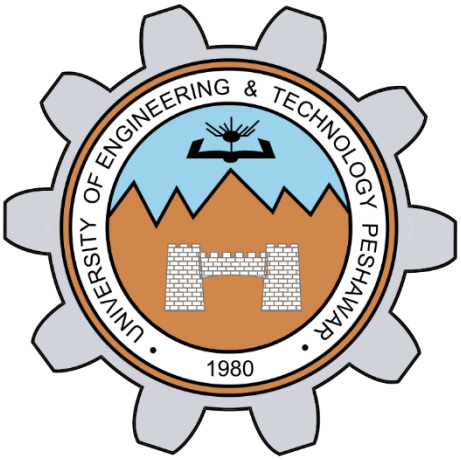
OBJECT ORIENTED PROGRAMMING LAB

FLAPPY BIRD GAME IN PYGME

PROJECT REPORT



FALL 2024

CSE-208L: OBJECT ORIENTED PROGRAMMING LAB

SUBMITTED BY: RAHMA TAIEB

MARWA RAFIQUE

FATIMA INNAYAT

REGISTERATION NO: 23pwcse2291

23pwcse2304

23pwcse2266

CLASS SECTION: A

"On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work."

Student Signature:

SUBMITTED TO: Engr. Sumayyea Salahuddin

FEBRUARY 2ND ,2025

DEPARTMENT OF COMPUTER SYSTEMS ENGINEERING

UNIVERSITY OF ENGINEERING AND TECHNOLOGY, PESHAWAR

FLAPPY BIRD GAME IN PYGAME

1. INTRODUCTION:

The objective of this project was to develop an arcade-style game inspired by the widely recognized Flappy Bird, utilizing the Pygame library within the Python programming environment. The game serves as an application of fundamental programming concepts and game development techniques.

2. BACKGROUND:

Flappy Bird is a minimalist game characterized by its simple mechanics and challenging gameplay. The core gameplay involves navigating a bird through a series of obstacles (pipes) by controlling its vertical movement, while gravity constantly influences its trajectory. This project leverages these mechanics to create an interactive application that demonstrates proficiency in real-time graphics rendering, user input handling, and collision detection.

3. OBJECTIVES:

The primary objectives of the project include:

- **Game Mechanics Implementation:** Developing realistic physics for the bird's motion, including gravity and thrust.
- **Obstacle Generation:** Creating a dynamic and challenging environment by generating obstacles at regular intervals.
- **Collision Detection:** Implementing reliable collision detection between the bird and obstacles to determine game state transitions.
- **User Interface:** Designing an intuitive interface that manages game states such as start, gameplay, and game-over screens.

4. METHODOLOGY:

The development process was carried out using the following approach:

- **Library Utilization:** The Pygame library was employed to handle graphics rendering, event management, and real-time updates.
- **Modular Design:** The project was structured into distinct modules for handling different aspects of the game, such as input handling, physics simulation, and obstacle management.
- **Iterative Testing:** Continuous testing was conducted throughout development to ensure each game component functioned correctly and to maintain an engaging user experience.

4. SIGNIFICANCE:

This project not only illustrates the practical application of programming principles but also reinforces problem-solving skills in the context of

interactive game development. The structured approach to implementing and testing the game components provides valuable insights into the challenges and best practices associated with developing real-time applications.

This structured introduction should fit well within a formal lab report, providing clear context and a detailed overview of the project without using informal language.

5. OBJECT-ORIENTED PROGRAMMING CONCEPTS USED IN THE FLAPPY BIRD GAME:

5.1. INTRODUCTION:

The flappy bird game is structured using several OOP principles, which help in organizing code, promoting reusability, and improving maintainability.

5.2. CLASSES AND OBJECTS:

5.2.1. DEFINITION AND ROLE:

- **Classes:** Serve as blueprints for creating objects. In your project, classes are used to represent different entities or components of the game.
- **Objects:** Instances of these classes that hold state (data) and behavior (methods).

5.2.2. CODE:

5.2.2.1. GAME CLASS:

Manages the overall game flow, user input, screen rendering, collision checking, and level selection.

```
class Game:
    def __init__(self):
        # Setting window config
        self.width = 600
        self.height = 768
        self.scale_factor
        self.win = pg.disp                    de((self.width, self.height))
        self.clock = pg.time.Clock()
        self.move_speed = 250
        self.start_monitoring = False
```

Loading...

5.2.2.2. BIRD CLASS:

Represents the player-controlled bird, encapsulating its image, position, velocity, and movement behavior.

```python
import pygame as pg

class Bird:
    def __init__(self, scale_factor):
        self.scale_factor = scale_factor
        self.image = pg.image.load("assests/birdup.png").convert_alpha()
        self.image = pg.image.load("assests/birddown.png").convert_alpha
        self.image = pg.transform.scale(self.image, (int(34 * self.scale_
        self.rect = self.image.get_rect(center=(100, 300))
        self.velocity = 0
        self.update_on = False  # Initially, the bird doesn't move
        self.flap_strength = -8  # Default value, changes in setGameDiff
```

5.2.2.3. PIPE CLASS:

Represents the obstacles in the game. Each pipe object maintains its own position and image data.

```python
import pygame as pg
from random import randint

class Pipe:
    def __init__(self, scale_factor, move_speed, pipe_gap):  # Accept pi
        self.img_up = pg.transform.scale_by(pg.image.load("assests/pipeu
        self.img_down = pg.transform.scale_by(pg.image.load("assests/pip
        self.rect_up = self.img_up.get_rect()
        self.rect_down = self.img_down.get_rect()
        self.pipe_distance = pipe_gap  # Use gap from passed argument

        self.rect_up.y = randint(250, 520)
        self.rect_up.x = 600
        self.rect_down.y = self.rect_up.y - self.pipe_distance - self.re
        self.rect_down.x = 600
        self.move_speed = move_speed
```

5.3. ENCAPSULATION:

5.3.1. DEFINITION AND ROLE:

Encapsulation involves bundling the data (attributes) and the methods (functions) that operate on that data into a single unit or class. It also helps in hiding the internal state of the object from the outside.

5.3.2. CODE:

5.3.2.1. DATA HIDING:

Attributes such as self.score, self.move_speed, and self.bird in the Game class are encapsulated within the class, meaning that they are managed internally.

```
class Game:
    def __init__(self):
        # Setting window config
        self.width = 600
        self.height = 768
        self.scale_factor = 1.5
        self.win = pg.display.set_mode((self.width, self.height))
        self.clock = pg.time.Clock()
        self.move_speed = 250
        self.start_monitoring = False
        self.score = 0
```

- Other internal states are also encapsulated.

5.3.2.2. METHOD ENCAPSULATION:

Methods like updateEverything(), drawEverything(), and checkCollisions() in the Game class control the game state without exposing the internal workings to the rest of the program.

5.4. ABSTRACTION:

5.4.1. DEFINITION AND ROLE:

Abstraction involves reducing complexity by hiding the detailed implementation and exposing only the necessary components or interfaces.

5.4.2. CODE:

5.4.2.1. GAME COMPONENTS:

The `Game` class abstracts the overall game functionality. Users of the class do not need to know how the bird or pipes are updated or drawn; they simply call methods like `gameLoop()`.

```
def gameLoop(self):
    last_time = time.time()
    while True:
        new_time = time.time()
        dt = new_time - last_time
        last_time = new_time

        for event in pg.event.get():
            if event.type == pg.QUIT:
                pg.quit()
                sys.exit()
            if event.type == pg.KEYDOWN and self.is_game_started:
                if event.key == pg.K_RETURN:
                    self.is_enter_pressed = True
                    self.bird.update_on = True
                if event.key == pg.K_SPACE and self.is_enter_pressed
                    self.bird.flap(dt)
                    self.flap_sound.play()
            if event.type == pg.MOUSEBUTTONUP:
                if self.restart_text_rect.collidepoint(pg.mouse.get_
                    self.restartGame()

        self.updateEverything(dt)
        self.checkCollisions()
        self.checkScore()
        self.drawEverything()
        pg.display.update()
        self.clock.tick(60)
```

5.4.2.2. BIRD AND PIPE BEHAVIOR:

The details of how the bird's movement is updated (gravity, flapping) and how pipes are generated and moved are hidden inside their respective classes. This allows the main game loop to work at a higher level of abstraction.

5.5. INHERITANCE:

In Python, every class automatically inherits from the built-in object class if no parent class is explicitly provided. This means that your classes already take advantage of inheritance even though you have not written an explicit parent class.

5.5.1. IMPLICIT INHERITANCE FROM PYTHON'S OBJECT:

5.5.1.1. CODE:

```
class Game:
    def __init__(self):
        # Setting window config
        self.width = 600
        self.height = 768
        self.scale_factor = 1.5
        self.win = pg.display.set_mode((self.width, self.height))
        self.clock = pg.time.Clock()
        self.move_speed = 250
        self.start_monitoring = False
        self.score = 0
```

5.5.1.2. EXPLANATION:

Although the code did not specify a parent class (such as class Game(object):), Python automatically makes Game a subclass of object. This means Game inherits basic methods and attributes (like __str__, __repr__, etc.) from object.

