Flappy Bird Game

A PROJECT REPORT
for
MINI PROJECT(ID201B)
Session (2024-25)

Submitted by

Sangam Kumar
(202410116100181)
Sandeep Kumar
(202410116100180)
Satyam Gupta
(202410116100189)
Saqib Mehdi
(202410116100185)

Submitted in partial fulfilment of the
Requirements for the Degree of

MASTER OF COMPUTER APPLICATION

Under the Supervision of
Dr. Vipin Kumar
Assistant Professor

Submitted to

DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206

CERTIFICATE

CERTIFIED THAT SANDEEP KUMAR 202410116100180, SATYAM GUPTA 202410116100189, SANGAM KUMAR 202410116100181 , SAQIB MEHDI 202410116100185 HAVE CARRIED OUT THE PROJECT WORK HAVING "PORTFOLIO BUILDER (MINI PROJECT-II, ID201B) FOR MASTER OF COMPUTER APPLICATION FROM DR. A.P.J. ABDUL KALAM TECHNICAL UNIVERSITY (AKTU) (FORMERLY UPTU), LUCKNOW UNDER MY SUPERVISION. THE PROJECT REPORT EMBODIES ORIGINAL WORK, AND STUDIES ARE CARRIED OUT BY THE STUDENT HIMSELF/HERSELF AND THE CONTENTS OF THE PROJECT REPORT DO NOT FORM THE BASIS FOR THE AWARD OF ANY OTHER DEGREE TO THE CANDIDATE OR TO ANYBODY ELSE FROM THIS OR ANY OTHER UNIVERSITY/INSTITUTION.

DR. VIPIN KUMAR

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER APPLICATION
KIET GROUP OF INSTITUTIONS, GHAZIABAD

DR. ARUN KR. TRIPATHI

DEAN

DEPARTMENT OF COMPUTER APPLICATIONS
KIET GROUP OF INSTITUTION, GHAZIABAD

ABSTRACT

THE FLAPPY BIRD GAME IS A 2D SIDE-SCROLLING ARCADE GAME DEVELOPED USING CORE JAVA WITH A FOCUS ON SMOOTH ANIMATION, PHYSICS-BASED MECHANICS, AND AN INTERACTIVE USER INTERFACE. THE OBJECTIVE OF THE GAME IS TO CONTROL A BIRD AND NAVIGATE IT THROUGH GAPS BETWEEN RANDOMLY PLACED PIPES. THE GAME INCORPORATES A GRAVITY-BASED MOVEMENT SYSTEM, REQUIRING THE PLAYER TO PRESS A KEY (OR TAP ON THE SCREEN) TO MAKE THE BIRD FLAP UPWARDS WHILE CONTINUOUSLY AVOIDING OBSTACLES.

THE GAME FOLLOWS A WELL-DEFINED WORKFLOW, BEGINNING WITH GAME INITIALIZATION, WHERE ASSETS SUCH AS SPRITES, BACKGROUNDS, AND SOUNDS ARE LOADED, AND NECESSARY VARIABLES (SUCH AS SCORE, BIRD POSITION, AND VELOCITY) ARE SET UP. THE PLAYER STARTS FROM A MENU SCREEN, AND UPON INPUT, THE GAME TRANSITIONS INTO ACTIVE GAMEPLAY. THE BIRD'S MOVEMENT IS PHYSICS-BASED, MEANING IT NATURALLY FALLS DUE TO GRAVITY, AND FLAPS COUNTERACT THIS FORCE. PIPE OBSTACLES ARE GENERATED DYNAMICALLY, MOVING FROM RIGHT TO LEFT AT A CONSTANT SPEED. COLLISION DETECTION ALGORITHMS ENSURE THAT IF THE BIRD HITS A PIPE OR THE GROUND, THE GAME ENDS.

A SCORE SYSTEM IS IMPLEMENTED, REWARDING THE PLAYER FOR SUCCESSFULLY PASSING THROUGH THE GAPS BETWEEN PIPES. THE GAME-OVER SCREEN DISPLAYS THE FINAL SCORE AND PROVIDES OPTIONS TO EITHER RESTART THE GAME OR EXIT THE APPLICATION. THE PROJECT APPLIES JAVA'S OBJECT-ORIENTED PROGRAMMING (OOP) PRINCIPLES FOR STRUCTURING CODE EFFICIENTLY, ENSURING BETTER MAINTAINABILITY AND EASIER FUTURE ENHANCEMENTS.

THIS GAME SERVES AS AN ENGAGING PROJECT THAT SHOWCASES JAVA PROGRAMMING, EVENT HANDLING, GRAPHICAL RENDERING (USING JAVA'S GUI FRAMEWORKS LIKE SWING OR JAVAFX), AND FUNDAMENTAL GAME DEVELOPMENT CONCEPTS. THE STRUCTURED DESIGN MAKES IT EASY TO ADD FUTURE IMPROVEMENTS, SUCH AS DIFFICULTY SCALING, NEW VISUAL THEMES, OR SOUND EFFECTS, MAKING IT A STRONG FOUNDATION FOR FURTHER GAME DEVELOPMENT.

TABLE OF CONTENTS

Certificate

Abstract

Acknowledgment

1. INTRODUCTION.....2
   1.1
   1.2 Project scope
   1.3 Project Overview

2. FEASIBILITY STUDY.....4

   1. Technical
   2. Operational
   3. Behavioral

4. HARDWARE AND SOFTWARE 8

5. CODE IMPLIMENTATION 9

6. PROJECT FLOW 25

7. PROJECT OUTCOME.....42

8. USER INTERFACE 45

9. REFERENCES.....53

Introduction

The Flappy Bird game is a 2D side-scrolling arcade game where the player controls a bird that continuously falls due to gravity. The primary goal is to keep the bird airborne by tapping or pressing a key to make it flap its wings. The player must navigate the bird through a series of vertical pipes that have gaps at random heights. If the bird collides with a pipe or the ground, the game ends. The difficulty of the game increases over time as the speed of the pipes gradually accelerates. The game is designed using Core Java, leveraging Java Swing and AWT for graphical rendering, as well as basic physics mechanics to control movement and collision detection.

1.2 Project Scope

The scope of this project includes the complete development of a Flappy Bird clone using Core Java. The project aims to create a fully functional, interactive, and visually appealing game with smooth mechanics and realistic gameplay behavior. Key features of the game include:

- Bird Movement: The bird moves downward due to gravity and can ascend when the player presses a key.
- Obstacle Generation: Randomly generated pipes appear on the screen at varying heights.
- Collision Detection: The game ends when the bird collides with an obstacle or the ground.
- Score System: Players earn points by successfully passing through pipe gaps.
- Game UI: A start screen, game screen, and game-over screen for a seamless user experience.
- Optimized Performance: Ensuring the game runs smoothly on any system with Java installed.

1.3 Project Overview

The game consists of multiple Java classes, each handling a specific aspect of the gameplay, including:

- Graphics Rendering: Drawing the bird, pipes, background, and score display using Java Swing.
- User Input Handling: Detecting key presses to control the bird’s movement.
- Physics Implementation: Applying gravity and motion mechanics to simulate realistic movement.
- Collision Detection System: Detecting when the bird collides with an obstacle or the ground.

- Game Loop Management: Ensuring smooth frame rendering and game updates.
- Score Tracking: Keeping count of successful pipe passes and displaying the score.

Chapter 2
Feasibility Study

2.1 Technical Feasibility

This section focuses on the technical aspects of developing and running the Flappy Bird game, highlighting the simplicity and efficiency of using Core Java with Java Swing and AWT.

