**Analysis of Python Mini Project: Game using Pygame**

**Introduction**

The objective of this project is to design and implement two engaging 2D games using Python's Pygame library. The project showcases basic game development principles, such as sprite control, collision detection, scoring systems, and game loops. The two games — "Space Invaders", a classic arcade-style shooting game, and "The Assassin", a stealth-based obstacle dodging game — demonstrate different gameplay mechanics while maintaining efficient structure and readability for learning and further development.

**Objectives**

The primary objectives of the project "Space Invaders & The Assassin" are:

* To understand the fundamentals of game development using Python and the Pygame library.
* To design and develop two functional 2D games showcasing different game genres and mechanics — a classic arcade shooter and a stealth-based strategy game.
* To implement real-time game features such as player movement, enemy behavior, collision detection, score tracking, and level progression.
* To explore and apply object-oriented programming concepts in game development through structured code using classes and reusable components.
* To develop an interactive and user-friendly gaming experience by integrating sounds, graphics, animations, and intuitive controls.
* To improve problem-solving and debugging skills by handling logical challenges such as collision mechanics, enemy AI, and frame rate synchronization.
* To lay a foundation for future projects in more advanced game development, including 3D games or web/mobile game deployment.

**Software Requirements**

* Operating System: Windows 10 or above / Linux / macOS
* Programming Language: Python 3.x
* Game Development Library: Pygame (latest stable version)
* Code Editor/IDE: Visual Studio Code, PyCharm, or any preferred Python IDE
* Additional Tools:
  + Python package installer (pip) for installing Pygame
  + Image editing software (optional) for creating/modifying game sprites (e.g., GIMP, Photoshop)
  + Audio editing software (optional) for sound effects (e.g., Audacity)

**Hardware Requirements**

* Processor: Intel i3 or equivalent (minimum)
* RAM: 4 GB or more recommended
* Graphics: Integrated graphics sufficient for 2D game rendering
* Storage: Minimum 500 MB free space for project files and assets
* Input Devices: Keyboard and mouse (for game controls and development)

**Methodology**

**Space Invaders**

Space Invaders Game

│  
├── Initialization  
│ ├── Initialize Pygame  
│ ├── Set Screen Dimensions  
│ ├── Load Background, Icons, and Images  
│ ├── Load Sounds and Play Background Music  
│ ├── Set Game Title and Icon  
│ ├── Initialize Player Properties  
│ ├── Generate Multiple Enemies with Random Positions  
│ ├── Initialize Bullet Properties  
│ └── Setup Fonts for Score and Game Over Text  
│  
├── Main Game Loop  
│ ├── Fill Background and Draw Background Image  
│ ├── Handle Events  
│ │ ├── Quit Event  
│ │ ├── Key Presses (Left, Right, Space)  
│ │ └── Key Releases (Stop Movement)  
│ ├── Update Player Position (with Screen Boundaries)  
│ ├── Enemy Logic  
│ │ ├── Move Enemy  
│ │ ├── Change Direction at Edges and Move Down  
│ │ ├── Detect Game Over (Enemy Reaches Bottom)  
│ │ ├── Detect Collision with Bullet  
│ │ └── Reset Enemy and Increase Score on Collision  
│ ├── Bullet Logic  
│ │ ├── Fire Bullet if State is "ready"  
│ │ ├── Move Bullet Upwards  
│ │ └── Reset Bullet if Off-Screen  
│ ├── Draw Player, Enemies, Bullet, and Score  
│ ├── Display "Game Over" if Triggered  
│ └── Update the Display (Render Everything)  
│  
└── User Interaction  
├── Move Player with Arrow Keys  
├── Fire Bullet with Spacebar  
└── Exit Game with Close Button

**The Assassin**

Start Game.run()

|

V

Load & play music (loop)

Load & play ambience sound (loop)

|

V

+-------------------------+

| Start Game Loop | <---------------------------+

+-------------------------+ |

| Loop

V

Clear display surfaces

Draw background on display\_2

Reduce screen shake timer if > 0

|

V

Are all enemies defeated?

|Yes No

| |

Increase transition timer Is transition < 0?

| |Yes

Is transition > 30? Increase transition timer

|Yes |

| No

Load next level (increment level) |

| V

V Is player dead?