5.5.1.3. BIRD CLASS:

```
import pygame as pg

class Bird:
    def __init__(self, scale_factor):
        self.scale_factor = scale_factor
        self.image = pg.image.load("assests/birdup.png").convert_alpha()
        self.image = pg.image.load("assests/birddown.png").convert_alpha
        self.image = pg.transform.scale(self.image, (int(34 * self.scale_
        self.rect = self.image.get_rect(center=(100, 300))
        self.velocity = 0
        self.update_on = False  # Initially, the bird doesn't move
        self.flap_strength = -8  # Default value, changes in setGameDiff
```

5.5.1.4. EXPLANATION:

Like the Game class, Bird implicitly inherits from object. All the basic functionalities provided by Python's base class are available to Bird without extra code.

5.5.1.5. PIPE CLASS:

```
import pygame as pg
from random import randint

class Pipe:
    def __init__(self, scale_factor, move_speed, pipe_gap):  # Accept pi
        self.img_up = pg.transform.scale_by(pg.image.load("assests/pipeu
        self.img_down = pg.transform.scale_by(pg.image.load("assests/pip
        self.rect_up = self.img_up.get_rect()
        self.rect_down = self.img_down.get_rect()
        self.pipe_distance = pipe_gap  # Use gap from passed argument

        self.rect_up.y = randint(250, 520)
        self.rect_up.x = 600
        self.rect_down.y = self.rect_up.y - self.pipe_distance - self.re
        self.rect_down.x = 600
        self.move_speed = move_speed
```

5.5.1.6. EXPLANATION:

The Pipe class, just like the other two, automatically inherits from object. This provides it with a set of built-in methods and attributes that are standard to all Python objects.

5.6. MODULARITY AND CODE ORGANIZATION:

5.6.1. DEFINITION AND ROLE:

Modularity involves breaking down a program into separate modules or files. This makes the code easier to manage, test, and reuse.

5.6.2. CODE:

- **Separate Files:** Your project is organized into multiple files:
    - `game.py`: Contains the `Game` class and the main game loop.
    - `bird.py`: Contains the `Bird` class.
    - `pipe.py`: Contains the `Pipe` class.

This modular organization makes it easier to locate and modify specific parts of the game without affecting the entire codebase.

5.7. CONSTRUCTORS (__init__ METHODS):

5.7.1. DEFINITION AND ROLE:

Constructors in Python are defined by the __init__ method. They initialize the object's attributes and set up any necessary state when an object is created.

5.7.2. CODE:

- **`Game` Class Constructor:**

```
class Game:
    def __init__(self):
```

- **`Bird` Class Constructor:**

```
class Bird:
    def __init__(self, scale_factor):
```

- **`Pipe` Class Constructor:**

```
class Pipe:
    def __init__(self, scale_factor, move_speed, pipe_gap):
```

5.8. INSTANCE METHODS AND THE self KEYWORD:

5.8.1. DEFINITION AND ROLE:

Instance methods are functions defined within a class that operate on the instance (object) of that class. The self keyword refers to the current instance and is used to access variables and methods associated with the object.

5.8.2. CODE

- **Accessing Attributes and Methods:** In the `Bird` class, the `update()`, `flap()`, and `resetPosition()` methods use `self` to modify the bird's state.

```
def update(self, dt):
    if self.update_on:
        self.velocity += 0.5  # Gravity effect
        self.rect.y += self.velocity
        # Additional logic for boundary checking
```

- **Interacting with Other Objects:** In the `Game` class, the `gameLoop()` method uses `self.bird` to call methods on the Bird object.

```
if event.key == pg.K_SPACE and self.is_enter_pressed:
    self.bird.flap(dt)
    self.flap_sound.play()
```

5.9. COMPOSITION:

5.9.1. DEFINITION AND ROLE:

Composition is a design principle where a class is composed of one or more objects from other classes, indicating a "has-a" relationship. It allows for building complex objects by combining simpler ones.

5.9.2. CODE:

- **Game Composition:** The `Game` class creates and manages instances of the `Bird` and `Pipe` classes. This illustrates how the game is built from several independent components working together.

```
class Game:
    def __init__(self):
        # ...
        self.bird = Bird(self.scale_factor)
        self.pipes = []
        # ...
```

- **Inter-object Interaction:** The game loop coordinates updates, collision checks, and drawing routines by interacting with these composed objects.

EXPLANATION OF GAME.PY CODE:

1. EXPLANATION OF THE GAME INITIALIZATION CODE:

1.1. INTRODUCTION:

This section of the code is responsible for setting up the game environment, initializing important modules, loading assets, and preparing the game window. Although the full project code is distributed among several files (e.g., game.py, bird.py, pipe.py), the following explanation focuses on the Game class initialization in game.py.

1.2. MODULE IMPORTS AND INITIALIZATION:

```
import pygame as pg
import sys
import time
from bird import Bird
from pipe import Pipe
```

1.2.1. PURPOSE AND FUNCTIONALITY:

- **Pygame (pg):**
  - The code imports the Pygame library, which is used to create and manage the game's graphics, sounds, and events.
  - The alias `pg` is used throughout the code for brevity.
- **sys and time Modules:**
  - The `sys` module is imported to allow graceful exits from the program.
  - The `time` module is used for tracking time intervals, such as managing game loops or animation timing.
- **Custom Modules (`bird` and `pipe`):**
  - The `Bird` and `Pipe` classes are imported from their respective modules. These classes encapsulate the logic and attributes related to the game's player (the bird) and the obstacles (the pipes).

1.2.2. INITIALIZATION CALLS:

```
pg.init()
pg.mixer.init()  # Initialize the mixer for sound effects
```

- **`pg.init()`:**
  - Initializes all Pygame modules, preparing them for use.

- **`pg.mixer.init()`:**
    - Specifically initializes the mixer module responsible for handling sound effects. This ensures that audio assets are correctly processed when the game runs.

1.3. THE GAME CLASS CONSTRUCTOR:

1.3.1. OVERVIEW:

The Game class encapsulates the entire game logic and display setup. The constructor method (__init__) sets up the game window, clock, game parameters, assets (fonts, images, sounds), and game objects.

1.3.2. DETAILED BREAKDOWN:

1.3.2.1. WINDOW CONFIGURATION AND DISPLAY SETTINGS:

```
class Game:
    def __init__(self):
        # Setting window config
        self.width = 600
        self.height = 768
        self.scale_factor = 1.5
        self.win = pg.display.set_mode((self.width, self.height))
        self.clock = pg.time.Clock()
```

- **Window Dimensions:**
    - The game window is set to a width of 600 pixels and a height of 768 pixels.
- **Scale Factor:**
    - A scale factor (`1.5`) is defined to adjust the size of graphical assets appropriately.
- **Display Setup:**
    - `pg.display.set_mode` creates the game window with the specified dimensions.
- **Clock:**
    - `pg.time.Clock()` is used to control the frame rate of the game, ensuring smooth updates and animations.

1.3.2.2. GAME MECHANICS VARIABLES:

```
self.move_speed = 250
        self.start_monitoring = False
        self.score = 0
```

- **Movement Speed:**
    - `move_speed` determines how fast the game elements (like pipes and ground) move, affecting the overall game difficulty.
- **Score Tracking and Monitoring:**
    - `start_monitoring` is a flag used for determining when to start monitoring the bird's position relative to obstacles.
    - `score` initializes the game score to zero.

1.3.2.3. TEXT AND FONT SETUP:

```
self.font = pg.font.Font("assests/font.ttf", 24)
```

```
        self.score_text = self.font.render("Score: 0", True, (0, 0, 0))
        self.score_text_rect = self.score_text.get_rect(center=(100, 30))
        self.restart_text = self.font.render("Restart: 0", True, (0, 0, 0))
        self.restart_text_rect = self.restart_text.get_rect(center=(300, 700))
```

- **Font Loading:**
  - A custom font is loaded from the assets folder.
- **Score and Restart Text:**
  - Text surfaces are created for displaying the score and a restart prompt.
  - Their positions on the screen are set using `get_rect` to center them appropriately.

1.3.2.4. GAME OBJECT INITIALIZATION:

```
self.bird = Bird(self.scale_factor)
 self.pipes = []
```

- **Bird Instance:**
  - An instance of the `Bird` class is created, passing in the scale factor so that the bird's image is correctly scaled.
- **Pipe List:**
  - An empty list for pipes is initialized. Pipes will be added dynamically during the game.

1.3.2.5. DIFFICULTY AND TIMING VARIABLES:

```
self.difficulty_message = self.show_difficulty_message = False
        self.message_display_time = 2  # Time to display message (seconds)
        self.message_timer = 0
        self.is_enter_pressed = False
        self.is_game_started = False  # Start the game only after level
selection
```

- **Difficulty Messages:**
  - Variables to manage the display of difficulty messages.
- **Message Timer:**
  - Used to time the duration for which messages are displayed.
- **Game State Flags:**
  - `is_enter_pressed` tracks whether the player has started the game.
  - `is_game_started` ensures that gameplay begins only after the level selection process is complete.