- Core Java:
    - Core Java provides all the necessary tools for developing a game without relying on third-party libraries. Using Java Swing (for the graphical user interface) and AWT (for rendering and user input handling) allows you to create a fully functional game with built-in features.
    - By avoiding external libraries, the project remains lightweight, meaning it has fewer dependencies, which makes it easier to maintain and distribute.
- Graphical Rendering:
    - Swing provides various built-in components for rendering the game's graphical elements, such as buttons, panels, and images. AWT and Swing together allow you to draw custom graphics like the bird, pipes, score counter, and background.
    - For rendering smooth animations (like the bird's wing flapping), the game loop can be implemented in a way that ensures continuous updating of the graphical display. Swing's paintComponent() method can be used to redraw the screen at each frame.
- Event Handling:
    - Java provides an event-handling system that is very effective for handling user inputs, such as mouse clicks or keyboard presses (in this case, the spacebar). The game can detect when the player clicks the mouse or presses the spacebar to make the bird flap, and this event will trigger the appropriate game response (bird movement).
    - The KeyListener or MouseListener interfaces can be used to handle player inputs, ensuring smooth interaction without delays.
- Threading for Game Loop:

- Java's multithreading capabilities allow for smooth, real-time game loops. By creating a separate thread for the game's updates and rendering, you can ensure that the game runs smoothly even when handling player inputs, physics calculations, and graphics updates.
- This helps keep the game responsive by decoupling the rendering of the game graphics from other operations like collision detection or scoring updates, which ensures that the game continues running smoothly without freezing.

2.2 Operational Feasibility

This section highlights how easily the game can be run on different systems and its suitability for various players.

- System Requirements:
  - The game only requires Java Runtime Environment (JRE), which is available on almost all platforms (Windows, macOS, Linux). Since Java is platform-independent, the game can be played on any system that has Java installed, meaning there's no need for special configurations.
  - The game has minimal system resource requirements. Unlike more graphically demanding games, Flappy Bird uses relatively low CPU and memory resources. This makes it accessible even on computers with lower specifications, ensuring a wide audience can enjoy it.
- Simple Controls:
  - The controls are incredibly simple, making the game easy for anyone to pick up and play. The use of the spacebar or mouse clicks to control the bird's movement means that no complicated control schemes or tutorials are needed. Players simply need to press the spacebar or click to make the bird flap.
  - The simplicity of the controls makes it accessible to all ages and provides a seamless experience without requiring a steep learning curve.
- Target Audience:
  - This game is ideal for casual gaming, meaning it is suitable for short gaming sessions that players can easily pick up and play during their free time.

- It's also great for educational purposes. The game can be used as a teaching tool for beginners learning Java, demonstrating important concepts such as game loops, event handling, and graphic rendering. Moreover, it provides an opportunity for students to practice their Java skills in a fun and interactive environment.

2.3 Behavioral Feasibility

This section examines the player experience and engagement, focusing on the psychological factors that make the game appealing.

- Popular Mechanics:
  - Since Flappy Bird is a widely recognized game with simple yet addictive mechanics, most players will immediately understand the rules and objectives. The goal of the game is clear: the bird must fly through gaps in the pipes without touching them. The simplicity of the objective makes it easy for anyone to grasp.
  - Because the game follows familiar mechanics from the original Flappy Bird, players are already conditioned to know how to play, which minimizes any barriers to entry.
- Challenging Yet Simple:
  - The game strikes the perfect balance between challenge and simplicity. The controls are easy to learn, but as the pipes scroll faster and their positions change, the game gradually becomes harder, encouraging players to improve their skills.
  - This progressive difficulty keeps players engaged by providing a constantly evolving challenge that feels rewarding when overcome. Players are motivated to keep playing to beat their previous scores or reach a higher level, which leads to replay value.
- Engaging and Competitive Aspect:
  - The game's competitive nature is driven by the score system. Players are motivated to surpass their own high scores or compete with friends, creating a sense of competition. This can lead to repeated playthroughs, as players try to improve their performance and achieve a higher score.
  - Additionally, the short game sessions make it easy for players to pick up the game and play several rounds in one sitting, which encourages repeated engagement. Even when players fail, they can quickly restart and try again, making the game feel like an ongoing challenge rather than a one-time experience.

- Replayability:
    - The simple but difficult nature of the game encourages replayability. Players are likely to return to the game multiple times, trying to achieve a better score or simply enjoying the challenge. The addictive gameplay loop — try, fail, improve — is what keeps players hooked.
    - The lack of complex storylines or features means players can focus purely on improving their skill and score, which provides a timeless and evergreen gameplay loop.

Chapter 3

Hardware and Software Requirements

4.1 Hardware Requirements:

- Minimum 2GB RAM (Recommended: 4GB or higher)
- Intel or AMD processor (1GHz or higher)
- Standard keyboard and mouse

4.2 Software Requirements:

- Java Development Kit (JDK) 8 or later
- Integrated Development Environment (IDE) such as Eclipse, IntelliJ IDEA, or NetBeans
- Java Swing and AWT for UI and graphical components
- Sound and Image Resources for enhanced visuals and effects

Chapter 4

CODE Implementation

1. Flappy Bird Game

```
package com.mycompany.flappybird;

import javax.swing.*;

public class FlappyBirdGame {

    public static void main(String[] args) {
        int boardWidth = 360;
        int boardHeight = 640;

        JFrame frame = new JFrame("Flappy Bird");
        frame.setSize(boardWidth, boardHeight);
        frame.setLocationRelativeTo(null);
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        GamePanel gamePanel = new GamePanel();
        frame.add(gamePanel);
        frame.pack();
        gamePanel.requestFocus();
        frame.setVisible(true);
    }
```

```
}
```