Is player dead? |Yes

|Yes |

| Increase dead timer

Increase dead timer |

If dead timer >= 10, increase transition timer

If dead timer > 40, reload level

|No |No

| |

V V

Smoothly scroll camera to player position

Spawn leaf particles randomly

Update & render clouds

Render tilemap

Update & render enemies

|

V

Is player alive?

|Yes |No

| |

Update player Skip player update

Render player

|

V

Update & render projectiles

- Move projectile

- Check collisions with tilemap

- Check collisions with player

- Remove if collided or timeout

|

V

Update & render sparks

Remove finished sparks

|

V

Create & render silhouette mask for shading

|

V

Update & render particles (leaves sway)

Remove finished particles

|

V

Handle user input events:

- Quit event -> exit

- Keydown -> set movement, jump, dash

- Keyup -> unset movement

|

V

Is transition active?

|Yes

| Draw transition circle effect

|

V

Blit display to screen with screen shake offset

Update display

Cap FPS at 60

|

V

Repeat Game Loop

**Implementation Details**

* The games Space Invaders and The Assassin are implemented using Python programming language with the Pygame library for 2D game development.
* The game logic includes player movement, enemy behavior, collision detection, scoring, and game state management.
* Graphics are rendered using Pygame’s surface and sprite classes.
* Sound effects and background music are integrated to enhance gameplay experience.
* The games utilize event handling for player input (keyboard controls).
* Modular programming approach is followed by dividing code into multiple classes and functions for better organization and reusability.
* The game loop continuously updates game state, renders graphics, and processes user input until the game ends.

**Tools and Techniques**

**Tools:**

* Python 3.x: Primary programming language.
* Pygame: Library for handling game graphics, events, and sounds.
* Visual Studio Code / PyCharm: IDEs used for writing and debugging the code.
* Git (optional): Version control system for managing code changes.
* Image and Audio Editing Software (optional): Tools like GIMP and Audacity for creating/editing game assets.

**Techniques:**

* Event-driven Programming: Handling user inputs (keyboard events).
* Object-Oriented Programming (OOP): Using classes for players, enemies, projectiles, and game objects.
* Collision Detection: Detecting interactions between player bullets and enemies, or player and enemy.
* Game Loop: Continuous loop to update and render the game state.
* Sprite Management: Efficient handling of multiple game objects with Pygame sprites.
* Modular Design: Dividing the code into modules/functions for clarity and maintainability.

**Source Code**

**For Space Invaders**

# Importing required libraries

import math      # For calculating distance between bullet and enemy

import random    # For generating random positions for enemies

import pygame    # Main library for game development

from pygame import mixer  # For playing sound/music

# Initialize all pygame modules

pygame.init()

# Create the game screen with width = 800 and height = 600

screen = pygame.display.set\_mode((800, 600))

# Load background image

background = pygame.image.load('background.png')

# Load and play background music on loop (-1 means infinite loop)

mixer.music.load("background.wav")

mixer.music.play(-1)

# Set window title and icon

pygame.display.set\_caption("Space Invaders")

icon = pygame.image.load('ufo.png')

pygame.display.set\_icon(icon)

# Load player spaceship image and set initial position

playerImg = pygame.image.load('player.png')

playerX = 370  # Horizontal position

playerY = 480  # Vertical position (fixed)

playerX\_change = 0  # Change in X (used for movement)

# Load enemy spaceship images and set their initial positions and speed

enemyImg = []         # List to hold enemy images

enemyX = []           # List to hold X positions of enemies

enemyY = []           # List to hold Y positions of enemies

enemyX\_change = []    # List to hold X-axis movement speed

enemyY\_change = []    # List to hold Y-axis drop after hitting screen edge

num\_of\_enemies = 6    # Number of enemies on screen

# Generate multiple enemies

for i in range(num\_of\_enemies):

    enemyImg.append(pygame.image.load('enemy.png'))

    enemyX.append(random.randint(0, 736))   # Random X position

    enemyY.append(random.randint(50, 150))  # Random Y position

    enemyX\_change.append(4)                # Horizontal speed

    enemyY\_change.append(40)               # Drop down when edge is hit

# Bullet setup

bulletImg = pygame.image.load('bullet.png')

bulletX = 0

bulletY = 480  # Initial Y position of bullet

bulletX\_change = 0

bulletY\_change = 10  # Speed of bullet going upward

bullet\_state = "ready"  # "ready" = can fire, "fire" = bullet is on screen

# Score setup

score\_value = 0

font = pygame.font.Font('freesansbold.ttf', 32)

textX = 10  # X position of score display

testY = 10  # Y position of score display

# Game Over font

over\_font = pygame.font.Font('freesansbold.ttf', 64)