1.3.2.6. PIPE GENERATION AND SPEED INCREASE SETTINGS:

```
self.pipe_generate_counter = 71
        self.setUpBgAndGround()
        self.level_threshold = 5
        self.max_increases = 20
        self.speed_increase_per_level = 10
```

- **Pipe Generation Counter:**
  - Used to control when a new pipe should be generated.
- **Level and Speed Settings:**
  - Variables to handle game difficulty progression. For example, the game might increase the move speed after a certain number of points or levels.

1.3.2.7. BACKGROUND AND GROUND SETUP:

```
self.setUpBgAndGround()
```

- **Method Call:**
  - This method (defined elsewhere in the code) sets up the background and ground images, ensuring they are scaled and positioned correctly in the game window.

1.3.2.8. LOADING AUDIO FILES:

```
self.dead_sound = pg.mixer.Sound("assests/sfx/dead.wav")
        self.score_sound = pg.mixer.Sound("assests/sfx/score.wav")
        self.flap_sound = pg.mixer.Sound("assests/sfx/flap.wav")
```

- **Audio Assets:**
  - Sound effects for events like the bird's death, scoring, and flapping are loaded using Pygame's mixer.
  - These sounds enhance user experience by providing auditory feedback for game actions.

1.3.2.9. LEVEL SELECTION SCREEN INITIATION:

```
self.showLevelSelectionScreen()
```

- **Starting the Game:**
  - The game begins by calling the `showLevelSelectionScreen()` method.
  - This method presents a level selection screen to the player, allowing them to choose the game's difficulty before the main game loop starts.

2. LEVEL SELECTION SCREEN EXPLANATION:

2.1. PURPOSE OF THE METHOD:

The showLevelSelectionScreen() method displays a level selection menu where the player can choose between three difficulty levels: **Simple, Medium, and Hard**. This function ensures that the game does not start immediately, but instead waits for the player to select a difficulty level before initiating gameplay.

2.2. BREAKDOWN OF THE CODE:

2.2.1. PLAYING THE START SOUND:

```
self.start_sound = pg.mixer.Sound("assests/sfx/start.wav")  # Load start sound
        self.start_sound.play()  # Play start sound
```

- Loads and plays a **start sound effect** when the level selection screen appears.
- This enhances user experience by providing audio feedback when the game is launched.

2.2.2. LOADING AND SCALING THE LEVEL SELECTION BACKGROUND:

```
 level_screen_img = pg.image.load("assests/level2.png").convert()
        level_screen_img = pg.transform.scale(level_screen_img, (self.width,
self.height))
```

- **Loads** the background image for the level selection screen.
- **Scales** the image to fit the window dimensions (`self.width = 600`, `self.height = 768`).
- This ensures that the selection screen has a proper background, making it visually appealing.

2.2.3. CREATING BUTTONS FOR DIFFICULTY LEVELS:

```
 simple_button_rect = pg.Rect(self.width / 4 - 50, self.height / 2 - 50, 100,
50)
        medium_button_rect = pg.Rect(self.width / 2 - 50, self.height / 2 -
50, 100, 50)
        hard_button_rect = pg.Rect(3 * self.width / 4 - 50, self.height / 2 -
50, 100, 50)
```

- Creates **three rectangular buttons** using `pg.Rect()` for **Simple**, **Medium**, and **Hard** levels.
- These buttons are positioned at **equal intervals across the screen** to make them easy to distinguish.
- Each button has a **width of 100 pixels and height of 50 pixels**, ensuring they are clickable.

2.2.4. DISPLAYING THE LEVEL SELECTION SCREEN:

```
selected_level = None

        while selected_level is None:
            self.win.blit(level_screen_img, (0, 0))
            pg.draw.rect(self.win, (0, 255, 0), simple_button_rect)
            pg.draw.rect(self.win, (255, 255, 0), medium_button_rect)
            pg.draw.rect(self.win, (255, 0, 0), hard_button_rect)
```

- The `while selected_level is None` **loop ensures that the player must select a level before proceeding**.
- The background image is drawn on the screen using `self.win.blit(level_screen_img, (0, 0))`.
- The three buttons are drawn using `pg.draw.rect()` with different colors:
  - **Green** (`(0, 255, 0)`) for Simple
  - **Yellow** (`(255, 255, 0)`) for Medium
  - **Red** (`(255, 0, 0)`) for Hard

2.2.5. DISPLAYING TEXT LABELS ON BUTTONS:

```
simple_text = self.font.render("Simple", True, (0, 0, 0))
            medium_text = self.font.render("Medium", True, (0, 0, 0))
            hard_text = self.font.render("Hard", True, (0, 0, 0))

            self.win.blit(simple_text, simple_button_rect.topleft)
            self.win.blit(medium_text, medium_button_rect.topleft)
            self.win.blit(hard_text, hard_button_rect.topleft)
```

- **Text is rendered** using `self.font.render()` for each difficulty level.
- The **black text (`(0, 0, 0)`)** ensures readability against the colored buttons.
- The text is **blitted (drawn) onto the buttons** at their respective positions using `self.win.blit()`.

2.2.6. HANDLING PLAYER INPUT (MOUSE CLICK ON DIFFICULTY BUTTONS):

```
for event in pg.event.get():
                if event.type == pg.QUIT:
                    pg.quit()
                    sys.exit()
                if event.type == pg.MOUSEBUTTONDOWN:
                    if simple_button_rect.collidepoint(event.pos):
                        selected_level = "simple"
                    elif medium_button_rect.collidepoint(event.pos):
                        selected_level = "medium"
                    elif hard_button_rect.collidepoint(event.pos):
                        selected_level = "hard"
```

- **Event Handling:**
  - If the player clicks the **close button**, `pg.quit()` is called to exit the game.
  - If the player **clicks on a difficulty button**, the `selected_level` variable is updated accordingly.
- **`collidepoint(event.pos)`:**
  - This function checks if the **mouse click position** (`event.pos`) is inside a button's rectangle.
  - Based on the clicked button, the difficulty level is selected.
- **Exit Condition:**
  - The loop **continues until a level is selected** (`selected_level` is no longer `None`).
  - Once a level is chosen, the game proceeds to adjust the settings accordingly.

2.2.7. SETTING THE GAME DIFFICULTY AND STARTING THE GAME:

```
self.setGameDifficulty(selected_level)
        self.gameLoop()
```

- **Calls `setGameDifficulty(selected_level)`:**

- Adjusts game parameters (like pipe speed, gap size, and bird flap strength) based on the selected difficulty.
- This ensures that the game responds dynamically to the player's choice.

- **Calls `self.gameLoop()`:**
  - Starts the main game loop after the difficulty has been set.

2.3. SUMMARY OF THE CODE FUNCTIONALITY:

Feature	Description
Audio Feedback	Plays a start sound when the level selection screen is displayed.
Visual Design	Displays a background image and buttons for selecting difficulty levels.
Button Colors	Green (Simple), Yellow (Medium), Red (Hard) for clear distinction.
Text Labels	Displays difficulty names on respective buttons for clarity.
Mouse Input Handling	Detects mouse clicks on buttons to determine the selected level.
Game Transition	Calls `setGameDifficulty()` and `gameLoop()` after selection to start gameplay.

2.4. CONCLUSION:

The showLevelSelectionScreen() method serves as an interactive **pre-game menu** where the player selects a difficulty level. It uses **Pygame's graphics, event handling, and sound functionalities** to create an intuitive and user-friendly interface. This method ensures that the game only starts after a difficulty level is chosen, improving the overall game experience.

3. SETTING THE GAME DIFFICULTY:

3.1. PURPOSE OF THE METHOD:

The setGameDifficulty() method adjusts the game's difficulty settings based on the player's choice from the **level selection screen**. It modifies key game parameters such as the **speed of obstacles (pipes), the bird's flap strength, and the gap between pipes** to increase or decrease the challenge level.

3.2. BREAKDOWN OF THE CODE:

3.2.1. SETTING PARAMETERS FOR SIMPLE MODE:

```
if level == "simple":
            self.move_speed = 200  # Slow pipe movement
            self.bird.flap_strength = -8  # Slower bird flap
            self.pipe_gap = 250  # Large gap between pipes
```

- **Pipe Speed:**

- Pipes move **slowly** at `200 pixels per second`, making it easier for the player to navigate.

- **Bird Flap Strength:**
  - The bird's flap strength is set to **-8**, meaning the bird **rises less forcefully** when the player presses SPACE.
- **Pipe Gap:**
  - The gap between pipes is **250 pixels**, giving the player a **larger space to pass through**.
- **Overall Effect:**
  - The game becomes easier because the obstacles move slowly, the bird has **gentler movements**, and the pipes have **wider gaps**.

