Plumber Run – Documentation

Computer Architecture and Organization Project by:

- 66010991 Diyaan Pulikkal
- 66011148 Phathompol Siripichaiprom

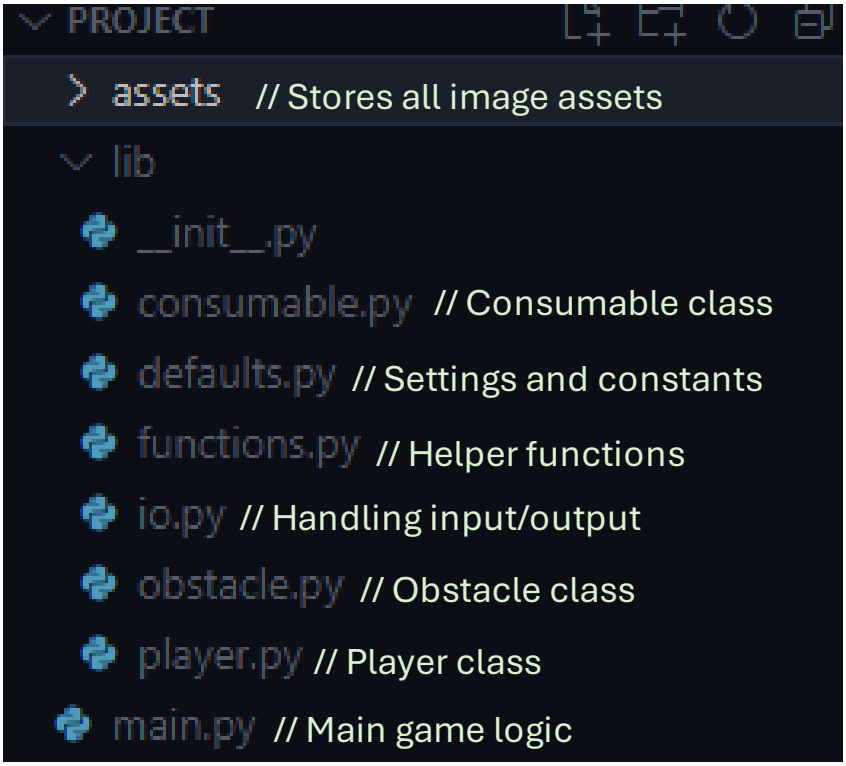
General Explanation

Plumber Run is an arcade-style, 2-player, competitive video game. Players each take control of Mario (red controller) and Luigi (green controller). The players then intuitively tilt their controller to specific directions to run and jump so they can avoid obstacles and pick up consumables. The game ends when a player's hit point reaches 0 due to colliding with obstacles. Finally, the winner is declared on the LEDs and the players can choose to restart by pressing the pulldown button.

Essential Components

The game is developed on Python using the libraries pygame, RPi.GPIO, and mpu6050. Raspberry Pi 400's GPIO and I2C is utilized for input/output.

Project Structure



/lib Directory Explanation

io.py

Setting up the input/output pins of Raspberry Pi 400. `mario` is bound to mpu6050 sensor at the i2c address 0x69 while `luigi` is bound to address 0x68. LEDs and the button are also set up here:

```
import RPi.GPIO as GPIO
from mpu6050 import mpu6050

GPIO.setmode(GPIO.BCM)

GREEN_LED = 21
RED_LED = 20
PULLDOWN_BUTTON = 16
mario = mpu6050(0x69)
luigi = mpu6050(0x68)
GPIO.setwarnings(False)
GPIO.setup(GREEN_LED, GPIO.OUT)
GPIO.setup(RED_LED, GPIO.OUT)
GPIO.setup(PULLDOWN_BUTTON, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

Simple io functions are also defined in this file:

```
# 0 means red, 1 means green
def turn_on_led(color):
    if color == 1:
        GPIO.output(GREEN_LED, GPIO.HIGH)
        return
    GPIO.output(RED_LED, GPIO.HIGH)

def turn_off_led(color):
    if color == 1:
        GPIO.output(GREEN_LED, GPIO.LOW)
        return
    GPIO.output(RED_LED, GPIO.LOW)

def is_button_pressed():
    return GPIO.input(PULLDOWN_BUTTON) == GPIO.HIGH
```

The rest of the file contains functions for retrieving accelerometer sensor data from the mpu6050 for usage in the main game. (a `cleanup()` function is called after a game completion)

```
# 0 means mario, 1 means luigi
def is_jumping(player):
    if player == 1:
        return luigi.get_accel_data()['y'] < 0
    return mario.get_accel_data()['y'] < 0

def is_walking_forward(player):
    if player == 1:
        return luigi.get_accel_data()['x'] < 0
    return mario.get_accel_data()['x'] < 0

def cleanup():
    GPIO.cleanup()
```

defaults.py

This file contains all the constants used for game logic. The constants' names are self-explanatory.