# Function to show score on screen

def show\_score(x, y):

    score = font.render("Score : " + str(score\_value), True, (255, 255, 255))

    screen.blit(score, (x, y))

# Function to display "GAME OVER" text

def game\_over\_text():

    over\_text = over\_font.render("GAME OVER", True, (255, 255, 255))

    screen.blit(over\_text, (200, 250))

# Function to draw player spaceship on screen

def player(x, y):

    screen.blit(playerImg, (x, y))

# Function to draw enemy spaceship on screen

def enemy(x, y, i):

    screen.blit(enemyImg[i], (x, y))

# Function to fire the bullet

def fire\_bullet(x, y):

    global bullet\_state

    bullet\_state = "fire"  # Bullet is now moving

    screen.blit(bulletImg, (x + 16, y + 10))  # Adjust bullet position for center

# Function to detect collision between bullet and enemy

def isCollision(enemyX, enemyY, bulletX, bulletY):

    distance = math.sqrt(math.pow(enemyX - bulletX, 2) + (math.pow(enemyY - bulletY, 2)))

    if distance < 27:  # If they are close enough, it’s a hit

        return True

    else:

        return False

# Main Game Loop

running = True

while running:

    # Fill the screen with black color before drawing everything

    screen.fill((0, 0, 0))

    # Draw background image

    screen.blit(background, (0, 0))

    # Loop through events (keyboard, mouse, etc.)

    for event in pygame.event.get():

        if event.type == pygame.QUIT:  # Quit the game

            running = False

        # If a key is pressed down

        if event.type == pygame.KEYDOWN:

            if event.key == pygame.K\_LEFT:  # Move left

                playerX\_change = -5

            if event.key == pygame.K\_RIGHT:  # Move right

                playerX\_change = 5

            if event.key == pygame.K\_SPACE:  # Fire bullet

                if bullet\_state == "ready":  # Only fire if bullet is not already moving

                    bulletSound = mixer.Sound("laser.wav")

                    bulletSound.play()

                    bulletX = playerX  # Set bullet to current player position

                    fire\_bullet(bulletX, bulletY)

        # If key is released, stop movement

        if event.type == pygame.KEYUP:

            if event.key == pygame.K\_LEFT or event.key == pygame.K\_RIGHT:

                playerX\_change = 0

    # Update player position

    playerX += playerX\_change

    # Keep player within screen boundaries

    if playerX <= 0:

        playerX = 0

    elif playerX >= 736:  # 800 - player image width (64)

        playerX = 736

    # Enemy Movement and Collision Detection

    for i in range(num\_of\_enemies):

        # Check if any enemy has reached close to the player (game over)

        if enemyY[i] > 440:

            for j in range(num\_of\_enemies):

                enemyY[j] = 2000  # Move all enemies off-screen

            game\_over\_text()

            break

        # Move enemy left/right

        enemyX[i] += enemyX\_change[i]

        # Reverse direction and move down when hitting the edge

        if enemyX[i] <= 0:

            enemyX\_change[i] = 4

            enemyY[i] += enemyY\_change[i]

        elif enemyX[i] >= 736:

            enemyX\_change[i] = -4

            enemyY[i] += enemyY\_change[i]

        # Check for collision between this enemy and bullet

        collision = isCollision(enemyX[i], enemyY[i], bulletX, bulletY)

        if collision:

            explosionSound = mixer.Sound("explosion.wav")

            explosionSound.play()

            bulletY = 480  # Reset bullet position

            bullet\_state = "ready"  # Bullet can be fired again

            score\_value += 1  # Increase score

            # Respawn enemy at random position

            enemyX[i] = random.randint(0, 736)

            enemyY[i] = random.randint(50, 150)