3.2.2. SETTING PARAMETERS FOR MEDIUM MODE:

```
elif level == "medium":
            self.move_speed = 300  # Medium pipe movement
            self.bird.flap_strength = -10  # Normal bird flap
            self.pipe_gap = 200  # Medium gap between pipes
```

- **Pipe Speed:**
  - Pipes move at a **medium speed** of `300 pixels per second`, making the game slightly harder.
- **Bird Flap Strength:**
  - The bird's flap strength is **-10**, meaning it **rises higher** compared to simple mode.
- **Pipe Gap:**
  - The pipe gap is **200 pixels**, making it **smaller** than in simple mode, increasing difficulty.
- **Overall Effect:**
  - The game becomes **moderately difficult**, requiring better timing for flaps and obstacle avoidance.

3.2.3. SETTING PARAMETERS FOR HARD MODE:

```
        elif level == "hard":
            self.move_speed = 400  # Fast pipe movement
            self.bird.flap_strength = -12  # Stronger bird flap
            self.pipe_gap = 150  # Smallest gap between pipes
```

- **Pipe Speed:**
  - Pipes move **fast** at `400 pixels per second`, requiring quick reactions.
- **Bird Flap Strength:**
  - The bird's flap strength is set to **-12**, making it **jump higher per flap**, requiring better control.
- **Pipe Gap:**
  - The gap between pipes is **only 150 pixels**, making it much harder to pass through.
- **Overall Effect:**
  - The game becomes **challenging**, requiring **precise timing** and **fast reflexes** to avoid obstacles.

3.2.4. STARTING THE GAME AFTER SETTING DIFFICULTY:

```
self.is_game_started = True  # Start the game
```

- **Game Start Condition:**
  - Once the difficulty is set, `self.is_game_started` is set to `True`.
  - This allows the main game loop to start and the bird to begin moving.

3.3. PHYSICS CONCEPT: GRAVITY IN BIRD'S MOVEMENT:

The code indirectly includes a **gravity-like effect** through the bird's **flap strength**.

- **Flap Strength (self.bird.flap_strength)**
  - This represents the **upward force** applied when the player presses SPACE.
  - In different difficulty levels, the strength varies (-8, -10, -12), affecting how high the bird jumps.
- **Gravity-Like Effect in Bird Update (bird.py)**
  - The bird's velocity **increases over time**, pulling it downward.
  - If no flap occurs, the bird continues to fall, simulating **gravity**.

Even though there is no explicit gravity variable, the combination of **flap strength and velocity update** creates a gravity-like effect.

3.4. SUMMARY OF THE CODE FUNCTIONALITY:

Feature	Simple Mode	Medium Mode	Hard Mode
Pipe Speed	200 px/sec (slow)	300 px/sec (medium)	400 px/sec (fast)
Bird Flap Strength	-8 (low jump)	-10 (medium jump)	-12 (high jump)
Pipe Gap	250 pixels (large)	200 pixels (medium)	150 pixels (small)
Overall Difficulty	Easy	Moderate	Hard

3.5. CONCLUSION:

The setGameDifficulty() method customizes the **game experience** by adjusting **pipe speed, bird movement, and obstacle gaps**. It ensures that players can select a difficulty level that matches their skill and makes the game more engaging by providing **dynamic gameplay adjustments**.

4. GAME LOOP EXPLANATION:

4.1. PURPOSE OF THE GAME LOOP:

The gameLoop() function is responsible for **continuously updating the game state**, handling **user inputs**, updating the **bird's motion**, checking for **collisions**, and **rendering graphics**. This loop ensures that the game runs smoothly by executing repeatedly at a fixed frame rate (60 FPS).

4.2. BREAKDOWN OF THE CODE:

4.2.1. TIME MANAGEMENT FOR SMOOTH FRAME RATE:

```
last_time = time.time()
        while True:
            new_time = time.time()
            dt = new_time - last_time
            last_time = new_time
```

- `time.time()` retrieves the current time in seconds.
- `dt` (delta time) calculates the **time difference between frames** to ensure smooth motion, independent of system performance.
- This prevents inconsistencies in movement speed on **faster or slower computers**, maintaining a **uniform experience**.

4.2.2. HANDLING USER INPUT (KEYBOARD AND MOUSE EVENTS):

```
for event in pg.event.get():
                if event.type == pg.QUIT:
                    pg.quit()
                    sys.exit()
```

- This **detects if the user closes the window** and terminates the program properly.

```
if event.type == pg.KEYDOWN and self.is_game_started:
                        if event.key == pg.K_RETURN:
                            self.is_enter_pressed = True
                            self.bird.update_on = True
```

- **Pressing ENTER starts the game**, allowing the bird to move.

```
if event.key == pg.K_SPACE and self.is_enter_pressed:
                            self.bird.flap(dt)
                            self.flap_sound.play()
```

- **Pressing SPACE makes the bird flap upward.**
- The `flap_sound` is played for **auditory feedback**.

```
if event.type == pg.MOUSEBUTTONUP:
                        if
    self.restart_text_rect.collidepoint(pg.mouse.get_pos()):
                            self.restartGame()
```

- If the user clicks the **restart button**, the game resets.

4.2.3. UPDATING GAME ELEMENTS:

```
 self.updateEverything(dt)
```

- This updates all game objects, including **bird movement, pipe positions, and ground movement**.

```
                self.checkCollisions()
```

- Checks **if the bird collides** with pipes or the ground, triggering a game-over state.

```
                self.checkScore()
```

- Increases the **player's score** when the bird successfully passes a pipe.

4.2.4. RENDERING GRAPHICS AND MAINTAINING FRAME RATE:

```
self.drawEverything()
            pg.display.update()
            self.clock.tick(60)
```

- `self.drawEverything()` **renders** all game elements.
- `pg.display.update()` **refreshes** the screen.
- `self.clock.tick(60)` **ensures the game runs at 60 FPS**, preventing excessive CPU usage.

4.3. PHYSICS CONCEPTS IN THE GAME LOOP:

4.3.1. GRAVITY AND FREE FALL IN THE BIRD'S MOTION:

The game loop **continuously updates** the bird's position using **Newton's laws of motion**:

$v=u+gtv = u + gt$

where:

- v = final velocity
- u = initial velocity
- g = gravitational acceleration (simulated as 0.5 per frame in your code)
- t = delta time (dt)

Since the game loop **updates the bird's motion every frame**, the velocity increases over time, causing the bird to fall faster—just like in real physics.

4.3.2. IMPULSE FORCE APPLIED DURING FLAPPING:

When the player **presses SPACE**, the bird applies an **impulse force** upward:

$F=ma$

- **Impulse force** is applied only once when the button is pressed.
- The **flap strength** acts as an initial velocity in the opposite direction of gravity, following:

$h=v0t+12gt2$

where:

- h = bird's upward displacement
- vO = initial velocity (flap strength: -8, -10, or -12 based on difficulty)
- g = downward acceleration (0.5 per frame)
- t = time elapsed after the flap

This equation explains why **flapping moves the bird up initially**, but gravity **pulls it down** over time.

5. RESTART GAME EXPLANATION:

5.1. PURPOSE:

This function **resets the game state** when the player chooses to restart after a game over.

5.2. BREAKDOWN OF THE CODE:

```
self.over_sound = pg.mixer.Sound("assests/sfx/over.wav")
        self.over_sound.play()
```

- **Plays a game-over sound effect** to indicate the restart.

```
self.score = 0
        self.score_text = self.font.render("Score: 0", True, (0, 0, 0))
```

- **Resets the player's score to 0** and updates the displayed score text.

```
 self.is_enter_pressed = False
        self.is_game_started = False
```

- Ensures that the game does **not start automatically** after restarting.

```
self.bird.resetPosition()
```

- Calls a method to **reset the bird's position** to the starting point.

```
self.pipes.clear()
        self.pipe_generate_counter = 71
```

- **Removes all pipes from the game** so that they do not persist after restarting.

```
self.bird.update_on = False
```

- Prevents the bird from moving until the player starts the game again.

```
 self.showLevelSelectionScreen()
```

- **Redirects the player back to the level selection screen** to choose a difficulty before playing again.

6. CHECK SCORE EXPLANATION:

6.1. PURPOSE:

This function **updates the player's score** when the bird successfully passes through a pipe.

6.2. BREAKDOWN OF THE CODE:

```
        if len(self.pipes) > 0:
```

- Ensures that there are pipes in the game before checking for scoring conditions.

```
 if self.bird.rect.left > self.pipes[0].rect_down.left and
self.bird.rect.right < self.pipes[0].rect_down.right and not
self.start_monitoring:
                self.start_monitoring = True
```

- When the bird **enters the space between pipes**, `self.start_monitoring` is set to `True`, marking that the bird is in the scoring zone.

```
if self.bird.rect.left > self.pipes[0].rect_down.right and
self.start_monitoring:
                self.start_monitoring = False
                self.score += 1
```

- When the bird **completely passes the pipe**, `self.score` increases by **1**.

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

- Updates the **score text on the screen** and plays a **scoring sound effect**.

7. EXPLANATION OF checkCollisions()METHOD:

7.1. PURPOSE OF THE METHOD:

The checkCollisions() method detects **whether the bird collides with the ground or pipes**. If a collision occurs, the game **stops the bird's movement** and plays a **death sound**.