2. Game Panel

```
package com.mycompany.flappybird;

import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.util.Random;
import javax.sound.sampled.*;
import javax.swing.*;

public class GamePanel extends JPanel implements ActionListener, KeyListener {

    int boardWidth = 360;
    int boardHeight = 640;

    //Images
    Image backgroundImg;
    Image birdImg;
    Image topPipeImg;
    Image bottomPipeImg;
```

```
//Bird
int birdX = boardWidth / 8;
int birdY = boardHeight / 2;
int birdWidth = 34;
int birdHeight = 24;

class Bird {

    int x = birdX;
    int y = birdY;
    int width = birdWidth;
    int height = birdHeight;
    Image img;

    Bird(Image img) {
        this.img = img;
    }
}

//Pipes
int pipeX = boardWidth;
int pipeY = 0;
int pipeWidth = 64;
int pipeHeight = 512;

class Pipe {

    int x = pipeX;
    int y = pipeY;
```

```
    int width = pipeWidth;
    int height = pipeHeight;
    Image img;
    boolean passed = false;

    Pipe(Image img) {
        this.img = img;
    }
}

//game logic
Bird bird;
int velocityX = -4;
int velocityY = 0;
int gravity = 1;

ArrayList<Pipe> pipes;
Random random = new Random();

Timer gameLoop;
Timer placePipesTimer;

boolean gameOver = false;
boolean gameStarted = false;
boolean easyMode=false;
boolean hardMode=false;
static long highscore = 0;
double score = 0;
```

```
    Clip jumpClip;
    Clip pointClip;
    Clip hitClip;
    Clip dieClip;
    Clip swooshClip;
    Clip backgroundClip;

    GamePanel() {
        setPreferredSize(new Dimension(boardWidth, boardHeight));
        setFocusable(true);
        addKeyListener(this);

        //loading image files
        backgroundImg = new
ImageIcon(getClass().getResource("./flappybirdbg.png")).getImage();
        birdImg = new ImageIcon(getClass().getResource("./flappy_bird.gif")).getImage();
        topPipeImg = new ImageIcon(getClass().getResource("./toppipe.png")).getImage();
        bottomPipeImg = new
ImageIcon(getClass().getResource("./bottompipe.png")).getImage();

        //loading audio files
        try {
            jumpClip = AudioSystem.getClip();

jumpClip.open(AudioSystem.getAudioInputStream(getClass().getResource("./jump.wav"
)));

            pointClip = AudioSystem.getClip();
```

```
pointClip.open(AudioSystem.getAudioInputStream(getClass().getResource("./point.wav"
)));

        hitClip = AudioSystem.getClip();

hitClip.open(AudioSystem.getAudioInputStream(getClass().getResource("./hit.wav")));

        dieClip = AudioSystem.getClip();

dieClip.open(AudioSystem.getAudioInputStream(getClass().getResource("./die.wav")));

        swooshClip = AudioSystem.getClip();

swooshClip.open(AudioSystem.getAudioInputStream(getClass().getResource("./swooshi
ng.wav")));

        backgroundClip = AudioSystem.getClip();

backgroundClip.open(AudioSystem.getAudioInputStream(getClass().getResource("./Tint
in.wav")));
    } catch (Exception e) {
        e.printStackTrace();
    }

    //bird
    bird = new Bird(birdImg);
    //pipes
    pipes = new ArrayList<Pipe>();
```

```
    //place pipes timer
    placePipesTimer = new Timer(1500, new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            placePipes();
        }
    });

    //game timer
    gameLoop = new Timer(1000 / 60, this);

    playBackgroundSound();
}

//play jump sound
public void playJumpSound() {
    if (jumpClip != null) {
        jumpClip.setFramePosition(0);
        jumpClip.start();
    }
}
//play point sound
public void playPointSound() {
    if (pointClip != null) {
        pointClip.setFramePosition(0);
        pointClip.start();
    }
}
//play hit sound
```

```
    public void playHitSound() {
        if (hitClip != null) {
            hitClip.setFramePosition(0);
            hitClip.start();
        }
    }
    //play die sound
    public void playDieSound() {
        if (dieClip != null) {
            dieClip.setFramePosition(0);
            dieClip.start();
        }
    }
    //play swooshing sound
    public void playSwooshSound() {
        if (swooshClip != null) {
            swooshClip.setFramePosition(0);
            swooshClip.start();
        }
    }
    //play background sound
    public void playBackgroundSound() {
        if (backgroundClip != null) {
            backgroundClip.setFramePosition(0);
            backgroundClip.start();
            FloatControl gainControl = (FloatControl)
backgroundClip.getControl(FloatControl.Type.MASTER_GAIN);
            gainControl.setValue(-3.0f); //reducing the volume
        }
```

```
    }

    public void placePipes() {
        int randomPipeY = (int) (pipeY - pipeHeight / 4 - Math.random() * (pipeHeight /
2));
        int openingSpace = boardHeight / 4;

        Pipe topPipe = new Pipe(topPipeImg);
        topPipe.y = randomPipeY;
        pipes.add(topPipe);

        Pipe bottomPipe = new Pipe(bottomPipeImg);
        bottomPipe.y = topPipe.y + pipeHeight + openingSpace;
        pipes.add(bottomPipe);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        draw(g);
    }

    public void draw(Graphics g) {
        //background
        g.drawImage(backgroundImg, 0, 0, boardWidth, boardHeight, null);

        //bird
        g.drawImage(bird.img, bird.x, bird.y, bird.width, bird.height, null);

        //pipes
```

```
for (int i = 0; i < pipes.size(); i++) {
    Pipe pipe = pipes.get(i);
    g.drawImage(pipe.img, pipe.x, pipe.y, pipe.width, pipe.height, null);
}

//welcome screen
if (!gameStarted) {
    g.setColor(Color.yellow);
    g.setFont(new Font("Serif", Font.BOLD, 24));
    g.drawString("WELCOME TO FLAPPY BIRD", 10, 125);
    g.setColor(Color.cyan);
    g.setFont(new Font("Monospaced", Font.BOLD, 25));
    g.drawString("PLay by clicking SPACE", 12, 160);
    g.setColor(Color.white);
    g.setFont(new Font("Serif", Font.BOLD, 25));
    g.drawString("Press 'D' to play in Day Mode", 18, 195);
    g.setColor(Color.gray);
    g.setFont(new Font("Serif", Font.BOLD, 25));
    g.drawString("Press 'N' to play in Night Mode", 12, 230);
    g.drawImage(bird.img, bird.x, bird.y, bird.width, bird.height, null);
}

//score
g.setColor(Color.yellow);
g.setFont(new Font("Arial", Font.BOLD, 32));
if (gameOver) {
    g.setFont(new Font("Serif", Font.BOLD, 32));
    g.drawString("GAME OVER", 10, 35);
    g.drawString("Score: " + String.valueOf((int) score), 10, 70);
```

```
            g.setColor(Color.white);
            g.setFont(new Font("Monospaced", Font.BOLD, 23));
            g.drawString("Press ENTER to play again", 5, 100);
            g.setFont(new Font("Monospaced", Font.BOLD, 19));
            if(easyMode)
                g.drawString("Press 'H' to shift to HARD Mode", 5, 128);
            else if(hardMode)
                g.drawString("Press 'E' to shift to EASY Mode", 5, 128);
        }
        else if (!gameOver && gameStarted) {
            g.setColor(Color.white);
            g.setFont(new Font("Arial", Font.BOLD, 15));
            g.drawString("High Score: " + String.valueOf(highscore), 10, 20);
            g.setColor(Color.yellow);
            g.setFont(new Font("Serif", Font.BOLD, 32));
            g.drawString("Score: " + String.valueOf((int) score), 10, 55);
        }
    }

    public void move() {
        //bird
        velocityY += gravity;
        bird.y += velocityY;
        bird.y = Math.max(bird.y, 0);

        //pipes
        for (int i = 0; i < pipes.size(); i++) {
            Pipe pipe = pipes.get(i);
            pipe.x += velocityX;
```

```
            if (!pipe.passed && bird.x > pipe.x + pipe.width) {
                pipe.passed = true;
                score += 0.5;
                playPointSound();
            }

            if (collision(bird, pipe)) {
                gameOver = true;
                playHitSound();
            }
        }

        if (bird.y > boardHeight) {
            gameOver = true;
            playDieSound();
        }
    }

    public boolean collision(Bird a, Pipe b) {
        return a.x < b.x + b.width
                && a.x + a.width > b.x
                && a.y < b.y + b.height
                && a.y + a.height > b.y;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        move();
```

```
    repaint();
    if (gameOver) {
        placePipesTimer.stop();
        gameLoop.stop();
    }
}

@Override
public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_SPACE && gameStarted) {
        velocityY = -9;
        playJumpSound();
    }
    //play again
    if (e.getKeyCode() == KeyEvent.VK_ENTER && gameOver) {
        playSwooshSound();
        //restrart the game
        if (score > highscore) {
            highscore = (long) score;
        }
        bird.y = birdY;
        velocityY = 0;
        pipes.clear();
        score = 0;
        gameOver = false;
        gameLoop.start();
        placePipesTimer.start();
    }
    //easy mode
```

```
if (e.getKeyCode() == KeyEvent.VK_E && gameOver && hardMode) {
    playSwooshSound();
    //restrart the game
    if (score > highscore) {
        highscore = (long) score;
    }
    bird.y = birdY;
    velocityY = 0;
    pipes.clear();
    score = 0;
    highscore = 0;
    gameOver = false;
    gameLoop.start();
    placePipesTimer = new Timer(1500, new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            placePipes();
        }
    });
    placePipesTimer.start();
    velocityX = -4;
    easyMode=true;
    hardMode=false;
}
//hard mode
if (e.getKeyCode() == KeyEvent.VK_H && gameOver && easyMode) {
    playSwooshSound();
    //restrart the game
    if (score > highscore) {
```

```
            highscore = (long) score;
        }
        bird.y = birdY;
        velocityY = 0;
        pipes.clear();
        score = 0;
        highscore = 0;
        gameOver = false;
        gameLoop.start();
        placePipesTimer = new Timer(750, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                placePipes();
            }
        });
        placePipesTimer.start();
        velocityX = -8;
        hardMode = true;
        easyMode=false;
    }
    //day mode
    if (e.getKeyCode() == KeyEvent.VK_D && !gameStarted) {
        backgroundImg = new
ImageIcon(getClass().getResource("./flappybirdbg.png")).getImage();
        gameStarted = true;
        playSwooshSound();
        gameLoop.start();
        placePipesTimer.start();
        easyMode=true;
```

```
        }
        //night mode
        if (e.getKeyCode() == KeyEvent.VK_N && !gameStarted) {
            backgroundImg = new
ImageIcon(getClass().getResource("./fbBG_night.png")).getImage();
            gameStarted = true;
            playSwooshSound();
            gameLoop.start();
            placePipesTimer.start();
            easyMode=true;
        }
    }

    @Override
    public void keyTyped(KeyEvent e) {
    }

    @Override
    public void keyReleased(KeyEvent e) {
    }
}
```

Chapter 5
Project Flow

The game follows a structured workflow to ensure smooth execution and interaction. Below is the step-by-step process:

5.1 Game Initialization

This is the first step that sets up everything required before the gameplay begins.

