Innovative & Immersive Fitness Experience

Source Code Documentation

Created For

Sense | Think | Act

Company's Contact: Farjana Salahudin | Khalid Al Ali

Advising Professor: Selma Limam Mansar

Course: Information System Consulting Project | Spring 2023

Team Members: Ahmed Issaoui | Jewel Dsouza | Mariam Al Thani | Raghad Abunabaa

Date: Thursday 13th April 2023

Carnegie Mellon University Qatar

STA Sense | Think | Act

Source Code Documentation

Purpose

The purpose of this document is to document and explain all the code about the immersive fitness experience created for STA. From this document, one can learn what the code does, how to install, configure, and maintain it, opportunities for updating and future development, and the testing conducted to ensure a working, functional, and smooth experience. In addition to the aforementioned information, the author's (initial developers) information is provided at the end, and the Python file contains consistent comments throughout, describing and explaining the purpose of code snippets and their relation to the experiences' functionality and presentation.

Description of code's purpose and functionality

The code's purpose and functionality is to launch an endless runner game, in which the user can manipulate the character's actions through physical gestures.

Softwares

The code behind the experience (excluding the environment video background) is created solely on Python 3. It has been developed on multiple versions of Python 3 including 3.1, 3.7, 3.9, and are functional on all three versions. Visual Studio Code was used to develop the python files.

Python Libraries Required

The following libraries (and sub-libraries) were utilized during the development of the final code:

Pygame - 2.1.3

Pygame is the base of the code and is used for game mechanics, obstacle creation, collision detection, font presentation, and more.

OpenCV - 4.7.0.68

OpenCV is a computer vision library that specializes in real-time computer vision. It is used in combination with other libraries to capture the background video and connect with the camera.

Numpy - 1.24.1

Numpy is a mathematical-based library and is utilized in array usage and data transformation of the motion-detected landmarks to a format that is accepted by pygame.

Mediapipe - 0.9.1.0

Mediapipe is an open-source framework that uses machine learning to read data from a video or audio source and detect poses from it. It is utilized in detecting the user, their motion, creating the landmarks, the character, and more.

Pandas - 1.5.3

Pandas is a python library that is focused on data manipulation and analysis. It it utilized in updating the leaderboard file and identifying the top 10 scores.

Instructions on how to install the required libraries.

To install the required packages, navigate to the directory containing the requirements.txt file in your terminal by using the "cd" command. Once you're in the directory, execute the following command: "pip install -r requirements.txt"

Code Breakdown

I. Motion detection

```
# Initialize Mediapipe Pose Detection
mp_pose = mp.solutions.pose

# Start video capture
cap = cv2.VideoCapture(0)

def checkHandsJoined(landmarks):
    # Get the left wrist landmark x and y coordinates.
    left_wrist_landmark = (landmarks.landmark[mp_pose.PoseLandmark.LEFT_WRIST].x * WIDTH,
                           landmarks.landmark[mp_pose.PoseLandmark.LEFT_WRIST].y * HEIGHT)

    # Get the right wrist landmark x and y coordinates.
    right_wrist_landmark = (landmarks.landmark[mp_pose.PoseLandmark.RIGHT_WRIST].x * WIDTH,
                            landmarks.landmark[mp_pose.PoseLandmark.RIGHT_WRIST].y * HEIGHT)

    # Calculate the euclidean distance between the left and right wrist.
    euclidean_distance = int(math.hypot(left_wrist_landmark[0] - right_wrist_landmark[0],
                                        left_wrist_landmark[1] - right_wrist_landmark[1]))
    # Compare the distance between the wrists with a appropriate threshold to check if both hands are joined
    if euclidean_distance < 130:
        return True
    return False
```

This Python script uses the Mediapipe and OpenCV libraries to detect the pose of a person in a video capture and determine if their hands are joined or not. First, the script initializes the Mediapipe Pose Detection model and starts capturing video from the default camera. Then, it defines a function called checkHandsJoined that takes in the pose landmarks detected by the Mediapipe model as a parameter. Within this function, the x and y coordinates of the left and right wrist landmarks are retrieved, and their euclidean distance is calculated. If the distance between the wrists is less than a certain threshold value of 130, it is considered that the hands are joined, and the function returns True. Otherwise, it returns False. This function can be used in a larger

application to detect and track the pose of a person in real-time video and determine if their hands are joined or not, for example, as a gesture control for a computer or a game.