7.2. BREAKDOWN OF THE CODE:

7.2.1. GROUND COLLISION DETECTION (BIRD FALLING TO THE GROUND):

```
if self.bird.rect.bottom > 568:
                self.bird.update_on = False
                self.is_enter_pressed = False
                self.is_game_started = False
                self.dead_sound.play()
```

- **Checks if the bird's bottom edge goes below a y-position of `568` pixels**, meaning it has hit the ground.
- **Stops the bird's movement** by setting `self.bird.update_on = False`.
- **Ends the game** by setting `self.is_game_started = False`.
- **Plays a "dead" sound** to indicate game over.

7.2.2. PHYSICS CONCEPT: FREE FALL & IMPACT WITH THE GROUND:

The bird **falls due to gravity**, following the equation of motion:

$v=u+gt$

where:

- v = final velocity before hitting the ground
- u = initial velocity (previous frame's velocity)
- g = gravitational acceleration (approximated as `0.5 per frame` in your code)
- t = time elapsed

When the bird **hits the ground**, it undergoes an **inelastic collision**, meaning:

1. **Velocity is suddenly reduced to zero.**
2. **Motion stops immediately.**

This follows the **impulse-momentum theorem**:

$F \cdot \Delta t = m \cdot \Delta v$

where:

- F = force experienced by the bird
- Δt = time of impact (very small)
- m = mass of the bird
- Δv = change in velocity (from falling speed to zero)

Since Δt is very small, **the force on impact is large**, which is why real-world objects break when hitting the ground. In the game, **this force is not visualized**, but the **game stops the bird's motion** when it collides with the ground.

7.2.3. COLLISION WITH PIPES (OBSTACLE COLLISION DETECTION):

```
if (self.bird.rect.colliderect(self.pipes[0].rect_down) or
self.bird.rect.colliderect(self.pipes[0].rect_up)):
                self.is_enter_pressed = False
                self.is_game_started = False
                self.dead_sound.play()
```

- **Checks if the bird's rectangle collides with either the top or bottom pipe.**
- **Stops the game immediately** when a collision is detected.
- **Plays the "dead" sound to signal game over.**

7.2.4. PHYSICS CONCEPT: COLLISION WITH AN OBSTACLE:

When the bird collides with a pipe, it experiences a **sudden change in velocity**, which follows the principle of **Newton's Third Law**:

For every action, there is an equal and opposite reaction. For every action, there is an equal and opposite reaction.

- The **bird moves forward** and **hits the pipe**, which applies a **reaction force** back onto the bird.
- Since the **bird's movement stops instantly**, this represents a **perfectly inelastic collision**, meaning **all kinetic energy is lost in the impact**.

The **momentum before collision** is:

$P_{before} = mv$

After the collision, the bird stops moving, so the **momentum after collision is zero**:

$P_{after} = 0$

This means the **impulse force** exerted by the pipe is:

$F \cdot \Delta t = m \cdot \Delta v$

Since **Δv\Delta v is large (the bird suddenly stops)** and **Δt\Delta t is small**, the force exerted by the pipe on the bird is **very high**, which is why the game **instantly ends the movement** upon impact.

8. EXPLANATION OF UPDATEEVERYTHING METHOD:

8.1. GROUND MOVEMENT LOGIC:

The ground scrolls continuously to create a moving effect by shifting two ground rectangles (ground1_rect and ground2_rect). When one moves completely off-screen, it is repositioned to the right of the other, ensuring a seamless loop.

8.2. PIPE GENERATION AND MOVEMENT:

New pipes are generated periodically when pipe_generate_counter exceeds 70, and the counter resets. Existing pipes move leftward based on the game's speed (move_speed). Pipes that move off-screen are removed from the list to optimize performance and free memory.

8.3. BIRD MOVEMENT:

The bird's update function is called continuously, ensuring its position and animations are updated based on the game state.

8.4. DIFFICULTY MESSAGE DISPLAY:

A difficulty message is temporarily displayed to the player. It remains visible for a set duration (message_display_time) and is hidden once the current time (time.time()) exceeds the recorded message start time (message_timer). This ensures that the message disappears automatically after the intended period.

This part of the code contributes to gameplay mechanics, optimizing resource management and enhancing the user experience through smooth animations and difficulty indicators.

9. EXPLANATION OF drawEverything():

9.1. PURPOSE OF drawEverything():

This function is responsible for rendering all game elements on the screen, ensuring that the graphical components of the game are displayed correctly.

- **Background Rendering:** The background image (`bg_img`) is drawn at a fixed position to give a static visual reference.
- **Pipe Rendering:** Each pipe in the `self.pipes` list is drawn using its `drawPipe()` method, ensuring that obstacles appear on the screen.
- **Ground Rendering:** The two ground images (`ground1_img` and `ground2_img`) are drawn at their respective positions, creating a scrolling ground effect.
- **Bird Rendering:** The bird sprite is drawn at its current position (`bird.rect`).
- **Score Display:** The player's score is displayed using `self.score_text`.

- **Restart and End Messages:** If the game has not started (`self.is_game_started` is `False`), the restart message is displayed. Additionally, a "Nice try! Game is ended. Try again" message appears when the game ends.
- **Difficulty Message Display:** If `self.show_difficulty_message` is `True`, a message appears in the center of the screen to indicate the difficulty level.

10. EXPLANATION OF `setUpBgAndGround()`:

10.2. PURPOSE OF setUpBgAndGround():

This function initializes the background and ground images for the game environment.

- **Loading Images:** The background (`bg.png`) and ground (`ground.png`) images are loaded from the `assets` folder and scaled according to `scale_factor`.
- **Creating Ground Rectangles:** Two ground images are used (`ground1_img` and `ground2_img`) to create a seamless scrolling effect.
- **Positioning the Ground:** The first ground image starts at `x = 0`, and the second one is placed immediately after it (`ground1_rect.right`). Both ground images are positioned at `y = 568` to align them properly at the bottom of the screen.

EXPLANATION OF BIRD.PY CODE:

1. BIRD CLASS EXPLANATION:

1.1. OVERVIEW OF THE Bird CLASS:

The Bird class represents the player-controlled bird in the game. It handles movement, gravity, and flapping mechanics. The bird's behavior is controlled by updating its position based on velocity and ensuring it stays within screen boundaries.

1.2. EXPLANATION OF KEY METHODS

1.2.1. __init__(self, scale_factor) – INITIALIZATION:

- Loads the bird's image from assets (`birdup.png` and `birddown.png`), but only the last loaded image is used due to overwriting.
- Scales the image based on `scale_factor` to fit game dimensions.
- Creates a rectangle (`self.rect`) to define the bird's position and hitbox.
- Initializes `velocity = 0` to keep the bird stationary at the start.
- `update_on = False` ensures the bird remains static until it flaps.
- `flap_strength = -8` determines the bird's upward force when flapping.

1.2.2. update(self, dt) – APPLYING GRAVITY AND MOVEMENT:

1.2.2.1. CODE:

```
def update(self, dt):
        if self.update_on:  # Update only if the game has started
            self.velocity += 0.5  # gravity effect
            self.rect.y += self.velocity
```

```
            if self.rect.top <= 0:
                self.rect.top = 0
            if self.rect.bottom >= 568:
                self.rect.bottom = 568
                self.velocity = 0  # Stop falling when hitting the ground
```

1.2.2.2. EXPLANATION:

- The `update_on` flag ensures that the bird starts moving only after the first flap.
- **Gravity is simulated** by increasing `self.velocity` by `0.5` in each update. This mimics the effect of a downward force acting on the bird.
- The bird's position (`self.rect.y`) is updated by adding the velocity.
- **Boundary Conditions:**
    - If the bird moves above the screen (`rect.top <= 0`), its position is fixed at the top.
    - If it falls below the ground (`rect.bottom >= 568`), it stops moving to prevent it from falling off-screen.

1.2.3. flap(self, dt) – MAKING THE BIRD JUMP:

1.2.3.1 CODE:

```
def flap(self, dt):
        self.velocity = self.flap_strength  # Apply the flap strength
        self.update_on = True  # Start the bird's movement after the first
flap
```

1.2.3.2. EXPLANATION:

- When the player flaps, `self.velocity` is set to `self.flap_strength (-8)`, giving the bird an **instant upward push**.
- This negative velocity counters gravity momentarily, allowing the bird to rise before gravity pulls it back down.
- `self.update_on = True` ensures that the bird starts moving after the first flap.

1.2.4. resetPosition(self) – RESETTING THE BIRD:

1.2.4.1. CODE:

```
def resetPosition(self):
        self.rect.center = (100, 300)
        self.velocity = 0
```

1.2.4.2. EXPLANATION:

- Resets the bird's position to the initial coordinates `(100, 300)`.
- Resets velocity to `0`, stopping any movement.

1.3. PHYSICS LAWS USED IN THIS CODE:

1.3.1. NEWTON'S FIRST LAW OF MOTION (INERTIA):

"An object remains at rest or in uniform motion unless acted upon by an external force."