- Asset Loading:
    - Load images: Bird sprites (with flapping frames), background, pipes, and ground.
    - Load sound effects: Flap sound, collision sound, score-up sound, game over music, etc.
    - Optional: Use buffered image loading with image caching for performance.
- Variable Initialization:
    - Bird's starting position (x, y) and its motion parameters like velocityY, gravity, and jumpStrength.
    - Game state enum or flag (START, PLAYING, GAME_OVER).
    - Score counter, high score, pipe list, pipe generation interval, pipe speed.
- Window Setup:
    - Use JFrame for the main game window.
    - Set size (e.g., 800x600), title, and close operation.
    - Add a JPanel or custom canvas for rendering and input.
- Game Loop Preparation:
    - Use javax.swing.Timer or a separate thread for the game loop.
    - The loop continuously calls the update and repaint functions (e.g., 60 frames per second).
    - Organize game logic updates, rendering, and input polling within this loop.

5.2 Start Screen Display

The game welcomes the player with an intuitive and clean start screen.

- Visual Elements:
    - Display the game's logo or title in a prominent and appealing font.
    - Show a "Press Space to Start" or "Click to Play" prompt using simple animations to draw attention.
- Audio:
    - Optional background music or ambient sound effects can enhance the feel.
- Interaction:
    - Wait for the player's input (key press or mouse click).
    - Once input is received, change the game state from START to PLAYING.

5.3 User Input Handling

Responsive controls are essential for gameplay precision.

- Key Detection:
    - Detect specific keys (like Space or Arrow Up) for flapping.
    - Ensure key events are debounced or rate-limited to avoid multiple triggers on a single press.
- Touch/Click Support (Optional):
    - Detect mouse clicks or touch gestures for broader platform compatibility.
- Start Screen Input:
    - A single press from the start screen initiates gameplay.
- In-Game Input:
    - A keypress during play causes the bird to jump, altering its vertical velocity upward.

5.4 Bird Movement

The bird's motion simulates real-world physics to create a challenging and fluid feel.

- Gravity and Velocity:

- Gravity increases the bird's downward velocity over time.
- A flap resets the velocity to a negative value (upward movement).

- Movement Update:
  - Update the bird's Y-position each frame using its velocity.
  - Prevent the bird from going above the screen or falling infinitely.
- Animation:
  - Rotate the bird slightly upward during a flap and downward when falling to simulate natural flight motion.
  - Switch between different sprite frames for a flapping animation.

5.5 Pipe Generation and Movement

Pipes provide the main obstacle and must be continuously generated and recycled.

- Timing:
  - Pipes are generated at fixed intervals (e.g., every 1.5 seconds).
- Randomization:
  - Each new pipe pair has a randomly determined gap position within a defined range.
- Movement:
  - Pipes move horizontally to the left at a fixed speed.
  - When pipes move off-screen, remove them from memory to optimize performance.
- Structure:
  - Each pipe set includes an upper pipe and a lower pipe, leaving a space between them for the bird to pass.

5.6 Collision Detection

The core gameplay challenge lies in avoiding collisions.

- Bounding Box Detection:
  - Use rectangles to represent the bird and pipes and check for intersection.

- Ground Collision:
    - If the bird touches the bottom of the screen (ground), it triggers game over.
- Accuracy:
    - Optimize the hitboxes to closely match sprite shapes.
    - Consider a margin or padding for a fair hitbox that isn't too strict.
- Result:
    - Trigger game-over state and play sound upon collision.

5.7 Score Updating

Scoring keeps the player engaged and motivated.

- Mechanism:
    - Each time the bird successfully passes a pipe pair (crosses the pipe's X center), increment the score.
- Display:
    - Render the score at the top of the screen in a large, readable font.
- Sound Feedback:
    - Play a point sound to reward the player.
- High Score:
    - Maintain a high score value stored in a local file or memory and display it after the game ends.

5.8 Game Over Detection

When the game ends, it's important to clearly communicate this to the player.

- Trigger:
    - Collision with a pipe or ground initiates the game over sequence.
- Transition:

    - Optional short pause before transitioning to the game over screen to allow the player to process what happened.

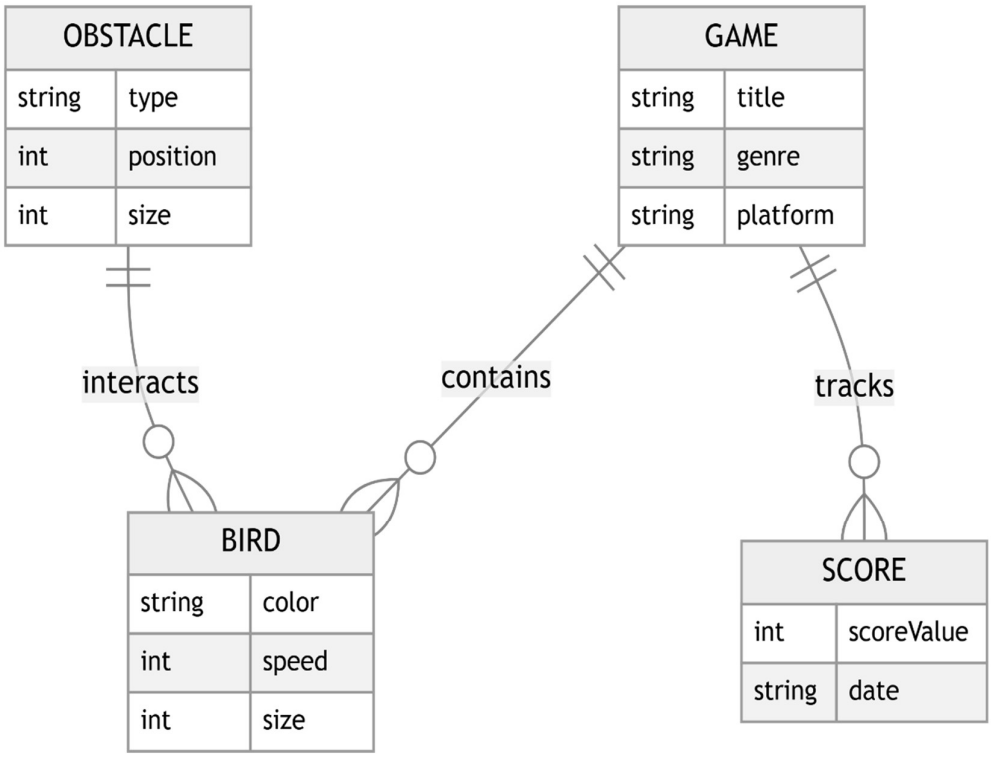
- UI:
    - Show “Game Over” text, final score, and optionally the best score.
    - Provide a button or key prompt for restarting or exiting the game.

5.9 Restart or Exit

Let the player control what happens next.

- Restart Option:
    - Reset all game variables (score, pipe list, bird position/velocity).
    - Clear screen and return to the start state or start a new game immediately.
- Exit Option:
    - Cleanly close the window using System.exit(0) and release any resources.
- Resource Management:
    - Ensure all graphics, sounds, and threads are properly stopped or disposed of to prevent memory leaks.