II. Player class

```
class Player():
    def __init__(self):
        self.lives = 3
        self.landmarks = None
    def update(self,landmarks):
        self.landmarks = Player.pygame_landmarks(landmarks)
    def pygame_landmarks(landmarks):
        # If landmarks are detected, draw them on frame
        if landmarks is not None:
            old = landmarks
            # Scale landmarks to make character smaller

            landmarks = np.array([(landmark.x, landmark.y, landmark.z) for landmark in landmarks.landmark])
            landmarks = landmarks[:, :2]

            factor = 0.4
            center_x = screen.get_width() / 2
            center_y = screen.get_height() * (1.2-factor)

            # Map landmarks to Pygame coordinates
            x = landmarks[:, 0]
            y = landmarks[:, 1]
            x = np.interp(x, (0, 1), (0, screen.get_width()))
            y = np.interp(y, (0, 1), (0, screen.get_height()))
            dist_x = x - center_x
            dist_y = y - center_y
            # Apply the dilation factor to the distances
            dist_x *= factor
            dist_y *= factor
            # Add the dilated distances to the center of dilation coordinates
            x = dist_x + center_x
            y = dist_y + center_y
            x = [i for i in x]
            y = [i for i in y]
            landmarks_pygame = list(zip(x, y))

            # Add points to fill the character more
            left_shoulder = landmarks_pygame[12]
            left_elbow = landmarks_pygame[14]
            left_arm = interpolate_points(left_shoulder,left_elbow, int(10*factor))
            left_hip = landmarks_pygame[24]
            left_body = interpolate_points(left_shoulder,left_hip, int(10*factor))
            left_knee = landmarks_pygame[26]
            left_thigh = interpolate_points(left_knee,left_hip, int(10*factor))
            landmarks_pygame += left_arm + left_body + left_thigh
            right_shoulder = landmarks_pygame[11]
            right_elbow = landmarks_pygame[13]
            right_arm = interpolate_points(right_shoulder, right_elbow, int(10*factor))
            right_hip = landmarks_pygame[23]
            right_body = interpolate_points(right_shoulder, right_hip, int(10*factor))
            right_knee = landmarks_pygame[25]
            right_thigh = interpolate_points(right_knee, right_hip, int(10*factor))
            landmarks_pygame += right_arm + right_body + right_thigh
            return landmarks_pygame
        return None
```

This is a Python class called Player that is used to represent a character in a game. The class has two methods: __init__ and update. In the __init__ method, the Player object is initialized with three lives and a landmarks variable that is set to None. In the update method, the landmarks variable is updated with the new pose landmarks detected by the Mediapipe model passed in as an argument. This method calls the pygame_landmarks method to convert the Mediapipe landmarks to Pygame coordinates and scales the character based on the screen size. The pygame_landmarks method takes in the Mediapipe landmarks as an argument and converts them to Pygame

coordinates, which are used to draw the character on the screen. If no landmarks are detected, the method returns None. If landmarks are detected, the landmarks are scaled to make the character smaller and mapped to Pygame coordinates. Then, the distance of each landmark from a center point is calculated and multiplied by a scaling factor to dilate the distance. The dilated distances are added to the center point coordinates to get the new coordinates of each landmark. Additional points are added to fill the character more by interpolating points between some landmarks. The interpolated points are added to the list of landmarks, and the final Pygame coordinates of all the landmarks are returned. Overall, this code is used to create a Player object in a game and update its position based on the pose landmarks detected by the Mediapipe model. It also scales the character based on the screen size and adds additional points to make the character appear more filled.