- Initially, the bird does not move (`update_on = False`), meaning no external force is applied.
- When the player flaps, an upward force (flap strength) is applied, changing the bird's motion.

1.3.2. NEWTON'S SECOND LAW OF MOTION:

"Force = Mass × Acceleration (F = ma)"

- Gravity is simulated by increasing `velocity` by `0.5`, which represents constant acceleration.
- When the bird flaps, a sudden force (negative velocity) is applied, overcoming gravity momentarily.

1.3.3. GRAVITY AND FREE FALL MOTION:

"Objects accelerate downward due to gravity unless a force opposes it."

- The bird experiences acceleration downward (`self.velocity += 0.5`) due to gravity.
- When the player flaps, the bird experiences an upward force, temporarily counteracting gravity.

1.4. SUMMARY:

Aspect	Details
Class Name	`Bird`
Main Purpose	Handles bird movement, gravity, and flapping mechanics
Gravity Simulation	`self.velocity += 0.5` increases velocity over time, mimicking gravity
Upward Motion	`self.velocity = self.flap_strength (-8)` applies an instant upward force
Boundaries	The bird is prevented from flying off-screen using conditions
Physics Laws	Newton's First and Second Laws, Gravity, Free Fall Motion

EXPLANATION OF PIPE.PY CODE:

1. PIPE CLASS EXPLANATION:

1.1. OVERVIEW OF THE Pipe CLASS:

The Pipe class represents the obstacles that the bird must avoid. It consists of two pipe images—one facing upwards and one facing downwards—spaced apart by a gap (pipe_gap). The pipes move from right to left across the screen at a defined speed (move_speed).

1.2. EXPLANATION OF KEY METHODS:

1.2.1. __init__(self, scale_factor, move_speed, pipe_gap) – INITIALIZATION:

1.2.1.1. CODE:

```
def __init__(self, scale_factor, move_speed, pipe_gap):  # Accept pipe_gap as
argument
        self.img_up =
pg.transform.scale_by(pg.image.load("assests/pipeup.png").convert_alpha(),
scale_factor)
        self.img_down =
pg.transform.scale_by(pg.image.load("assests/pipedown.png").convert_alpha(),
scale_factor)
        self.rect_up = self.img_up.get_rect()
        self.rect_down = self.img_down.get_rect()
        self.pipe_distance = pipe_gap  # Use gap from passed argument

        self.rect_up.y = randint(250, 520)
        self.rect_up.x = 600
        self.rect_down.y = self.rect_up.y - self.pipe_distance -
self.rect_up.height
        self.rect_down.x = 600
        self.move_speed = move_speed
```

1.2.1.2. EXPLANATION:

- Loads and scales the two pipe images (`pipeup.png` and `pipedown.png`).
- Defines `rect_up` and `rect_down` to track the positions of the upper and lower pipes.
- The position of the upper pipe is randomly generated (`randint(250, 520)`) to create variation.
- The lower pipe's position is determined using the upper pipe's position and the gap (`pipe_gap`).
- The pipes start at `x = 600` (off-screen on the right) and move leftward.
- The movement speed is stored in `self.move_speed`.

1.2.2. drawPipe(self, win) – RENDERING THE PIPES:

1.2.2.1. CODE:

```
def drawPipe(self, win):
        win.blit(self.img_up, self.rect_up)
        win.blit(self.img_down, self.rect_down)
```

1.2.2.2. EXPLANATION:

- Draws the upper and lower pipe images at their respective positions.

1.2.3. update(self, dt) – MOVING THE PIPES:

1.2.3.1. CODE:

```
    def update(self, dt):
        self.rect_up.x -= int(self.move_speed * dt)
        self.rect_down.x -= int(self.move_speed * dt)
```

1.2.3.2. EXPLANATION:

- Moves both pipes leftward at a speed determined by `self.move_speed * dt`.
- `dt` (delta time) ensures smooth movement regardless of frame rate.
- The pipes move continuously from right to left until they exit the screen.

1.3. PHYSICS LAWS USED IN THIS CODE:

1.3.1. NEWTON'S FIRST LAW OF MOTION (INERTIA):

"An object remains at rest or in uniform motion unless acted upon by an external force."

- The pipes **continuously move** to the left at a constant speed (`move_speed * dt`).
- No external force changes their velocity, so they keep moving in the same direction at a steady rate.

1.3.2. NEWTON'S SECOND LAW OF MOTION (F = ma):

"Force = Mass × Acceleration."

- In this case, the **pipes have a uniform motion**, meaning there is no acceleration (they move at a constant speed).
- However, if the difficulty increases (by increasing `move_speed`), their velocity increases, which simulates a greater force being applied to the pipes.

1.3.3. RELATIVE MOTION PRINCIPLE:

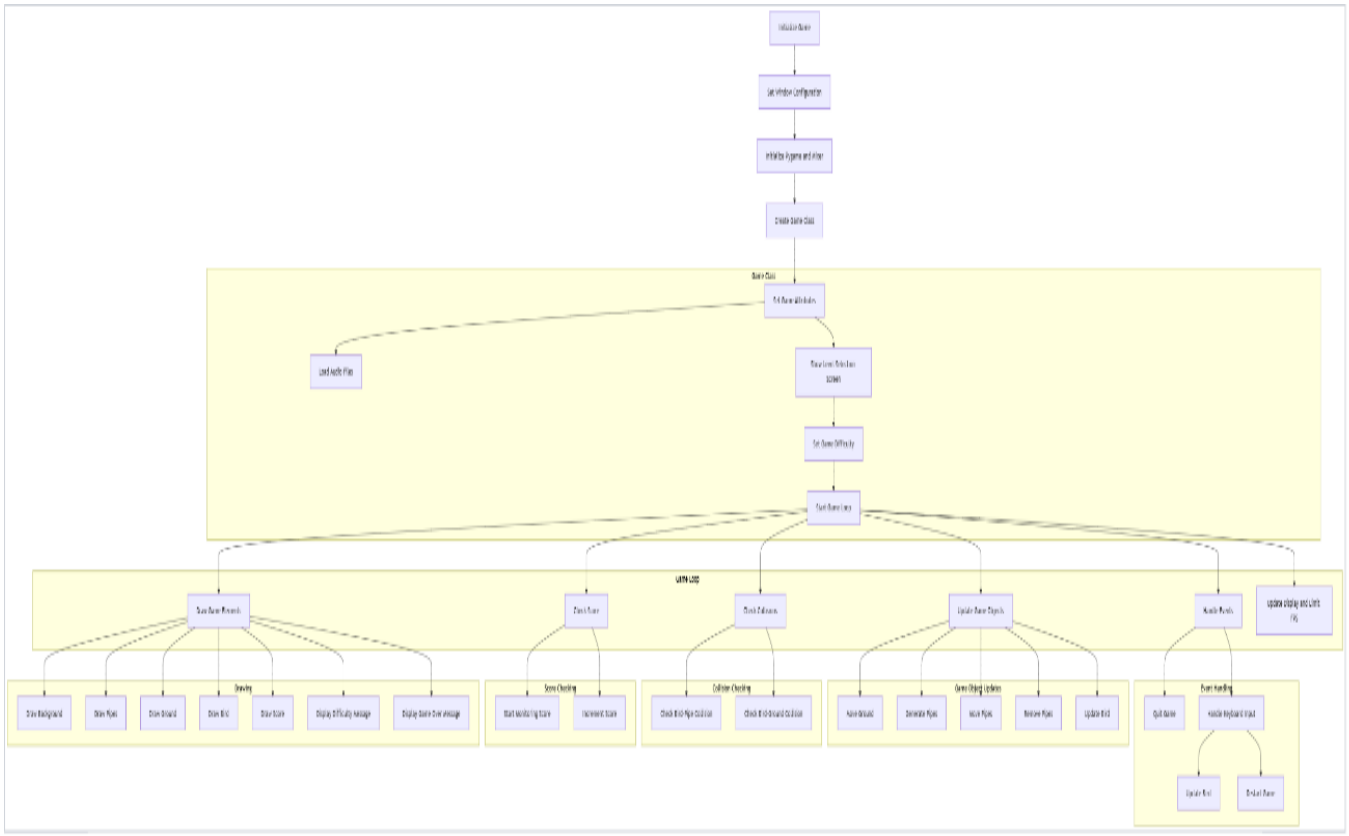
"Motion is relative to a chosen reference frame."