ER DIAGRAM



This ER (Entity-Relationship) diagram models the Flappy Bird Game in terms of its key components and their relationships. Let's break it down entity by entity and explain how they interact with one another:

Entities and Their Attributes

1. GAME

- Attributes:
    - title (string): Name of the game (e.g., "Flappy Bird").
    - genre (string): Game type (e.g., "Arcade").
    - platform (string): Platform it runs on (e.g., "Desktop", "Android").
- Role: Central entity that holds and manages the entire game state.

2. BIRD

- Attributes:
    - color (string): Bird's color (e.g., "yellow").
    - speed (int): Current velocity or movement speed.
    - size (int): Bird's size or radius for collision.
- Role: The main character controlled by the player.

3. OBSTACLE

- Attributes:
    - type (string): Type of obstacle (e.g., "pipe", "ground").
    - position (int): Position on screen (X or Y coordinate depending on logic).
    - size (int): Size or height/width of the obstacle.
- Role: Represents pipes or other elements the bird must avoid.

4. SCORE

- Attributes:
    - scoreValue (int): Points scored by the player.
    - date (string): Date and time the score was achieved.

- Role: Keeps track of game performance.

🔷 Relationships

1. GAME contains BIRD

- Multiplicity: 1 GAME contains exactly 1 BIRD.
- Explanation: Each game session has one bird that the player controls.

2. GAME tracks SCORE

- Multiplicity: 1 GAME can track multiple SCORE records.
- Explanation: The game tracks various scores across different sessions or players.

3. BIRD interacts with OBSTACLE

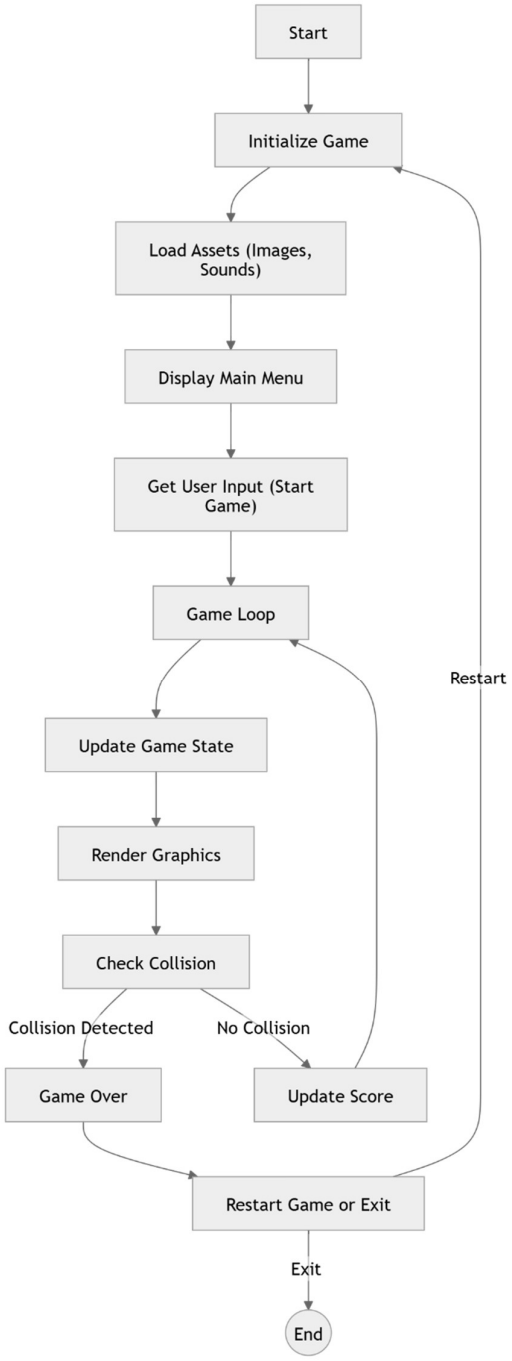
- Multiplicity: 1 BIRD can interact with many OBSTACLES; an OBSTACLE can also be interacted with by many BIRDs (in multiplayer or simulation contexts).
- Explanation: The bird interacts (typically collides or passes through) with obstacles like pipes and the ground.

🧩 Diagram Interpretation Summary

Entity	Attribute Examples	Connected To	Relationship	Notes
GAME	title, genre, platform	BIRD, SCORE	contains, tracks	Central unit managing bird and scores
BIRD	color, speed, size	GAME, OBSTACLE	interacts	Main player character
OBSTACLE	type, position, size	BIRD	interacts	Pipes and ground
SCORE	scoreValue,	GAME	tracked	Performance

Entity	Attribute Examples	Connected To	Relationship	Notes
	date		by	records

Flowchart



This flowchart provides a visual representation of the game lifecycle for your Flappy Bird game in Core Java, from launch to either restart or exit. Let's go through it step by step:

1. Start

- Entry point of the game. When the user launches the game application, the process begins here.

2. Initialize Game

- Game window and core components are set up.
- Variables like score, bird position, gravity, and speed are initialized.

3. Load Assets (Images, Sounds)

- Load all necessary resources:
    - Images: Bird sprite, pipes, background, ground, etc.
    - Sounds: Flap sound, collision sound, scoring sound, background music.

4. Display Main Menu

- The start screen or title screen is shown.
- May include:
    - Game logo/title
    - Instructions or a "Press Space to Start" message
    - Background visuals/music

5. Get User Input (Start Game)

- Waits for the player to press a key (like Spacebar) or click/tap to begin gameplay.
- On receiving valid input, the game transitions into the main game loop.

6. Game Loop (Main Gameplay Cycle)

This is the heart of the game, continuously repeating to drive real-time interactivity.

a. Update Game State

- Update bird position based on gravity and flap input.
- Move pipes from right to left.
- Generate new pipes and remove off-screen ones.

b. Render Graphics

- Draw the current game state on the screen:
    - Bird, pipes, background, ground, score, etc.

c. Check Collision

- Check for:
    - Bird hitting pipes or ground (collision).
    - Bird successfully passing pipes (no collision).

7. Collision Detected?

a. ✅ If No Collision

- Proceed to update the score if the bird passes between the pipes.
- Continue the game loop.

b. ❌ If Collision Detected

- Transition to the Game Over screen/state.
- Display final score.
- Play "hit" sound effect.
- Pause before moving to next step.

8. Restart Game or Exit

- Player is given two options:
    - Restart: Resets all variables, clears pipes, and returns to the main menu or directly restarts.
    - Exit: Ends the game.

9. Exit

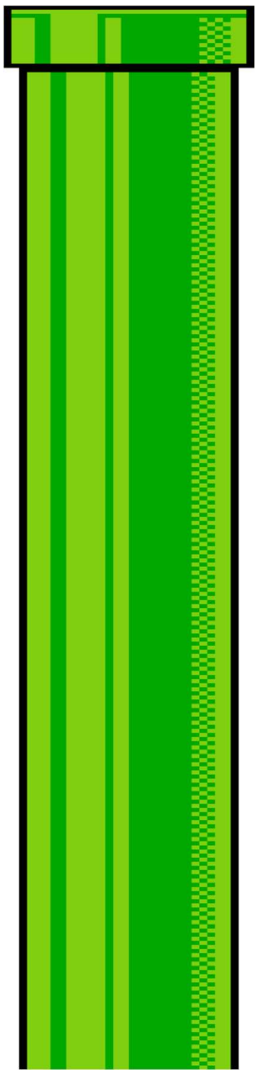
- Application shuts down.
- Resources are released.
- Game loop stops, and program terminates.