```python
import pygame

# General
WIDTH = 800
HEIGHT = 400
GROUND_HEIGHT = 15
FPS = 60
SCORE_MULTIPLIER = 1
GRAVITY = 1
HITBOX_SIZE = 20  # INCREASE THIS VALUE TO MAKE THE HITBOX SMALLER
BG_ASSET = pygame.image.load("assets/bg.jpg")


# Colors
WHITE = (255, 255, 255)
RED = (188, 24, 35)  # Player 1
BLUE = (0, 74, 173)  # Player 2


# Player
PLAYER1_ASSET = pygame.image.load("assets/redman.png")
PLAYER2_ASSET = pygame.image.load("assets/greenman.png")
PLAYER_SPEED = 5
PLAYER_HEALTH = 10
JUMP_HEIGHT = 20


# Obstacle
BARRIER_ASSET = pygame.image.load("assets/barrier.png")
SPIKETRAP_ASSET = pygame.image.load("assets/spiketrap.png")
THORN_ASSET = pygame.image.load("assets/thorn.png")
OBSTACLE_ASSETS = [BARRIER_ASSET, THORN_ASSET, SPIKETRAP_ASSET]
OBSTACLE_SPEED = 3


# Consumable
HEALTHORB_ASSET = pygame.image.load("assets/health_orb.png")
HEALTHORB_HEAL = 1
GOLDCOIN_ASSET = pygame.image.load("assets/coin.png")
GOLDCOIN_SCORE = 350
CONSUMABLE_ASSETS = [HEALTHORB_ASSET, GOLDCOIN_ASSET]
CONSUMABLE_SPEED = 3
CONSUMABLE_SPAWN_RATE = 300  # every X frames
```

functions.py

This file contains helper functions that are used multiple times in main.py so that main is more abstract and readable. The functions here include:

- `handle_jump(player)` – handles players' jumping logic

```python
def handle_jump(player: Player):
    if player.is_jumping:
        player.y -= player.jump_velocity
        player.jump_velocity -= defaults.GRAVITY
        if player.jump_velocity < -defaults.JUMP_HEIGHT:
            player.is_jumping = False
            player.jump_velocity = defaults.JUMP_HEIGHT
        return

    if player.y < defaults.HEIGHT - defaults.GROUND_HEIGHT - player.get_height():
        player.y += player.jump_velocity
        player.jump_velocity += 1
        return

    player.y = defaults.HEIGHT - defaults.GROUND_HEIGHT - player.get_height()
```

  When a player starts jumping, `jump_velocity` of that player is set to max velocity. When this function is called, the player's y position is subtracted by the velocity to rise them up on the screen's x-y plane. The `jump_velocity` is then subtracted by the gravity value on every loop. When the jump velocity is around 0, the player reaches the peak jumping height and will start to fall because `jump_velocity` will keep getting deducted. When it reaches negative max velocity, it means the player is now somewhere around the ground height. The jump state is then set to `False`, and the player can begin to initiate the next jump. If the player does not jump, this function will set the player to default ground height to ensure they are always at the correct elevation.

- Handling obstacles and consumables going off the screen

```python
def handle_off_screen_obs(obstacle: Obstacle):
    if obstacle.x < -obstacle.get_width():
        obstacle.x = defaults.WIDTH
        obstacle.speed += 0.5
        obstacle.y = defaults.HEIGHT - defaults.GROUND_HEIGHT - obstacle.get_height()
        obstacle.on_screen = False

def handle_off_screen_cons(consumable: Consumable):
    if consumable.x < -consumable.get_width():
        consumable.x = defaults.WIDTH
        consumable.y = defaults.HEIGHT - defaults.GROUND_HEIGHT - consumable.get_height()
        consumable.on_screen = False
```

  The functions ending with obs and cons are for obstacles and consumables, respectively. When an obstacle or a consumable has travelled outside of the screen, they are repositioned back to the rightmost part of the screen and is kept invisible until spawned by setting `on_screen` property to `False`

- This function is simply for creating pygame's `Rect` object around players, obstacles and consumables (which are just rectangular hitbox) for checking if there are any collision.