        # Draw enemy

        enemy(enemyX[i], enemyY[i], i)

    # Bullet Movement

    if bulletY <= 0:

        bulletY = 480  # Reset bullet Y position

        bullet\_state = "ready"  # Ready to fire again

    if bullet\_state == "fire":

        fire\_bullet(bulletX, bulletY)  # Draw the bullet

        bulletY -= bulletY\_change      # Move bullet upward

    # Draw player and score

    player(playerX, playerY)

    show\_score(textX, testY)

    # Update the screen with all drawings

    pygame.display.update()

**For The Assassin**

import os  # Operating system interaction

import sys  # System-specific parameters and functions

import math  # Math functions

import random  # Random number generation

import pygame  # Pygame library for game development

# Import custom modules and classes for game components and utilities

from scripts.utils import load\_image, load\_images, Animation

from scripts.entities import PhysicsEntity, Player, Enemy

from scripts.tilemap import Tilemap

from scripts.clouds import Clouds

from scripts.particle import Particle

from scripts.spark import Spark

class Game:  # Main game class

    def \_\_init\_\_(self):  # Initialization of the game

        pygame.init()  # Initialize pygame modules

        pygame.display.set\_caption('The Assassin')  # Set window title

        self.screen = pygame.display.set\_mode((640, 480))  # Create main screen window with size 640x480

        self.display = pygame.Surface((320, 240), pygame.SRCALPHA)  # Create smaller surface for rendering with alpha

        self.display\_2 = pygame.Surface((320, 240))  # Create another surface for layered rendering

        self.clock = pygame.time.Clock()  # Create clock to control FPS

        self.movement = [False, False]  # Movement flags for left and right

        # Load all game assets such as images and animations in a dictionary for easy access

        self.assets = {

            'decor': load\_images('tiles/decor'),  # Load decorative tiles images

            'grass': load\_images('tiles/grass'),  # Load grass tiles images

            'large\_decor': load\_images('tiles/large\_decor'),  # Load large decorative tiles images

            'stone': load\_images('tiles/stone'),  # Load stone tiles images

            'player': load\_image('entities/player.png'),  # Load player image (static)

            'background': load\_image('background.png'),  # Load background image

            'clouds': load\_images('clouds'),  # Load clouds images

            'enemy/idle': Animation(load\_images('entities/enemy/idle'), img\_dur=6),  # Load enemy idle animation

            'enemy/run': Animation(load\_images('entities/enemy/run'), img\_dur=4),  # Load enemy running animation

            'player/idle': Animation(load\_images('entities/player/idle'), img\_dur=6),  # Player idle animation

            'player/run': Animation(load\_images('entities/player/run'), img\_dur=4),  # Player running animation

            'player/jump': Animation(load\_images('entities/player/jump')),  # Player jumping animation

            'player/slide': Animation(load\_images('entities/player/slide')),  # Player sliding animation

            'player/wall\_slide': Animation(load\_images('entities/player/wall\_slide')),  # Player wall sliding animation

            'particle/leaf': Animation(load\_images('particles/leaf'), img\_dur=20, loop=False),  # Leaf particle animation

            'particle/particle': Animation(load\_images('particles/particle'), img\_dur=6, loop=False),  # Misc particle animation

            'gun': load\_image('gun.png'),  # Gun image

            'projectile': load\_image('projectile.png'),  # Projectile image

        }

        # Load sound effects into a dictionary

        self.sfx = {

            'jump': pygame.mixer.Sound('data/sfx/jump.wav'),  # Jump sound effect

            'dash': pygame.mixer.Sound('data/sfx/dash.wav'),  # Dash sound effect

            'hit': pygame.mixer.Sound('data/sfx/hit.wav'),  # Hit sound effect

            'shoot': pygame.mixer.Sound('data/sfx/shoot.wav'),  # Shoot sound effect

            'ambience': pygame.mixer.Sound('data/sfx/ambience.wav'),  # Ambient background sound