Summary Table

Step	Action
Start	Launch the game
Initialize Game	Set up variables and settings
Load Assets	Load images and sounds
Display Menu	Show title screen
Get Input	Start game on user action
Game Loop	Continually update game state
Check Collision	Determine if game ends
Game Over / Score Update	Respond to game status
Restart / Exit	Allow player to replay or quit
End	Close the application

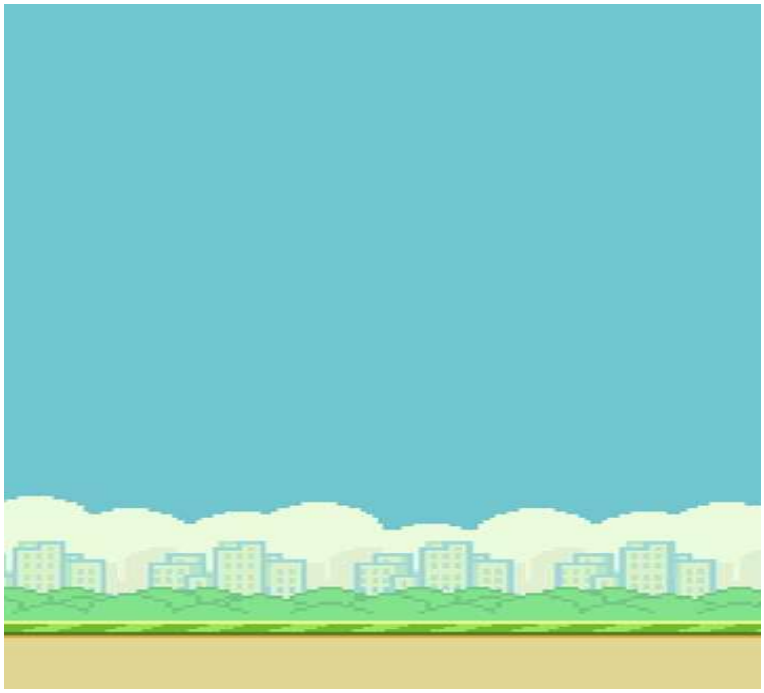
PNG IMAGES USED FOR CREATE BIRD AND BACKGROUND OF GAME

1. Top Pipe

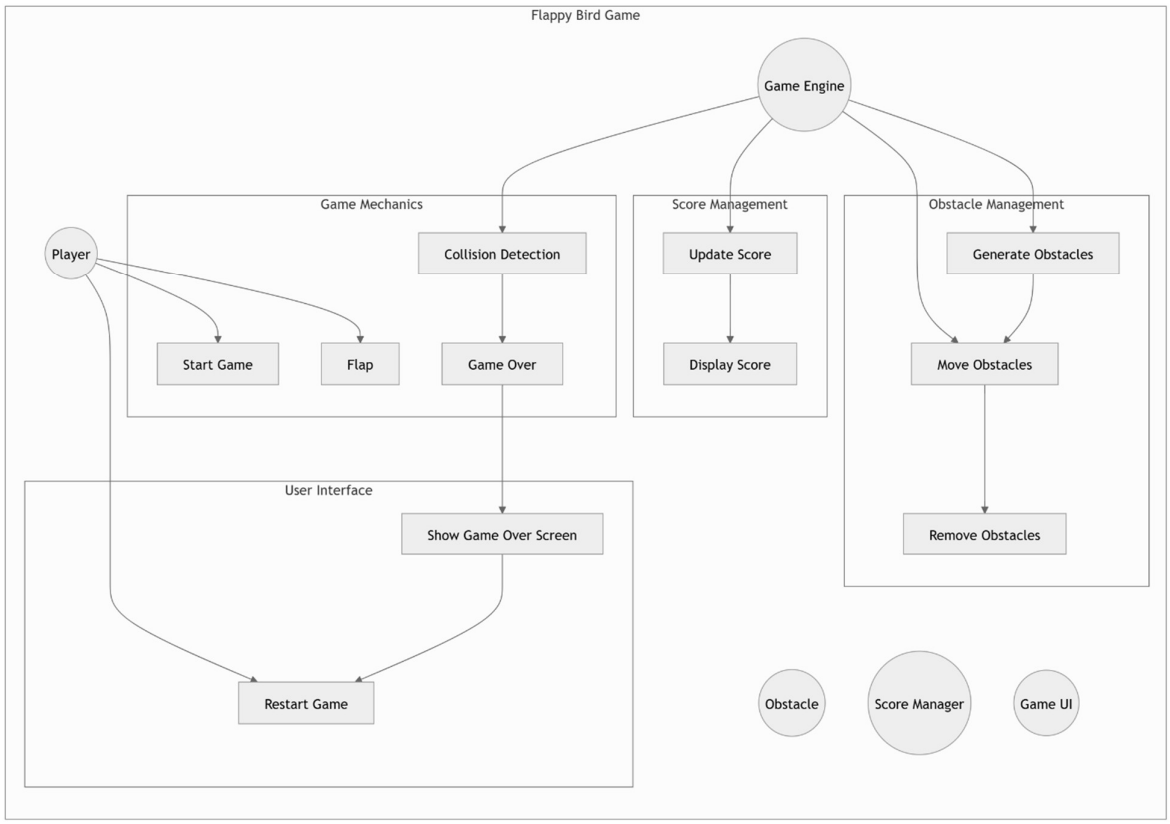


2. Flappy Bird

3. background image of Flappy bird game



Use – Case Diagram



Central Controller: Game Engine

- The Game Engine is the core system that manages all interactions in the game.
- It connects and controls:
    - Game Mechanics
    - Score Management
    - Obstacle Management
    - User Interface

1. Game Mechanics

This section handles all the core gameplay actions and rules, especially how the player interacts with the game.

Player

- The user controls the bird character.

Start Game

- Triggered when the player begins playing (e.g., presses Space or taps the screen).
- Initializes variables and starts the game loop.

Flap

- Every time the user presses the flap key, the bird moves upward (flap action).
- Gravity will eventually pull the bird down unless flapped again.

Collision Detection

- Constantly checks whether the bird collides with obstacles (pipes) or the ground.
- On collision, it triggers the Game Over sequence.

Game Over

- Stops gameplay.
- Sends signals to update the UI and stop obstacle movement.

2. Score Management

Handles how the player's performance is tracked and displayed.

Update Score

- Each time the bird passes a set of pipes successfully, the score is incremented.

Display Score

- Shows the current score on the screen during gameplay.
- May also show high scores or final score after game over.

3. Obstacle Management

Responsible for the behavior of the pipes (obstacles).

Generate Obstacles

- Creates pipe objects at regular intervals.

- Pipes have randomized heights to increase difficulty.

Move Obstacles

- Pipes scroll from right to left, simulating forward movement.

Remove Obstacles

- Once a pipe moves off-screen, it is removed to free up memory and resources.

4. User Interface (UI)

Handles all the visual feedback and interactions apart from the game's physical mechanics.

Show Game Over Screen

- Displays when the player loses.
- Shows the score and options to restart.

Restart Game

- Allows the user to replay the game.
- Resets game variables, obstacles, and score.

Flow Summary

1. Player starts the game and controls the bird (flap).
2. The Game Engine manages:

Bird movement

    - Obstacle creation/movement/removal
    - Score updating
    - Collision detection

3. If collision occurs, the game shows the Game Over screen.
4. The user can then restart the game, looping back into gameplay.

Chapter 6

Project Outcome

1. Optimized and Well-Structured Java Code

Clean, Modular Architecture

- The game will follow Object-Oriented Programming (OOP) principles.
- Major components will be encapsulated into separate classes, improving readability and reusability. For example:
    - Bird.java – handles bird movement, position, and rendering.
    - Pipe.java – manages the individual pipe's position and collision logic.
    - GamePanel.java – acts as the main game loop and rendering surface.
    - CollisionManager.java – detects and processes collisions.
    - ScoreManager.java – tracks score, high scores, and display logic.

Optimization Techniques

- Object Pooling: Reuse off-screen pipes instead of creating new ones. This saves memory and reduces garbage collection overhead.
- Minimized Redundant Calculations: For example, precalculating pipe gaps and positions or caching frequently used values like bird's gravity and lift.
- Separation of Concerns: Game logic, rendering, and input handling will be separated, making it easier to maintain and scale the code.