```
def generate_hitbox(obj):
    return pygame.Rect(obj.x + defaults.HITBOX_SIZE,
                       obj.y + defaults.HITBOX_SIZE,
                       obj.get_width() - defaults.HITBOX_SIZE,
                       obj.get_height() - defaults.HITBOX_SIZE
                       )
```

player.py

```python
import lib.defaults as defaults

class Player():
    def __init__(self, x, y, asset):
        self.x = x  // Stores x, y position, current jump velocity,
        self.y = y  health, score and jump state
        self.asset = asset
        self.jump_velocity = defaults.JUMP_HEIGHT
        self.health = defaults.PLAYER_HEALTH
        self.score = 0
        self.is_jumping = False
    // The methods' names are self-explanatory
    def heal(self, amount):
        self.health += amount

    def take_damage(self, damage):
        self.health -= damage

    def is_dead(self):
        return self.health <= 0

    def increase_score(self, multiplier):
        self.score += multiplier

    def get_width(self):
        return self.asset.get_width()

    def get_height(self):
        return self.asset.get_height()
```

obstacle.py

```python
import random
import lib.defaults as defaults

class Obstacle():
    def __init__(self, x, y, damage, asset):
        self.x = x
        self.y = y
        self.speed = defaults.OBSTACLE_SPEED
        self.damage = damage
        self.asset = asset
        self.spawned = False
        self.on_screen = False

    def random_spawn(self):
        self.spawned = random.randint(0, 1) == 0
        if not self.spawned:
            return
        rand_int = random.randint(0, 2)
        self.asset = defaults.OBSTACLE_ASSETS[rand_int]
        self.damage = rand_int + 1
        self.on_screen = True
        self.spawned = False

    def get_width(self):
        return self.asset.get_width()

    def get_height(self):
        return self.asset.get_height()

    def is_on_screen(self):
        return self.on_screen
```

Stores x,y position, move speed, damage, image asset, spawned state and a variable which indicates if the obstacle should be drawn on screen (on_screen).

random between 0 and 1 to decide if the obstacle should be spawned. If not spawned, then end function. random between 0 to 2 to decide what kind of obstacle it should be and its damage. Set on_screen to true to allow the obstacle to be drawn.

Getters for width, height and on_screen

consumable.py

```python
from lib.player import Player
import lib.defaults as defaults
import random

class Consumable():
    def __init__(self, x, y, type):
        self.x = x
        self.y = y
        self.type = type
        self.asset = defaults.CONSUMABLE_ASSETS[type]
        self.spawned = False
        self.on_screen = False
        self.speed = defaults.CONSUMABLE_SPEED


    def take_effect_on_player(self, player: Player):
        self.on_screen = False
        if self.type == 0:
            player.heal(defaults.HEALTHORB_HEAL)
            return

        player.increase_score(defaults.GOLDCOIN_SCORE)

    def random_spawn(self):
        self.spawned = random.randint(0, 1) == 0
        if not self.spawned:
            return
        self.on_screen = True
        self.spawned = False

    def set_type(self, type):
        self.type = type
        self.asset = defaults.CONSUMABLE_ASSETS[type]

    def is_on_screen(self):
        return self.on_screen

    def get_width(self):
        return self.asset.get_width()

    def get_height(self):
        return self.asset.get_height()
```

Stores x,y position, move speed, damage, image asset, spawned state and a variable which indicates if the consumable should be drawn on screen (on_screen).

Effect that will occur when the consumable hits the plater. If type is 0 then heal the player. If type is 1 then increase the player score.

random between 0 and 1 to decide if the consumable should be spawned. Set on_screen to true to allow the obstacle to be drawn.

Setter for consumable type (0 for Health orbs and 1 for Gold coin)

Getters for width, height and on_screen

Main Game Logic (main.py)

1. All objects are initialized for the main game loop.