        }

        # Set volume levels for each sound effect

        self.sfx['ambience'].set\_volume(0.2)  # Ambient sound volume

        self.sfx['shoot'].set\_volume(0.4)  # Shooting sound volume

        self.sfx['hit'].set\_volume(0.8)  # Hit sound volume

        self.sfx['dash'].set\_volume(0.3)  # Dash sound volume

        self.sfx['jump'].set\_volume(0.7)  # Jump sound volume

        self.clouds = Clouds(self.assets['clouds'], count=16)  # Create clouds effect with 16 cloud sprites

        self.player = Player(self, (50, 50), (8, 15))  # Create player object at position (50, 50) with size (8, 15)

        self.tilemap = Tilemap(self, tile\_size=16)  # Create tilemap to manage level tiles with 16x16 tiles

        self.level = 0  # Starting level index

        self.load\_level(self.level)  # Load level 0

        self.screenshake = 0  # Initialize screen shake effect amount

    def load\_level(self, map\_id):  # Load level data by id (map file)

        self.tilemap.load('data/maps/' + str(map\_id) + '.json')  # Load the map json file

        self.leaf\_spawners = []  # List of areas that spawn leaf particles

        for tree in self.tilemap.extract([('large\_decor', 2)], keep=True):  # Extract large decor tiles matching criteria

            self.leaf\_spawners.append(pygame.Rect(4 + tree['pos'][0], 4 + tree['pos'][1], 23, 13))  # Create rect for leaf spawn area

        self.enemies = []  # List of enemy objects

        for spawner in self.tilemap.extract([('spawners', 0), ('spawners', 1)]):  # Extract spawner tiles for player/enemies

            if spawner['variant'] == 0:  # If variant 0, set player starting position

                self.player.pos = spawner['pos']  # Set player position

                self.player.air\_time = 0  # Reset player's air time

            else:

                self.enemies.append(Enemy(self, spawner['pos'], (8, 15)))  # Create enemy at spawner position

        self.projectiles = []  # List for projectiles in the game

        self.particles = []  # List for particle effects in the game

        self.sparks = []  # List for spark effects

        self.scroll = [0, 0]  # Current scrolling offset for camera

        self.dead = 0  # Player death count or flag

        self.transition = -30  # Transition timer/flag for level change

    def run(self):  # Main game loop to run the game

        pygame.mixer.music.load('data/music.wav')  # Load background music

        pygame.mixer.music.set\_volume(0.5)  # Set music volume

        pygame.mixer.music.play(-1)  # Play music in a loop

        self.sfx['ambience'].play(-1)  # Play ambient sound effect in a loop

        while True:  # Game loop iteration

            self.display.fill((0, 0, 0, 0))  # Clear the display surface with transparent black

            self.display\_2.blit(self.assets['background'], (0, 0))  # Draw the background onto display\_2

            self.screenshake = max(0, self.screenshake - 1)  # Decrease screen shake effect over time

            if not len(self.enemies):  # If all enemies defeated

                self.transition += 1  # Increase transition timer

                if self.transition > 30:  # After delay

                    self.level = min(self.level + 1, len(os.listdir('data/maps')) - 1)  # Move to next level or last map

                    self.load\_level(self.level)  # Load new level

            if self.transition < 0:  # If during transition start delay

                self.transition += 1  # Increment transition

            if self.dead:  # If player is dead

                self.dead += 1  # Increment dead timer

                if self.dead >= 10:

                    self.transition = min(30, self.transition + 1)  # Start transition in after death

                if self.dead > 40:

                    self.load\_level(self.level)  # Reload current level

            # Smoothly scroll camera to player position

            self.scroll[0] += (self.player.rect().centerx - self.display.get\_width() / 2 - self.scroll[0]) / 30

            self.scroll[1] += (self.player.rect().centery - self.display.get\_height() / 2 - self.scroll[1]) / 30

            render\_scroll = (int(self.scroll[0]), int(self.scroll[1]))  # Integer scroll offset for rendering

            # Spawn leaf particles randomly within leaf spawner rectangles

            for rect in self.leaf\_spawners:

                if random.random() \* 49999 < rect.width \* rect.height:

                    pos = (rect.x + random.random() \* rect.width, rect.y + random.random() \* rect.height)  # Random position inside spawner

                    self.particles.append(Particle(self, 'leaf', pos, velocity=[-0.1, 0.3], frame=random.randint(0, 20)))

            self.clouds.update()  # Update cloud positions

            self.clouds.render(self.display\_2, offset=render\_scroll)  # Render clouds with scrolling offset

            self.tilemap.render(self.display, offset=render\_scroll)  # Render tiles on display surface