III. Obstacle class

```
# define the Obstacle class
class Obstacle(pygame.sprite.Sprite):
    s_line = 34 / 2
    b_line = 752 / 2
    distance = 370
    side = (distance) / (1-(s_line/b_line))
    speed = 5
    def __init__(self, position, style):
        super().__init__()
        self.style = style
        self.position = position
        self.color = self.get_color()
        self.height = self.get_height()
        self.width = self.get_width()
        self.y = self.get_position_y()
        self.x = self.get_position_x()

    def get_colision_line(self,nose_y,ankle_y):
        if self.style == "crouch":
            return ankle_y - 20
        elif self.style == "jump":
            return nose_y
        else:
            return (nose_y + ankle_y) / 2

    def get_color(self):
        if self.style == "stand":
            return BLUE
        elif self.style == "crouch":
            return BLACK
        else:
            return GREEN
    def get_height(self):
        initial_size = HEIGHT * 0.01
        if self.style == "stand":
            return initial_size
        else:  # Crouching or Jumping
            return initial_size / 4

    def get_width(self):
        if self.style == "stand":
            return self.height / 2
        else:  # Crouching or Jumping
            return self.height * 14

    def get_position_y(self):
        self.start_point = HEIGHT - Obstacle.distance
        if  self.style == "jump":
            self.start_point =  HEIGHT / 2 - 100
            return HEIGHT / 2 - 100
        return self.start_point
```

```
def get_position_x(self):
    center_x = WIDTH / 2 - self.width / 2
    if self.style == "stand":
        if self.position == "left":
            return center_x - self.width * 3.5
        elif self.position == "right":
            return center_x + self.width * 3.5
        else:  # center
            return center_x
    else:
        return center_x

def update(self):
    # start_point = HEIGHT - Obstacle.distance - self.height
    # move the obstacle down the screen
    self.y += Obstacle.speed + self.height / 10
    d = (self.y - self.start_point) + (Obstacle.side - Obstacle.distance)
    rate = (size(Obstacle.side,Obstacle.b_line,d)) / (Obstacle.s_line)
    self.height = self.get_height() * rate
    self.width = self.get_width()
    self.x = self.get_position_x()

def get_rect(self):
    return pygame.Rect(self.x, self.y, self.width, self.height)
```

This code defines a class called Obstacle that inherits from the Pygame Sprite class. It represents obstacles that the player must avoid in the game. The class has several class-level variables:

- **s_line, b_line**: these are values used to calculate the side distance of the obstacle
- **distance**: the distance of the obstacle from the player.
- **side**: the calculated side distance.
- **speed**: the speed at which the obstacle moves.

The class has an init method that initializes various instance-level attributes of the object such as its style (stand, crouch, or jump), position (left, center, or right), color, height, width, and x, y position. The position is calculated based on the style and position of the obstacle. The height and width of the obstacle are determined based on its style. The class also has several methods:

- **get_colision_line**: returns the collision line of the obstacle based on the player's nose and ankle positions.
- **get_color**: returns the color of the obstacle based on its style.
- **get_height**: returns the height of the obstacle based on its style.

- **get_width**: returns the width of the obstacle based on its style.
- **get_position_y**: returns the y position of the obstacle based on its style and position. **get_position_x**: returns the x position of the obstacle based on its style and position.

IV. Tutorial / Warm-Up Session

The tutorial/warm-up session aspect of the Python script is achieved by creating variables and points that track the state of the experience and the tutorial.

```
tutorial_completed = False

tutorial_hands_joined = False

tutorial_point = "not started"



positions_moved = []



obstacles_avoided = 0
```

This is because the player is prompted to join their hands at three points in the experience - initially, to start the tutorial, secondly, to start the game, upon completing the tutorial, and finally, if they choose to restart the game, after losing. So, these variables help keep track.

The first screen that is shown to the user informs them the need to join their hands to start the experience.

```
if tutorial_hands_joined == False:

    welcome = font.render("welcome...", True, WHITE)

    text = font.render("join your hands to start", True, WHITE)

    screen.blit(welcome, (WIDTH / 2 - welcome.get_rect().width / 2, HEIGHT / 4))

    screen.blit(text, (WIDTH / 2 - text.get_rect().width / 2, HEIGHT / 2))

    player.draw()
```

If the code detects landmarks on the user, it checks the following.

```
if checkHandsJoined(frame, results) == 'Hands Joined':



        #if hands joined for the first time or after reset, set tutorial hands completed to actually start tutorial and move from welcome screen

        if tutorial_hands_joined == False:

            tutorial_hands_joined = True

            #change tutorial point to first point, the motion detection highlight

            tutorial_point = "motion detection highlight"
```