Documentation

- Each class and method will be properly commented using JavaDoc-style documentation.
- Descriptive variable names and structured flow will help others (or future you) understand the code effortlessly.

2. Smooth Animation and Physics-Based Movement

Realistic Bird Movement

- The bird's vertical motion will simulate basic physics:

- Gravity: Pulls the bird downward over time.
- Lift: When the player presses the key/taps, an upward force is applied.
- The bird will naturally arc upward and then fall if not flapped again, mimicking the feel of flying.

Frame-Based Animation

- A consistent frame rate (e.g., 60 FPS) will ensure:
  - Smooth movement of the bird and pipes.
  - Proper animation of background and game elements.

Game Loop Logic

- The game loop will follow the pattern:

ensuring timely updates and fluid graphics rendering without lag.

3. Accurate Collision Detection System

Hitbox-Based Detection

- Every object (bird, pipe, ground) will have a bounding box (e.g., Rectangle).
- Collisions are checked by testing intersections between rectangles.

Fairness and Accuracy

- The detection will be pixel-aligned to prevent unfair “false positives” (e.g., touching the edge of a pipe) or “false negatives” (flying through without registering).
- Optionally, pixel-perfect collision detection can be implemented for better accuracy using image masks.

Collision Events

- On collision:
  - Game stops.
  - The score is finalized.
  - The UI transitions to the game over screen.

4. Engaging User Interface

Main UI Components

- Start Screen: Simple screen with a play button and title.
- Pause Button: Optionally allows players to pause/resume the game.
- Game Over Screen: Displays final score and a replay button.

Visuals and Feedback

- Backgrounds, buttons, and icons will be:
    - Clean and minimalistic so they don't distract from gameplay.
    - Consistently themed for a cohesive look.

User Experience (UX)

- All screens will transition smoothly.
- Sound effects and animations (e.g., flapping, score pop-ups) enhance interactivity and immersion.

5. Scoring Mechanism and Replay ability

Scoring System

- The score increases every time the bird passes between pipes.
- Pipes will have a scoring zone that triggers once per pipe set.

High Score Tracking

- The game will store and display the highest score achieved during the session or across sessions (if saved using file or database).

Increasing Challenge

- As time passes, the pipe speed may gradually increase, or gaps may shrink, adding difficulty.
- This keeps the player challenged and improves replay value.

Chapter 7
User Interface

The user interface (UI) of the game is designed to be simple yet visually engaging, ensuring a smooth and enjoyable player experience. Here's a more detailed breakdown of the UI components:

1. Game Screen (Main Gameplay Interface)

This is where the player spends the majority of the time, so it needs to be both functional and visually appealing.

a. Bird Character

- Visual Style:
    - The bird needs to be distinct and easy to spot against the background, as it's the player's primary focus. Typically, the bird is a small, colorful character (like a yellow bird with big eyes) that stands out against the sky, pipes, and other obstacles.
    - It should also have a clean outline or shadow to separate it from similar-colored backgrounds (e.g., light blue sky).
- Animation:
    - Flapping Wings: The bird's wings should flap at a consistent rate when the player taps, with the bird's body following a fluid arc. This gives a sense of lifelike motion and enhances the player's connection to the game.
    - Rotation/Tilt: The bird rotates slightly, tilting upwards when the player presses for a flap, and tilting downward when gravity takes over, mimicking the effect of falling. It should feel natural and responsive.
    - Feedback: Additional animations such as a subtle bounce when the bird flaps can add to the immersion. A simple trail effect, like a puff of air or a light streak behind the bird, can also make it feel more dynamic.

b. Moving Pipes (Obstacles)

- Design:

- The pipes are often designed as green vertical tubes, which serve as barriers for the bird. There are two pipes (one at the top and one at the bottom) with a gap in between, which the bird must pass through.
- The pipes should appear in a variety of positions along the Y-axis, randomly generated, to ensure variability in gameplay difficulty. This randomness increases the challenge and ensures no two runs are the same.

- Movement:
  - The pipes continuously scroll from right to left, creating a "forward" moving sensation as the player's bird flies towards them. The speed of the pipes can either remain constant or increase over time, depending on how hard you want the game to get as the player progresses.
  - Adjusting the speed and frequency of pipes adds a dynamic challenge, keeping players on their toes.
- Challenge:
  - The pipes must be positioned in such a way that the player must carefully navigate the gap between them. The size of the gap and the frequency of pipes can be dynamically adjusted based on the player's progress, allowing for smoother difficulty scaling.

c. Background

- Types:
  - Day Theme: This is the most common theme, featuring a clear blue sky, fluffy white clouds, and distant hills or mountains. Bright colors are typically used to give a cheerful, upbeat atmosphere.
  - Night Theme: A darker sky with stars, a moon, and possibly a darker landscape that creates a nighttime environment. The background color shifts to deeper tones (blues or purples), giving the game a more serene or mysterious feeling.
- Behavior:
  - Parallax Scrolling: To create the illusion of depth, the background can move slightly slower than the foreground (the bird and pipes). This adds a more sophisticated visual effect without distracting the player.

- Subtlety: The background shouldn't steal the spotlight from the gameplay. It's designed to enhance the experience but should remain subtle enough not to interfere with the bird's movement and the pipes.

d. Score Counter

- Location:
  - The score counter should be placed in the center-top of the screen for easy visibility. It should be a static element that doesn't move around to maintain consistency during gameplay.
- Design:
  - Use a large, bold font with high contrast, like white or yellow against a blue sky, to ensure readability.
  - A smooth animation when the score increases (e.g., a quick pop-up effect or bounce) can make the achievement of scoring feel more rewarding.
- Real-Time Update:
  - The score increases every time the bird successfully passes a set of pipes. Consider adding sound effects (like a ding or soft chime) when the score increases to reinforce the positive feedback loop.

2. Game Over Screen

When the bird collides with a pipe or the ground, the game enters this screen to end the session and give the player options.

a. Final Score Display

- Placement: The final score should be prominently displayed at the center of the screen for clarity.
- Additional Information: Optionally, you can show the highest score achieved so far to encourage players to try again and improve.

b. Restart Button

- Function: The restart button should immediately reset the game variables (bird position, score, pipes, etc.) and bring the player back to the gameplay screen. It provides quick, frictionless access to starting over.
- Design:

- The button should be large and clickable, with a label like “Try Again” or “Restart”. A subtle bounce effect when hovered over or clicked will give the button a responsive feel.

c. Exit Option

- This option should allow players to exit the game or return to the main menu (which could include options like Start Game, Exit, or Settings).
- To prevent accidental exits, you could implement a confirmation pop-up asking the player if they're sure they want to quit.

3. Additional UI Features

a. Smooth Transitions

- Game Start: When the game begins, use a fade-in transition from the main menu to the gameplay screen to create a smooth entry.
- Game Over: Similarly, when the game ends, a fade-out and fade-in transition into the Game Over screen helps maintain a smooth flow between the game's states.
- Restart: When the player clicks Restart, a quick fade transition back into gameplay (or a brief slide effect) can make the reset process feel fluid.

b. Sound Effects & Feedback

- Actions: Adding sound effects like a wing flap sound for each jump, a scoring chime for every successful pipe pass, and a crash sound for collisions enhances the gameplay experience. These feedback sounds create a strong sense of interaction and immersion.
- Visuals:
    - Adding visual effects like flashing or shaking the screen when a collision occurs gives additional feedback to the player that something has happened.
    - Particles, like feathers or sparks, can emphasize actions like flapping or collisions, making the game feel more dynamic and visually engaging.