            # Update and render enemies, remove them if killed

            for enemy in self.enemies.copy():

                kill = enemy.update(self.tilemap, (0, 0))

                enemy.render(self.display, offset=render\_scroll)

                if kill:

                    self.enemies.remove(enemy)

            if not self.dead:  # If player is alive

                self.player.update(self.tilemap, (self.movement[1] - self.movement[0], 0))  # Update player movement input

                self.player.render(self.display, offset=render\_scroll)  # Render player

            # For each projectile, update position, check collisions, and render

            for projectile in self.projectiles.copy():

                projectile[0][0] += projectile[1]  # Move projectile horizontally

                projectile[2] += 1  # Increment projectile timer

                img = self.assets['projectile']  # Get projectile image

                self.display.blit(img, (projectile[0][0] - img.get\_width() / 2 - render\_scroll[0], projectile[0][1] - img.get\_height() / 2 - render\_scroll[1]))  # Render projectile

                if self.tilemap.solid\_check(projectile[0]):  # Check if projectile hits solid tile

                    self.projectiles.remove(projectile)  # Remove projectile

                    for i in range(4):  # Create sparks on impact

                        self.sparks.append(Spark(projectile[0], random.random() - 0.5 + (math.pi if projectile[1] > 0 else 0), 2 + random.random()))

                elif projectile[2] > 360:  # Remove projectile if too old

                    self.projectiles.remove(projectile)

                elif abs(self.player.dashing) < 50:  # Check collision with player if not dashing strongly

                    if self.player.rect().collidepoint(projectile[0]):  # If projectile hits player

                        self.projectiles.remove(projectile)  # Remove projectile

                        self.dead += 1  # Increase death counter

                        self.sfx['hit'].play()  # Play hit sound

                        self.screenshake = max(16, self.screenshake)  # Trigger screen shake effect

                        for i in range(30):  # Create sparks and particles on player hit

                            angle = random.random() \* math.pi \* 2

                            speed = random.random() \* 5

                            self.sparks.append(Spark(self.player.rect().center, angle, 2 + random.random()))

                            self.particles.append(Particle(self, 'particle', self.player.rect().center, velocity=[math.cos(angle + math.pi) \* speed \* 0.5, math.sin(angle + math.pi) \* speed \* 0.5], frame=random.randint(0, 7)))

            # Update and render sparks; remove them if finished

            for spark in self.sparks.copy():

                kill = spark.update()

                spark.render(self.display, offset=render\_scroll)

                if kill:

                    self.sparks.remove(spark)

            # Create a silhouette mask effect around the display for shading

            display\_mask = pygame.mask.from\_surface(self.display)

            display\_sillhouette = display\_mask.to\_surface(setcolor=(0, 0, 0, 180), unsetcolor=(0, 0, 0, 0))

            for offset in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  # Draw shadow offsets around edges

                self.display\_2.blit(display\_sillhouette, offset)

            # Update and render particles; leaves slightly sway side to side

            for particle in self.particles.copy():

                kill = particle.update()

                particle.render(self.display, offset=render\_scroll)

                if particle.type == 'leaf':

                    particle.pos[0] += math.sin(particle.animation.frame \* 0.035) \* 0.3  # Sway leaves

                if kill:

                    self.particles.remove(particle)

            # Handle pygame events such as keyboard and window close

            for event in pygame.event.get():

                if event.type == pygame.QUIT:  # If window close button clicked

                    pygame.quit()  # Quit pygame

                    sys.exit()  # Exit program

                if event.type == pygame.KEYDOWN:  # Key pressed down

                    if event.key == pygame.K\_LEFT:  # Left arrow key pressed

                        self.movement[0] = True  # Set left movement flag

                    if event.key == pygame.K\_RIGHT:  # Right arrow key pressed

                        self.movement[1] = True  # Set right movement flag

                    if event.key == pygame.K\_UP:  # Up arrow pressed

                        if self.player.jump():  # Attempt to jump

                            self.sfx['jump'].play()  # Play jump sound

                    if event.key == pygame.K\_x:  # 'x' key pressed

                        self.player.dash()  # Player dash action

                if event.type == pygame.KEYUP:  # Key released

                    if event.key == pygame.K\_LEFT:  # Left arrow released

                        self.movement[0] = False  # Clear left movement flag

                    if event.key == pygame.K\_RIGHT:  # Right arrow released

                        self.movement[1] = False  # Clear right movement flag

            if self.transition:  # If transitioning between levels

                transition\_surf = pygame.Surface(self.display.get\_size())  # Create a surface same size as game display