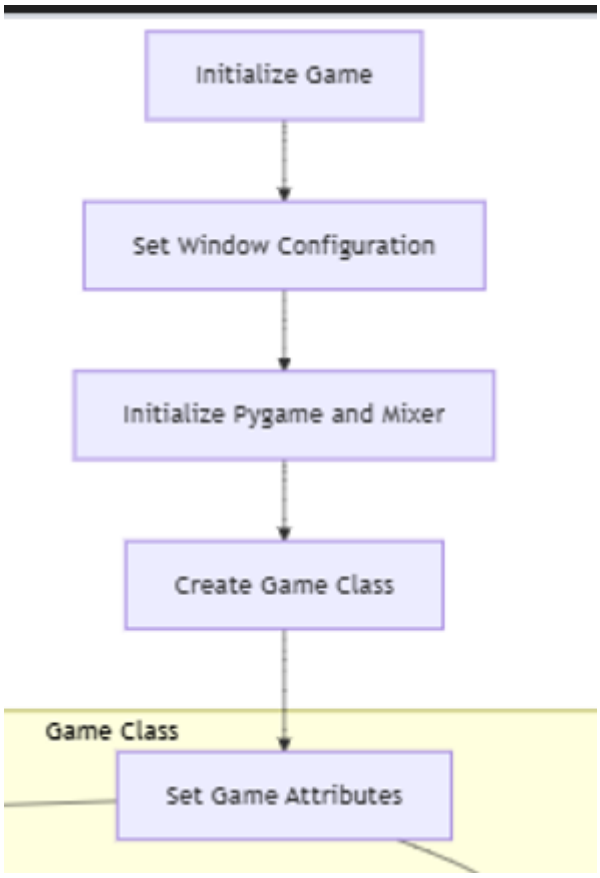
- The pipes **appear to move left**, but in reality, the bird is stationary while the background elements move.
- This gives the illusion that the bird is flying forward while the environment moves past it.

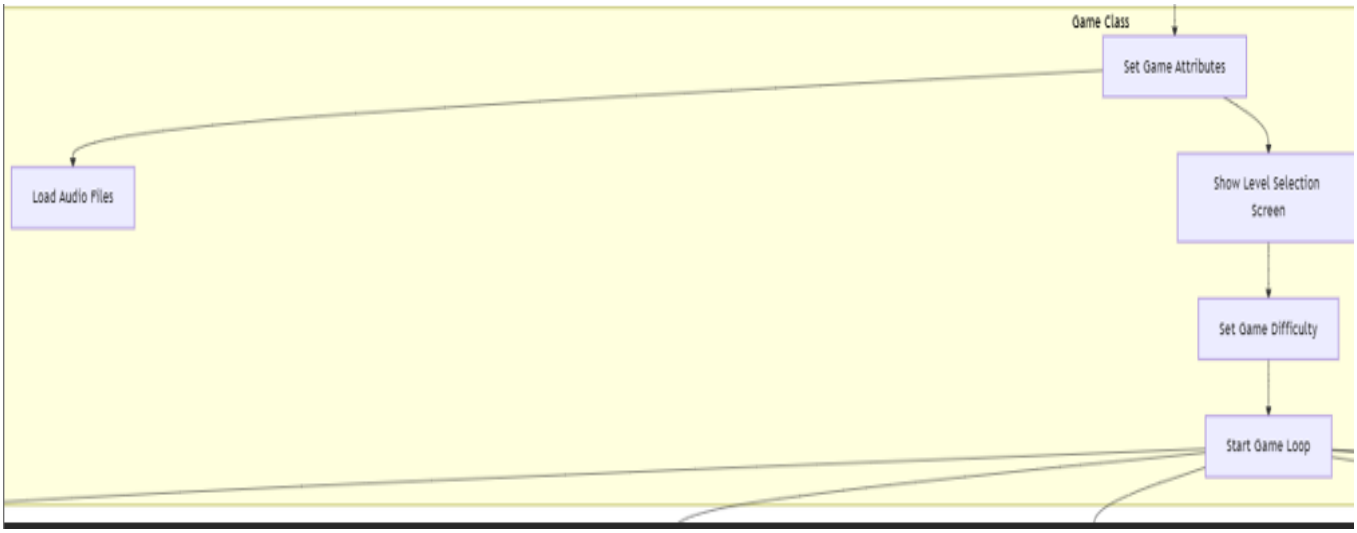
1.4. SUMMARY:

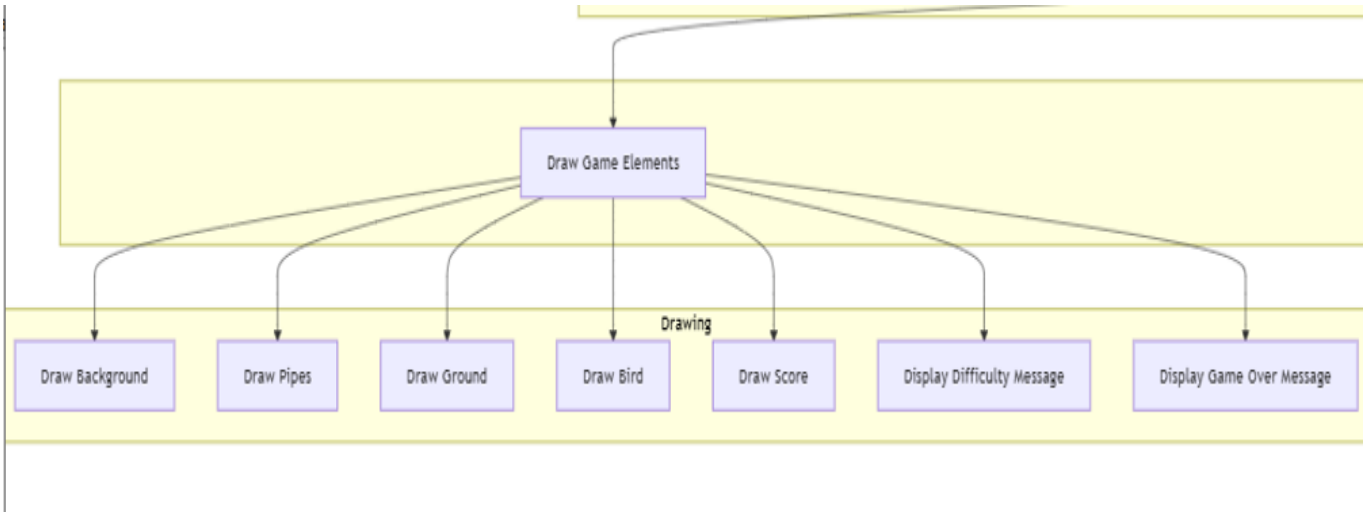
Aspect	Details
Class Name	`Pipe`
Main Purpose	Generates and moves obstacles for the game
Pipe Movement	Moves left at a constant speed (`move_speed * dt`)
Random Positioning	The upper pipe is randomly placed, and the lower pipe is adjusted accordingly
Physics Laws	Newton's First and Second Laws, Relative Motion Principle

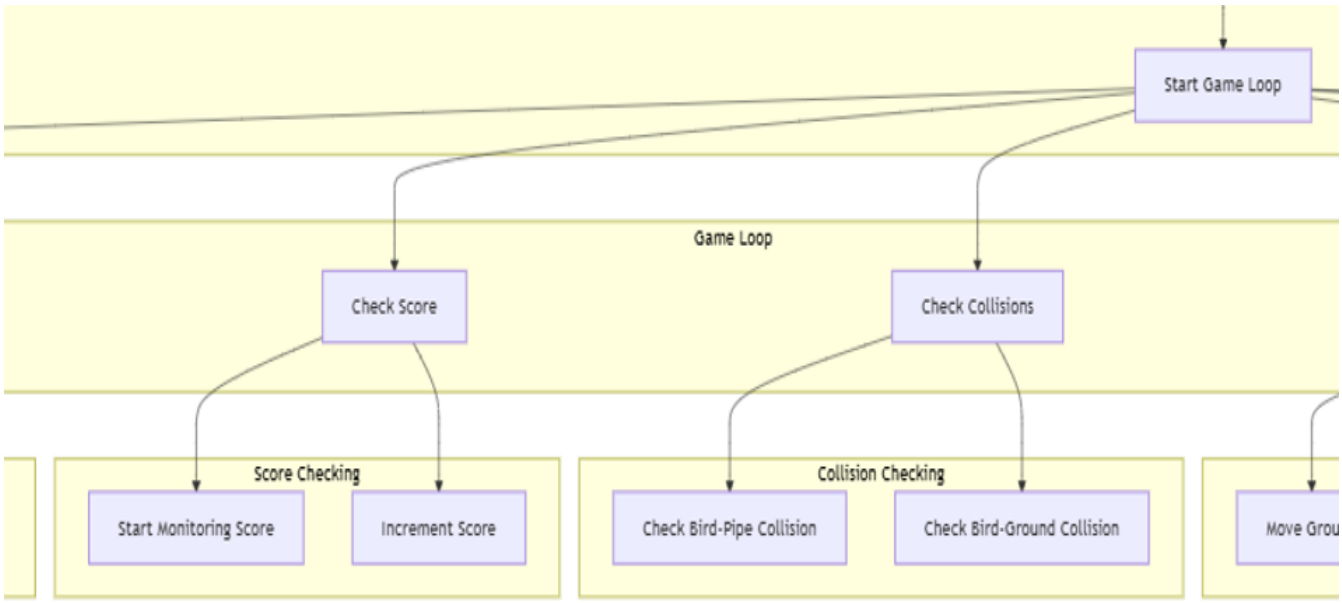
2. FLOWCHART:

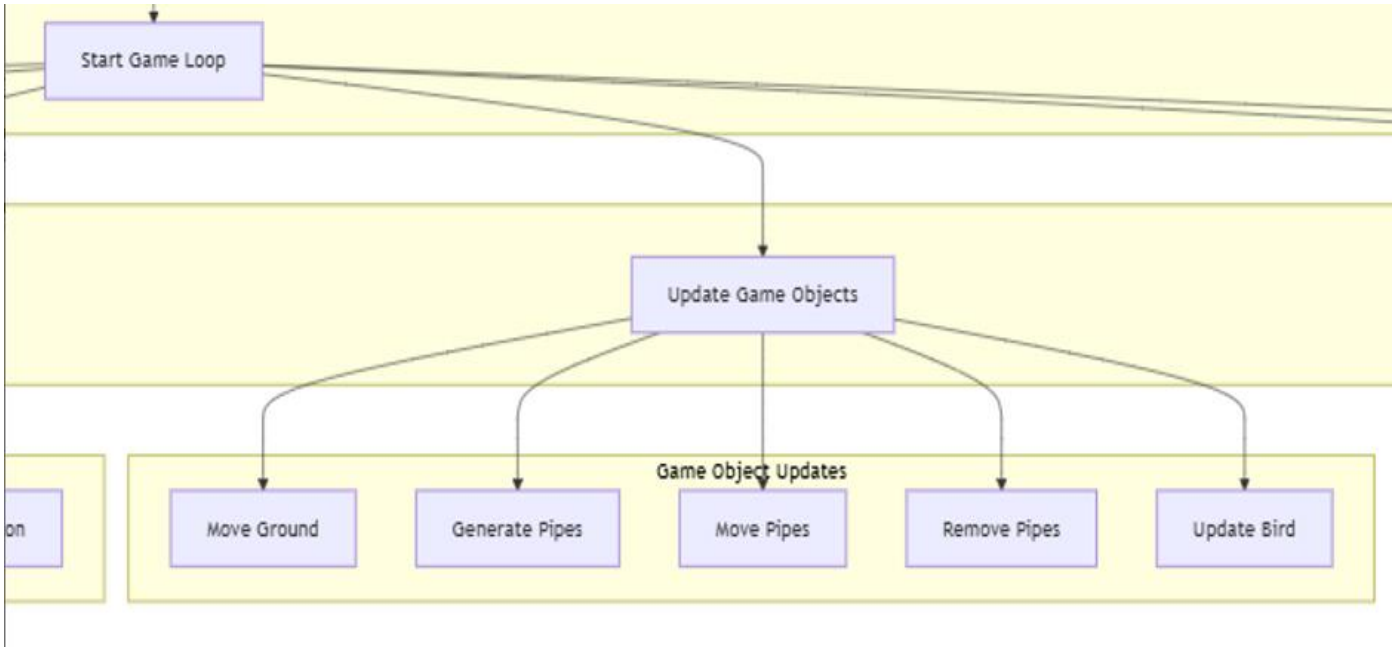


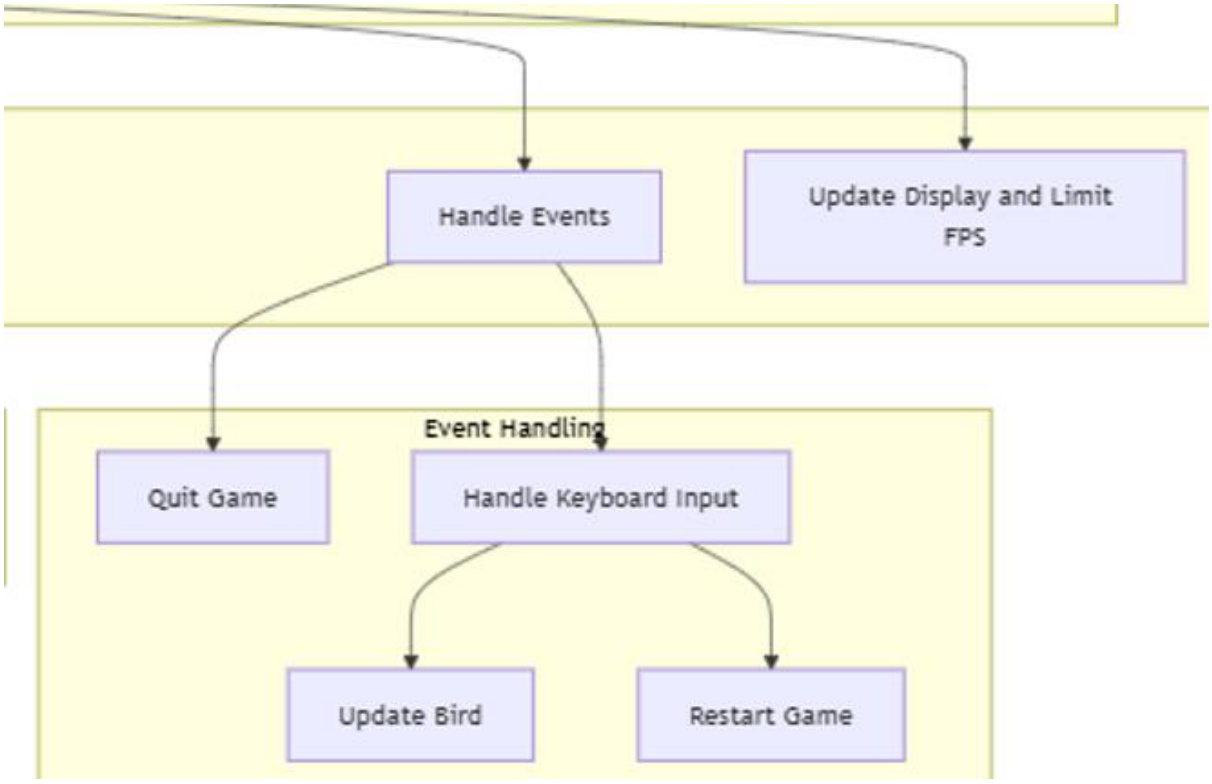

Initialize Game
Set Window Configuration
Initialize Pygame and Mixer
Create Game Class
Game Class
Set Game Attributes


Game Class
Set Game Attributes
Load Audio Files
Show Level Selection Screen
Set Game Difficulty
Start Game Loop


Draw Game Elements
Drawing
Draw Background
Draw Pipes
Draw Ground
Draw Bird
Draw Score
Display Difficulty Message
Display Game Over Message


Start Game Loop
Game Loop
Check Score
Check Collisions
Score Checking
Start Monitoring Score
Increment Score
Collision Checking
Check Bird-Pipe Collision
Check Bird-Ground Collision





3. CONCLUSION:

3.1. SUMMARY OF THE PROJECT:

This project involved the development of a 2D game where a bird navigates through obstacles while obeying physics-based movement rules. The key mechanics implement ed include:

- **Bird Movement:** The bird experiences gravity and moves upwards when flapped.

- **Pipe Obstacles:** Randomly placed pipes move from right to left, creating challenges for the player.
- **Scrolling Ground:** The ground moves continuously to simulate forward motion.
- **Difficulty Scaling:** The game adjusts difficulty dynamically by modifying pipe speed and gaps.

Through this project, I successfully integrated **physics principles** such as Newton's Laws of Motion and gravity to create a more realistic gaming experience.

3.2. KEY LEARNINGS:

This project provided valuable insights into:

- **Physics in Game Development:** Understanding how real-world physics can be simulated using code.
- **Smooth Rendering:** Ensuring seamless movement of objects without frame rate dependency.
- **Game Logic and State Management:** Implementing conditions for movement, collision detection, and difficulty scaling.
- **Optimization and Performance:** Managing game objects efficiently to prevent lag or unnecessary resource usage.

Overall, this project helped in bridging theoretical physics concepts with practical programming applications.

4. FUTURE IMPROVEMENTS:

4.1. ADDITIONAL PHYSICS ELEMENTS:

To enhance the physics-based realism, the following improvements can be made:

- **Wind Resistance:** Introduce air resistance that slightly slows down the bird's upward movement.
- **Variable Gravity:** Implement changing gravity based on different difficulty levels or special game modes.

4.2. ENHANCED GRAPHICS AND ANIMATION:

- **Improved Bird Animation:** Implement smoother frame transitions when the bird flaps.
- **Better Background Elements:** Add more layers of parallax scrolling to create depth in the environment.

4.3. MORE GAME FEATURES:

- **New Obstacles:** Introduce moving pipes or other dynamic challenges.
- **Power-ups:** Include special items like slow-motion or speed boosts.
- **Sound Effects & Music:** Improve player engagement by adding background music and interactive sound effects.

These enhancements will make the game more engaging and challenging while improving user experience.

5. REFERENCES:

I wrote most of the code from this YouTube link https://youtube.com/playlist?list=PLz7mmheDeooW795HsNzkLS6TFQSa4Idtw&si=cIfKj5MeJVSIHuG9 and also took help from ChatGPT.