As mentioned, it changes the tutorial state and changes the tutorial point to the first part of the tutorial - the motion detection highlight. After this, the code is structured consistently in that, an if statement is set for each point in the tutorial, and different conditions are within it. After meeting the conditions specific to that tutorial point, the variable is changed and the player and the code move to the next stage or point in the tutorial.

```
# All the different points and screens of the tutorial, past welcome screen
if tutorial_completed == False:
        if tutorial_point=="motion detection highlight":
            #encourage user to move left and right and move arms around
            tutorial_text = font.render("highlight this experience is based on motion detection ", True, WHITE)
            screen.blit(tutorial_text, (WIDTH / 2 - tutorial_text.get_rect().width / 2, HEIGHT / 4))

            #check that they moved around screen
            if player.current_position not in positions_moved:
                positions_moved.append(player.current_position)

            #once done, change tutorial point
            if len(positions_moved) ==3:
                tutorial_point = "obstacles highlight"

        if tutorial_point == "obstacles highlight":
            tutorial_text = font.render("showing obstacles coming towards them", True, GREEN)
            screen.blit(tutorial_text, (WIDTH / 2 - tutorial_text.get_rect().width / 2, HEIGHT / 4))
            for obstacle in obstacles:
                if player.get_rect().colliderect(obstacle.get_rect()):
                    # the player has collided with an obstacle, so notify them
                    obstacles.empty()

                if obstacle.y > HEIGHT-30:
                    if not player.get_rect().colliderect(obstacle.get_rect()):
                        # print(obstacles_avoided)
                        obstacles_avoided+=1

                if obstacles_avoided == 3:
                    tutorial_completed = True
                    #move to next point in tutorial

            # add obstacles
            current_time = pygame.time.get_ticks()
            # multiply last spawn time by 1.5 to slow the obstacles down for the tutorial
            if current_time - last_spawn_time*1.5 > spawn_time:
                last_spawn_time = current_time
                position = random.choice(["center","left","right"])
                style = "stand"
                obstacle = Obstacle(position,style)
                obstacles.add(obstacle)
                print("adding obstacles")

            # remove obstacles that have gone off the bottom of the screen
            for obstacle in obstacles:
                if obstacle.y > HEIGHT:
                    obstacles.remove(obstacle)

        for obstacle in obstacles:
            pygame.draw.rect(screen, obstacle.color, [obstacle.x, obstacle.y, obstacle.width, obstacle.height])

        if tutorial_point == "movement for obstacles highlight":
            print(0)
        if tutorial_point == "different obstacle types highilght":
            print("tutorial completed, go to start screen")
```

Alterations and developments will be made to the code upon completion of the tutorial mechanisms. It remains true that, upon completion of the final point in the tutorial, the variable will be set to true.

When the tutorial is completed, the player will once again be prompted to join their hands and, in doing so, start the actual game.

```
if tutorial_completed == True:

    if game_started == False:

        game_join = font.render("join your hands to start", True, GREEN)

        screen.blit(game_join, (WIDTH / 2 - game_join.get_rect().width / 2, HEIGHT / 4))
```

```
        if results.pose_landmarks:

            if checkHandsJoined(frame,results) == "Hands Joined":

                game_started = True

    player.draw()
```