                pygame.draw.circle(transition\_surf, (255, 255, 255), (self.display.get\_width() // 2, self.display.get\_height() // 2), (30 - abs(self.transition)) \* 8)  # Draw circle to reveal next level

                transition\_surf.set\_colorkey((255, 255, 255))  # Set white as transparent color key

                self.display.blit(transition\_surf, (0, 0))  # Draw transition mask on display

            self.display\_2.blit(self.display, (0, 0))  # Blit the game display surface on top of display\_2

            # Calculate screen shake offset randomly within shake magnitude

            screenshake\_offset = (random.random() \* self.screenshake - self.screenshake / 2, random.random() \* self.screenshake - self.screenshake / 2)

            # Blit the final scaled display\_2 surface to main screen with screenshake offset

            self.screen.blit(pygame.transform.scale(self.display\_2, self.screen.get\_size()), screenshake\_offset)

            pygame.display.update()  # Update the full display Surface to the screen

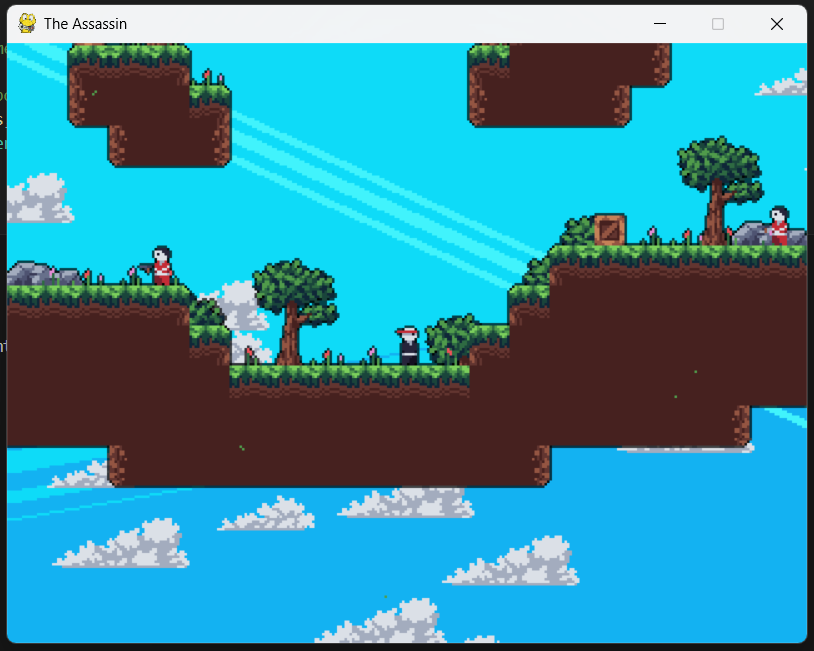
            self.clock.tick(60)  # Keep the game running at 60 frames per second

Game().run()  # Create a Game instance and start running it

**Results**

**Space Invaders : The Assassin :**

A screenshot of a video game

AI-generated content may be incorrect. 

**Project Summary**

This project involves the development of two 2D games, **Space Invaders** and **The Assassin**, using Python and the Pygame library. Both games demonstrate fundamental concepts of game development such as player movement, enemy behavior, collision detection, and event handling. The games are designed to be interactive and engaging, featuring graphical rendering, sound effects, and responsive controls.

The project showcases practical implementation of object-oriented programming and modular design to organize game elements efficiently. Through this project, core skills in game programming, user input management, and multimedia integration were developed. The resulting games serve as examples of classic arcade-style gameplay and modern game mechanics, highlighting the capabilities of Python and Pygame for educational and entertainment purposes.