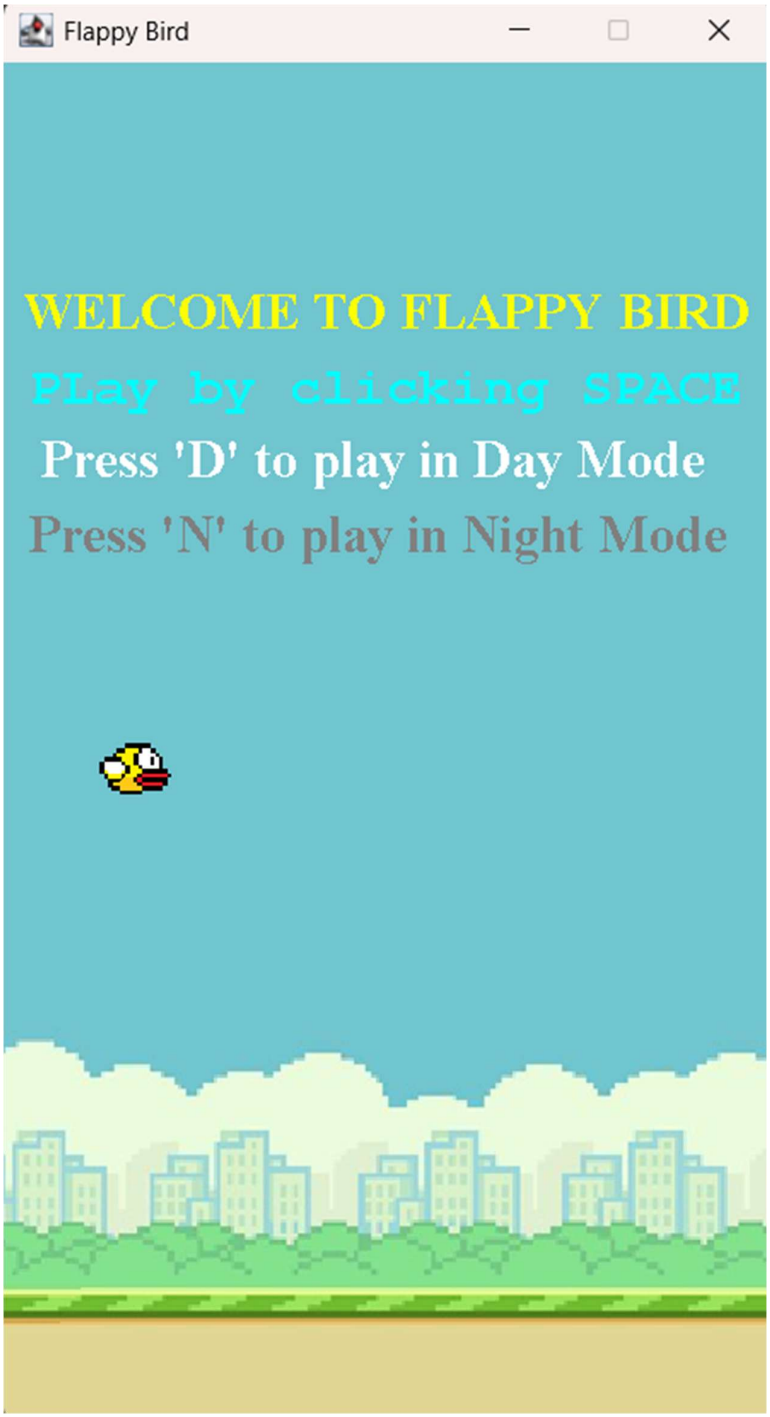
c. Minimalist and Intuitive Layout

- The UI layout should be clean and uncluttered. Buttons should be clearly visible and easy to interact with, but not oversized to avoid overcrowding the screen. The design should feel balanced.
- Icons: Any icons used should be intuitive and meaningful, immediately recognizable to the player (e.g., a gear icon for settings or a home icon for the main menu).

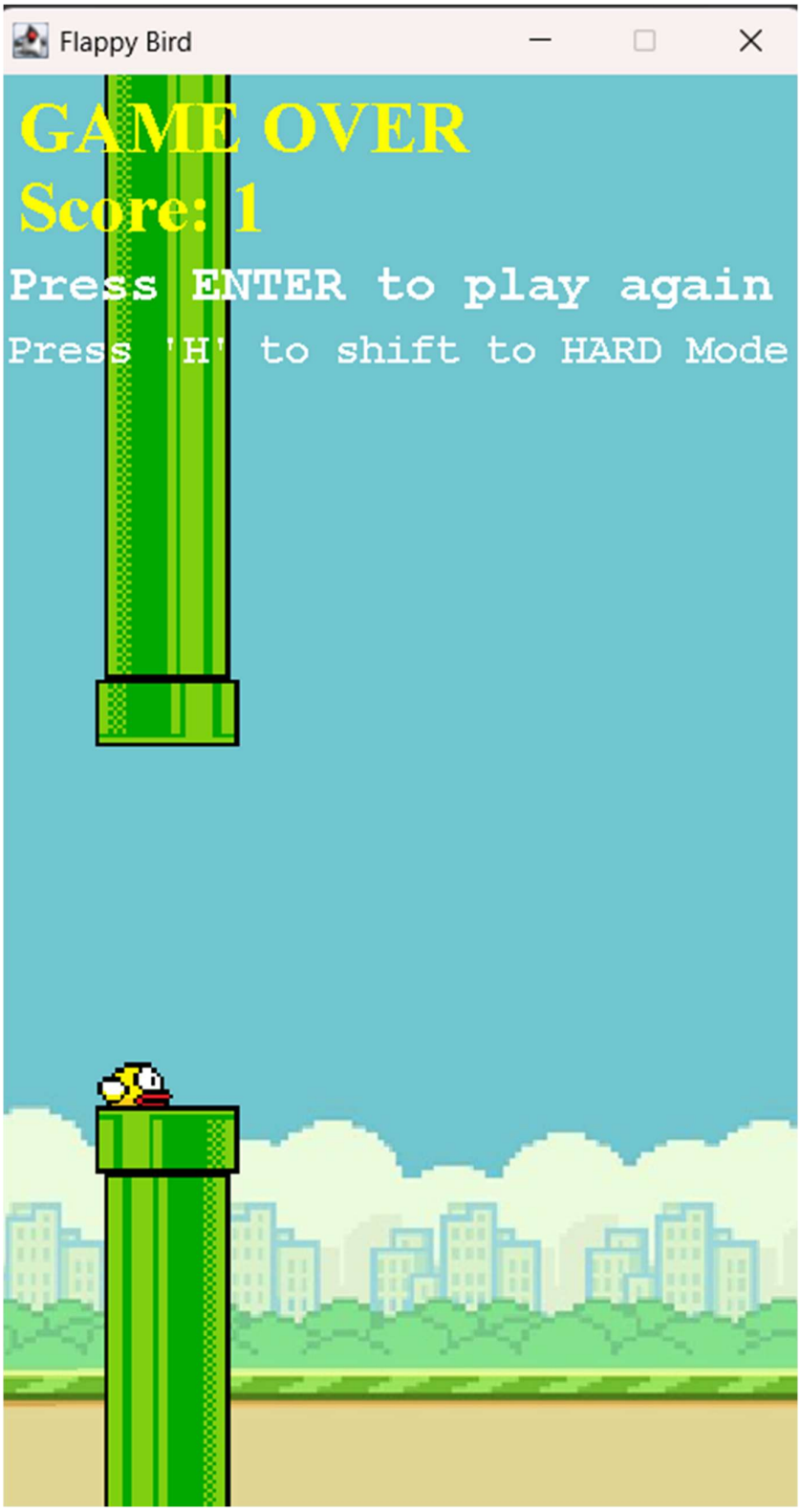
- Responsive UI: Make sure the UI adapts smoothly to different screen sizes and resolutions to accommodate a variety of devices (smartphones, tablets, desktops).

Screenshots of Flappy Bird Game

1. start display



2. day mode



3. Night mode


Flappy Bird
GAME OVER
Score: 0
Press ENTER to play again
Press 'H' to shift to HARD Mode

Chapter 8

References

- Java Documentation: https://docs.oracle.com/en/java/
- Game Development Concepts: https://www.geeksforgeeks.org/game-development-java/
- Flappy Bird Concept Reference: https://en.wikipedia.org/wiki/Flappy_Bird