V. Game loop

```
while True:
    # Read frame from camera
    ret, frame = cap.read()
    frame = cv2.flip(frame, 1)

    # Convert frame to RGB
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    success, video_image = video.read()
    if success:
        video_surf = pygame.image.frombuffer(video_image.tobytes(), video_image.shape[1::-1], "BGR")
    screen.blit(video_surf, (0, 0))
    # Detect pose landmarks
    with mp_pose.Pose(min_detection_confidence=0.5, min_tracking_confidence=0.5) as pose:
        results = pose.process(rgb)
        landmarks = results.pose_landmarks
        player.update(landmarks)
        obstacles.update()

        # Draw the character through a combination of lines and shapes
        if player.landmarks:
            x1, y1 = player.landmarks[0]
            x2, y2 = player.landmarks[8]
            r = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
            color = (255, 0, 0)
            line_width = 15

            pygame.draw.polygon(screen, color, [player.landmarks[11],player.landmarks[12],player.landmarks[24],player.landmarks[23]])
            pygame.draw.circle(screen, color, player.landmarks[0], r * 1.3, width=0)
            pygame.draw.line(screen, color, player.landmarks[12], player.landmarks[14], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[14], player.landmarks[16], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[16], player.landmarks[22], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[16], player.landmarks[18], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[16], player.landmarks[20], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[18], player.landmarks[20], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[12], player.landmarks[24], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[24], player.landmarks[26], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[26], player.landmarks[28], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[28], player.landmarks[30], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[28], player.landmarks[32], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[30], player.landmarks[32], width=line_width)

            pygame.draw.line(screen, color, player.landmarks[11], player.landmarks[13], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[13], player.landmarks[15], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[15], player.landmarks[21], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[15], player.landmarks[17], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[15], player.landmarks[19], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[17], player.landmarks[19], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[11], player.landmarks[23], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[23], player.landmarks[25], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[25], player.landmarks[27], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[27], player.landmarks[29], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[27], player.landmarks[31], width=line_width)
            pygame.draw.line(screen, color, player.landmarks[29], player.landmarks[31], width=line_width)
```

```
if game_started:
    if len(obstacles) == 0:
        addObstacle()
    for obstacle in obstacles: # this loop is used to store the obstacle in a variable
        pass
    # seting up the colison zone
    y1 = y2 = obstacle.get_colision_line(nose_y,ankle_y)
    y3 = y4 = y1 + 30
    d1 = (y1 - HEIGHT + Obstacle.distance) + (Obstacle.side - Obstacle.distance)
    d2 = (y3 - HEIGHT + Obstacle.distance) + (Obstacle.side - Obstacle.distance)
    w1 = (d1 * Obstacle.b_line) / Obstacle.side
    w2 = (d2 * Obstacle.b_line) / Obstacle.side
    w1 *= 2
    w2 *= 2
    x1 = WIDTH / 2 - w1 / 2
    x2 = WIDTH / 2 + w1 / 2
    x3 = WIDTH / 2 - w2 / 2
    x4 = WIDTH / 2 + w2 / 2
    vertices = [(x1, y1), (x2, y2), (x4, y4), (x3, y3)]
    # colision handling
    if player.landmarks:
        if obstacle.y > y1 and obstacle.y < y3:
            for x, y in player.landmarks:
                temp_rect = pygame.Rect(x, y, 1, 1)
                if temp_rect.colliderect(obstacle.get_rect()):
                        # the player has collided with an obstacle, so lose a life
                        if player.lives !=0:
                            player.lives -= 1
                        elif player.lives == 0:
                            # player has no lives left
                            # Create randomly generated username for player
                            # generating random strings
                            if username ==None:
                                username = ''.join(random.choices(string.ascii_uppercase, k=5))
                                user_score = [username,int(score)]
                                #!! in the end, this needs to be modified for the leaderboard to store the top 10
                                with open('leaderboard.csv', 'a+', newline='\n') as write_obj:
                                    # Create a writer object from csv module
                                    csv_writer = writer(write_obj)
                                    # Add contents of list as last row in the csv file
                                    csv_writer.writerow(user_score)
                            #end the game
                            # to-do
                        obstacles.empty()
                        break

    # remove obstacles that have gone off the bottom of the screen and add new obstacles
    for obstacle in obstacles:
        if obstacle.y > HEIGHT:
            obstacles.remove(obstacle)
            addObstacle()
            score += 5

    score += 0.1
```

```
      # drawing the collision zone
       pygame.draw.polygon(screen, BLUE, vertices)

       # drawing the obstacles
       for obstacle in obstacles:
           pygame.draw.rect(screen, obstacle.color, [obstacle.x, obstacle.y, obstacle.width, obstacle.height])

   else:
       if landmarks:
           game_started = checkHandsJoined(landmarks)
           nose_y, ankle_y = player.landmarks[0][1], player.landmarks[28][1]
       text = font.render("join your hands to start", True, WHITE)
       screen.blit(text, (WIDTH / 2 - text.get_rect().width / 2, HEIGHT / 4))


   screen.blit(font.render(f"score: {int(score)}", True, WHITE), (10, 10))
   screen.blit(font.render(f"lives: {player.lives}", True, WHITE), (WIDTH - 150, 10))

   # Display frame in Pygame window
   pygame.display.update()
   clock.tick(160)

   # Handle Pygame events
   for event in pygame.event.get():
       if event.type == pygame.QUIT:
           cap.release()
           cv2.destroyAllWindows()
           pygame.quit()
           exit(0)
```

The main loop of the program is initiated with a while statement that continues to run indefinitely until manually stopped. Within this loop, the program first reads a frame from a camera connected to the computer, flips the image horizontally, and then converts the frame from the default BGR format to RGB format.

