the people around me who helped me throughout this project with design advice, python logic and many more. These are necessary to be able to create **Red Alert**.

Sources

Resources:

https://bayat.itch.io/platform-game-assets

https://www.zapsplat.com/sound-effect-category/game-music-and-loops/

https://www.mapeditor.org/download.html

https://www.pygame.org/docs/

https://github.com/bitcraft/pytmx

Learning Resources:

https://www.youtube.com/watch?v=YWN8GcmJ-jA

https://www.youtube.com/watch?v=Ongc4EVqRjo

https://www.youtube.com/watch?v=N6xqCwblyiw

https://www.youtube.com/watch?v=Gmrf_3LbXu0

https://github.com/cjddmut/Unity-2D-Platformer-Controller

https://github.com/clear-code-projects/2D_Platformer_Logic

https://www.youtube.com/watch?v=wWAjRwiYmpw

https://pastebin.com/a6MhiExv

https://www.youtube.com/watch?v=MO2yJhgtMes