```python
import pygame
from lib import defaults, functions, io
from lib.obstacle import Obstacle
from lib.player import Player
from lib.consumable import Consumable
import random
import sys


# initialize pygame engine
pygame.init()


def main():
    # CREATE SCREEN
    screen = pygame.display.set_mode((defaults.WIDTH, defaults.HEIGHT))
    pygame.display.set_caption("Plumber Run")

    background_image = defaults.BG_ASSET

    # player instances
    player1 = Player(defaults.WIDTH // 2, defaults.HEIGHT - defaults.GROUND_HEIGHT -
                     defaults.PLAYER1_ASSET.get_height(), defaults.PLAYER1_ASSET)
    player2 = Player(defaults.WIDTH // 2, defaults.HEIGHT - defaults.GROUND_HEIGHT -
                     defaults.PLAYER2_ASSET.get_height(), defaults.PLAYER2_ASSET)

    # obstacle variables
    obstacle1 = Obstacle(defaults.WIDTH, defaults.HEIGHT - defaults.GROUND_HEIGHT -
                         defaults.BARRIER_ASSET.get_height(), 1, defaults.BARRIER_ASSET)
    obstacle2 = Obstacle(defaults.WIDTH, defaults.HEIGHT - defaults.GROUND_HEIGHT -
                         defaults.BARRIER_ASSET.get_height(), 1, defaults.BARRIER_ASSET)
    obstacle3 = Obstacle(defaults.WIDTH, defaults.HEIGHT - defaults.GROUND_HEIGHT -
                         defaults.BARRIER_ASSET.get_height(), 1, defaults.BARRIER_ASSET)
    obstacles = [obstacle1, obstacle2, obstacle3]

    winner = None

    # consumable variables
    consumable1 = Consumable(defaults.WIDTH, defaults.HEIGHT -
                             defaults.GROUND_HEIGHT - defaults.HEALTHORB_ASSET.get_height(), 1)
```

2. The `running` variable is set to `True` and will be set to `False` when the player quits the game, or the winning condition is met (a player dies). `accumulated_frames` is a variable that has the number of frames added up so far. This will be used to decide when the objects should decide to spawn. For example, one of the obstacles might be set to spawn every 90 frames. The program can check if `accumulated_frames` can be perfectly divided by 90 and run `random_spawn()` if so.

```
    # game loop
    running = True
    clock = pygame.time.Clock()
    accumulated_frames = 0
    while running:
        # check for quit event
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
```

3. Retrieve inputs using functions from io.py. Add/subtract each player's x position by default player speed to move forward or backward. Set `is_jumping` to `True` if the controller is tilted to jumping position. `handle_jump()` in functions.py is also called for each player.

```
    # handle player input (0 is mario and 1 is luigi)
    if not io.is_walking_forward(0) and player1.x > 0:
        player1.x -= defaults.PLAYER_SPEED
    if io.is_walking_forward(0) and player1.x < defaults.WIDTH - player1.get_width():
        player1.x += defaults.PLAYER_SPEED
    if io.is_jumping(0) and not player1.is_jumping:
        player1.is_jumping = True

    if not io.is_walking_forward(1) and player2.x > 0:
        player2.x -= defaults.PLAYER_SPEED
    if io.is_walking_forward(1) and player2.x < defaults.WIDTH - player2.get_width():
        player2.x += defaults.PLAYER_SPEED
    if io.is_jumping(1) and not player2.is_jumping:
        player2.is_jumping = True

    functions.handle_jump(player1)
    functions.handle_jump(player2)
```

4. Randomly spawn obstacles and consumables and update the position of the ones that are already on the screen.

```
        # randomly spawn obstacles
        for i in range(len(obstacles)):
            if accumulated_frames % (i*30 + 90) == 0 and not obstacles[i].on_screen:
                obstacles[i].random_spawn()

        # randomly spawn consumables
        if accumulated_frames % defaults.CONSUMABLE_SPAWN_RATE == 0 and not consumable1.on_screen:
            consumable_type = random.randint(0, 1)
            consumable1.set_type(consumable_type)
            consumable1.random_spawn()

        # update each obstacle's position
        for obstacle in obstacles:
            if obstacle.is_on_screen():
                obstacle.x -= obstacle.speed

        # update consumable position
        if consumable1.is_on_screen():
            consumable1.x -= consumable1.speed
```

5. Create a rectangular hitbox for each of the objects and check if players collide with them. If so, the player either takes damage from the obstacles or receives positive effect from consumables.

```
        # initialize hitboxes
        player1_rect = functions.generate_hitbox(player1)
        player2_rect = functions.generate_hitbox(player2)
        obstacle1_rect = functions.generate_hitbox(obstacle1)
        obstacle2_rect = functions.generate_hitbox(obstacle2)
        obstacle3_rect = functions.generate_hitbox(obstacle3)
        consumable1_rect = functions.generate_hitbox(consumable1)


        # check for collision
        for player in [(player1, player1_rect), (player2, player2_rect)]:
            for obstacle in [(obstacle1, obstacle1_rect), (obstacle2, obstacle2_rect), (obstacle3, obstacle3_rect)]:
                if player[1].colliderect(obstacle[1]):
                    player[0].take_damage(obstacle[0].damage)
                    obstacle[0].x = defaults.WIDTH
            if player[1].colliderect(consumable1_rect):
                consumable1.take_effect_on_player(player[0])
                consumable1.x = defaults.WIDTH
```

6. Check for winning condition. If any player dies, the winner is recorded, and running is set to `False`, so the loop is broken after this iteration. If not, the game continues, and the score is updated.