Next, the program attempts to read a video stream and if successful, converts the stream to an image that can be displayed on the screen using the Pygame library. The video image is then displayed on the screen.

The next section of the program uses the MediaPipe Pose library to detect pose landmarks from the converted RGB image. These landmarks are stored in a landmarks variable. The program then updates the position of the player's character based on the landmarks detected. The character is drawn on the screen using Pygame drawing functions, which are called in a specific order to create a polygonal shape that represents the character's body.

After drawing the character on the screen, the program checks if the game has started and if there are any obstacles present in the game. If there are no obstacles, a new obstacle is added to the game. The program then creates a collision zone around the obstacle, with four vertices that are defined based on the position of the obstacle and the dimensions of the collision zone. Finally, the program handles collisions between the player's character and any obstacles present in the game.

VI. Game restarting

The restart functionality is embedded within the main game loop and starts with declaring a number variable called countdown. This variable alongside the timer serve as how long the countdown is.

```
countdown, countdown_seconds = 15, '15'

pygame.time.set_timer(pygame.USEREVENT, 1500)
```

The idea is that, when a player loses all their lives, and subsequently the game, the screen will show two things 1) a countdown timer which allows the player to restart the game if they join their hands in time and 2) the leaderboard with the username and scores which we discuss below.

Restarting the game means that the player retains their username and does not have to undergo the tutorial/warm-up session again. The countdown works as follows:

```
for event in pygame.event.get():

    # countdown for restarting
```

```
    if player.lives == 0:
        if event.type == pygame.USEREVENT:
            countdown -= 1
            if countdown > 0:
                countdown_seconds = str(countdown)
```

In the code above, in every instance, the if statement checks if the player has lost all their lives and if the event is part of the timer. If both of those statements are true, the countdown gets reduced and the countdown seconds, its string equivalent, are reduced by one, as long as the countdown is not zero.

Later on in the code, the countdown is presented on the screen when the player loses all their lives through the below mechanism.

```
  if player.lives == 0:
    #player lost, show leaderboard
    obstacles.empty()
    #restart option - 15 seconds
    if countdown > 0:
        game_over = font.render(f"game over.. join hands in {countdown_seconds} seconds to restart", True, GREEN)
        screen.blit(game_over, (WIDTH / 2 - game_over.get_rect().width / 2, HEIGHT / 4))
```

Here, as the text prompts them to join their hands to restart, we check if they did so, before the countdown reaches zero.

```
#Player chooses to restart experience
        if checkHandsJoined(frame,results) == "Hands Joined":
            player.lives+=3
            score = 0
```

```
#Player has not restarted, reset experience
    if countdown == 0:
        # if checkHandsJoined(frame,results) == "Hands Not Joined":
            #reset the players attributes
            player.username =  ''.join(random.choices(string.ascii_uppercase, k=5))
            player.lives +=3
            score = 0

            #reset the tutorial attributes
            tutorial_completed = False
            tutorial_hands_joined = False
            tutorial_point = "not started"
            game_started = False
            reset = True
            positions_moved = []
            obstacles_avoided = 0

            #reset countdown
            countdown, countdown_seconds = 15, '15'
            pygame.time.set_timer(pygame.USEREVENT, 1500)
```

As shown above and in the comments, if the player does not join their hands and the countdown reaches 0, the experience resets and all the relevant variables related to the restart, tutorial, and game mechanisms are reset to their initial state.

VII. Data Storage & Leaderboard

Data generated from the experience is stored in a CSV file in the directory that is constantly updated. This is the leaderboard of the top 10 scores and usernames that are presented at the end of each session where a player loses all their lives. The username is stored as a variable both in the Player class and outside it to account for cases where a user restarts the experience, allowing them to retain the same username they had the first time. This is because usernames are randomly generated.

```
if game_started & tutorial_completed:
    if player.lives !=0:
        screen.blit(font.render(f"score: {int(score)}", True, WHITE), (10, 10))
        screen.blit(font.render(f"lives: {player.lives}", True, WHITE), (WIDTH - 150, 10))
    player.draw()
    for obstacle in obstacles:
        if player.get_rect().colliderect(obstacle.get_rect()):
            # the player has collided with an obstacle, so lose a life
            if player.lives !=0:
                player.lives -= 1
            if player.lives == 0:
                # player has no lives left
                # Create randomly generated username for player
                # generating random strings
                username = player.username
                user_score = [username,int(score)]

                with open('leaderboard.csv', 'a+', newline='\n') as write_obj:
                    # Create a writer object from csv module
                    csv_writer = writer(write_obj)
                    # Add contents of list as last row in the csv file
                    csv_writer.writerow(user_score)
                # Add the new row to the dataframe
                data.loc[len(data)] = user_score
                #end the game
                break
            obstacles.empty()
```

The code above checks if and when the players lives reach 0, it stores their username and score. The randomly generated string for the username is created within the Player class. Next, using the writer module, we add a new line to the leaderboard CSV file with the users name and score.

As this is a leaderboard, the csv file should only contain the top 10 rows. So, the rest of the leaderboard updating mechanism is conducted when the player reaches 0 lives and in the same case where the countdown screen is presented, as mentioned above.

```
if countdown > 0:
    game_over = font.render(f"game over.. join hands in {countdown_seconds} seconds to restart", True, GREEN)
    screen.blit(game_over, (WIDTH / 2 - game_over.get_rect().width / 2, HEIGHT / 4))
        #order dataset based on highest scores
    data = data.sort_values(by=['Score'], ascending=False)
    # get the top 10 scores
    data = data.head(10)
    usernames = data.loc[:,"Username"]
    #offset so the rows show one after the other
    offset=0
    for row in usernames:
        score = data.loc[data['Username'] == row]['Score'].values[0]
        #Highlight the users score if they make it on the leaderboard
        if row == username:
            user_row = font.render(f"{row.lower()}.................{str(score)}", True, GREEN)
            screen.blit(user_row, (WIDTH / 2 - user_row.get_rect().width / 2, (HEIGHT / 3)+offset))
        else:
            leader_row = font.render(f"{row.lower()}.................{str(score)}", True, WHITE)
            screen.blit(leader_row, (WIDTH / 2 - leader_row.get_rect().width / 2, (HEIGHT / 3)+offset))
        offset+=30
    #save the leaderboard to the updated csv so there will always be only 10 rows stored
    csv_save = data
    csv_save.to_csv("leaderboard.csv", index=False)
```

Here, we first sort the csv file by the highest scores, and limit it to the top 10 scores by getting the head rows. Additionally, to show the leaderboard appropriately, we instantiate an offset for the presentation and store all the usernames to loop through.

In the loop, we check if the username we are on is the same as the user's username. This means that the user reached a high enough score to make it to the leaderboard, To distinguish this from the others, we present the text in green rather than white.

After the updating and presentation of the leaderboard, we save the CSV file as data - the variable that contains the sorted top 10 scores and usernames of all time. This ensures that we will always only have 10 usernames and scores in the CSV file.

Testing

All the testing for the code was constantly conducted manually throughout the development of the code both by the developers and through usability testing with various groups of users and scenarios.

Bugs

Currently, there are no bugs in the code that we are aware of but will update this section as we move forward with completing the project, conducting further testing, and finalizing the code.

Author Information

Ahmed Issaoui

E-mail: aissaoui@andrew.cmu.edu

Github: https://github.com/aissaoui-cmu

Mariam Al Thani

E-mail: mathani@andrew.cmu.edu

Github: https://github.com/mariamalth

Jewel Dsouza

E-mail: jdsouza@andrew.cmu.edu

Github: https://github.com/jeweldsouza

Raghad Abunabaa

E-mail: rmabunab@andrew.cmu.edu

Github: https://github.com/raghadabunabaa