```
        # check if player is dead
        if player1.is_dead() or player2.is_dead():
            if player1.is_dead():
                winner = 1
            else:
                winner = 0
            running = False

        # update player score
        player1.increase_score(defaults.SCORE_MULTIPLIER)
        player2.increase_score(defaults.SCORE_MULTIPLIER)

```

7. Draw every objects using x,y positions that are calculated earlier and also draw the player score and health texts.

```
    # draw everything
    screen.blit(background_image, (0, 0))
    screen.blit(player1.asset, (player1.x, player1.y))
    screen.blit(player2.asset, (player2.x, player2.y))
    for obstacle in obstacles:
        if obstacle.is_on_screen():
            screen.blit(obstacle.asset, (obstacle.x, obstacle.y))
    if consumable1.is_on_screen():
        screen.blit(consumable1.asset, (consumable1.x, consumable1.y))

    # display player score
    gameOverFont = pygame.font.Font(None, 36)
    player1_score_text = gameOverFont.render(f"Player 1 Score: {str(player1.score)}, Health: {str(player1.health)}", True, defaults.RED)
    player2_score_text = gameOverFont.render(f"Player 2 Score: {str(player2.score)}, Health: {str(player2.health)}", True, defaults.BLUE)
    screen.blit(player1_score_text, (10, 10))
    screen.blit(player2_score_text, (defaults.WIDTH - 390, 10))
    pygame.display.flip()
```

8. Check for any object that has gone of the screen and handle it. Then, increment `accumulated_frame` and reset it to 0 if its value is unneccessarily large. The loop begins again if the value of running is `True` and step 2-8 is repeated.

```
138         # check if objects has gone off screen
139         for obstacle in [obstacle1, obstacle2, obstacle3]:
140             functions.handle_off_screen_obs(obstacle)
141         functions.handle_off_screen_cons(consumable1)
142 
143         # count frames to decide when to spawn obstacles
144         accumulated_frames += 1
145         if accumulated_frames % 3000 == 0 and accumulated_frames != 0:
146             accumulated_frames = 0
147         clock.tick(defaults.FPS)
```

9. If the loop is broken, “Game Over” is drawn and the game prompts user to press the button to restart. Winner is returned.

```
149     # create game over screen
150     gameOverFont = pygame.font.Font(None, 72)
151     game_over_text = gameOverFont.render("Game Over!", True, (128, 0, 0))
152     press_any_key_text = pygame.font.Font(None, 36)
153     press_any_key_text = press_any_key_text.render(
154         "Press button to restart", True, (0, 0, 0))
155     screen.blit(game_over_text, (defaults.WIDTH //
156                 2 - 150, defaults.HEIGHT // 2 - 50))
157     screen.blit(press_any_key_text, (defaults.WIDTH //
158                 2 - 150, defaults.HEIGHT // 2 + 50))
159 
160     pygame.display.flip()
161     return winner
```

10. When main.py executes, main() is run and its return value is kept in winner. Depending on the value of winner, an appropriate LED will turn on (red LED means Mario won and green LED means Luigi won). The program will then wait for player to press the pulldown button. When the pulldown button is pressed, the LEDs are turned off and main() is called again. If the player decides to quit, both LEDs are turned off and the program exits.

```
163 if __name__ == "__main__":
164     while True:
165         pygame.time.delay(10)
166         winner = main()
167         if winner == 0:
168             io.turn_on_led(0)
169         else:
170             io.turn_on_led(1)
171         waiting_for_keydown = True
172         while waiting_for_keydown:
173             if io.is_button_pressed():
174                 io.turn_off_led(0)
175                 io.turn_off_led(1)
176                 waiting_for_keydown = False
177                 pygame.time.delay(800)
178             for event in pygame.event.get():
179                 if event.type == pygame.QUIT:
180                     io.turn_off_led(0)
181                     io.turn_off_led(1)
182                     pygame.quit()
183                     io.cleanup()
184                     sys.exit()
185                     break
```

End of Documentation