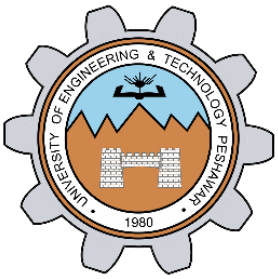
OBJECT ORIENTED PROGRAMMING LAB-PROJECT

DOODLE JUMP GAME
IN C++ USING SFML LIBRARY



FALL 2024-25

CSE208L OBJECT ORIENTED PROGRAMMING LAB

Submitted by:
SYED MUHAMMAD NOMAN (23PWCSE2227)

MUHAMMAD AFNAN KHAN (23PWCSE2241)

MUHAMMAD SAAD (23PWCSE2316)

Class Section: **C**

Group: **13**

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

Submitted to:

Engr. Sumayyea Salahuddin

January 27th, 2025

DEPARTMENT OF COMPUTER SYSTEMS ENGINEERING

UNIVERSITY OF ENGINEERING AND TECHNOLOGY, PESHAWAR

OBJECT ORIENTED PROGRAMMING LAB PROJECT
DOODLE JUMP GAME IN C++ USING SFML LIBRARY

INTRODUCTION:

Brief Overview:

- This project implements a simplified version of the classic Doodle Jump game using C++ and SFML (Simple and Fast Multimedia Library). It demonstrates the core principles of Object-Oriented Programming (OOP), such as encapsulation, inheritance, polymorphism, and abstraction.

Purpose:

- To learn OOP concepts.
- To explore game development.
- To apply SFML for rendering graphics and handling input.

TOOLS AND TECHNOLOGIES:

- **Programming Language**: C++
- **Graphics Library**: SFML
- **Editor/IDE**: DevC++, CodeBlocks.

GAME OBJECTIVES:

Goal of the Game:

- The player controls a character that continuously jumps off platforms to climb as high as possible, avoiding falling off the screen. The goal is to score points by reaching higher platforms.

Challenges:

- Players must navigate left and right to land on platforms.
- Platforms reset when the player moves higher, creating a dynamic environment.

FEATURES:

Game Features:

- Character movement (left and right).
- Jump mechanics with gravity simulation.
- Platforms that reset and move dynamically.
- Score tracking.
- Game Over and Retry options.

Additional Features:

- Custom character and background.
- Sound effects for player jumps.
- Responsive design for a larger window size.

CHALLENGES FACED:

Issues during implementation:

- Integrating SFML for the first time was challenging.
- Managing character movement and collision detection required precise calculations.
- Finding optimal images for the background and character in order to maintain optimal color combination.
- Resizing and conversion of the images from one form to another (e.g jpg to png etc.).
- Adjusting the Game window as well as adjusting platform and character spawn accordingly at the start as well as on resetting the game.

How we overcame these:

- Researched helping materials online.
- Revised OOP concepts for better understanding, implementation and design.
- Used the help of online websites for resizing and conversion of images according to our needs.
- Used AI tools for help like ChatGPT.

REFERENCE:

We used the following video, provided to us by our instructor, as reference:
https://www.youtube.com/watch?v=7Vf_vQIUk5Q&ab_channel=FamTrinli

WORKING OF THE GAME:

GAME OVERVIEW:

- The game consists of a player character that automatically falls due to gravity and jumps when landing on platforms.
- The player moves horizontally via keyboard inputs (Left and Right arrow keys).
- The objective is to keep jumping on platforms, avoid falling off the screen, and earn points as platforms reset (when they move off-screen).

OBJECT ORIENTED PROGRAMMING LAB-PROJECT

KEY COMPONENTS:

Game Classes:

We have used a total of 4 classes:

1. GameObject (Base Class).
2. Player (Derived Class).
3. Platform (Derived Class).
4. Game.

- **GameObject (Base):**

```
// Base class for all game objects
class GameObject {
protected:
    Vector2f position;

public:
    GameObject(float x, float y) : position(x, y) {}

    virtual void update() = 0; // Pure virtual function for
polymorphism
    virtual void draw(RenderWindow& window, Sprite& sprite) = 0;

    Vector2f getPosition() const {
        return position;
    }

    void setPosition(float x, float y) {
        position = {x, y};
    }
};
```

- **Player (Derived):**

The Player class defines the character and its behavior.
It controls the player's position, movement (left/right), gravity, and jumping mechanics.
Applies a vertical velocity (dy) to simulate falling and jumping.

```
class Player : public GameObject {
private:
    float dx, dy; // Horizontal and vertical velocity

public:
    Player() : GameObject(200, 200), dx(0.0f), dy(0.0f) {} // Initialize
position and velocity

    void moveLeft() { position.x -= 5.0f; } // Move left by 5 units
    void moveRight() { position.x += 5.0f; } // Move right by 5 units

    void applyGravity() {
        dy += 0.2f;          // Simulate gravity (increase vertical velocity)
        position.y += dy;    // Move the player down
    }
```

```
    void jump() { dy = -8.0f; } // Give upward velocity for a jump

    void reset() {
        position = {200, 200}; // Reset position to start
        dx = 0;
        dy = 0;
    }

    float getVelocityY() const { return dy; } // Return vertical velocity

    void update() override {
        applyGravity(); // Apply gravity each frame
    }

    void draw(RenderWindow& window, Sprite& sprite) override {
        sprite.setPosition(position); // Update sprite position
        window.draw(sprite);          // Draw the sprite
    }
};
```

- **Gravity Simulation**: The `applyGravity()` method increases the vertical velocity (`dy`) and moves the player down.
- **Jump**: When the player lands on a platform, `jump()` gives the player an upward velocity (`dy = -8.0f`).

- **Platform (Derived):**

  The `Platform` class represents the platforms the player jumps on. It inherits from `GameObject` and provides a simple vertical movement method.
  Moves vertically when the player jumps past a certain threshold.

```
class Platform : public GameObject {
public:
    Platform(float x, float y) : GameObject(x, y) {} // Initialize position

    void update() override {
        // Platforms don't have specific updates (currently empty)
    }

    void move(float dy) {
        position.y -= dy; // Move platform vertically (negative = up)
    }

    void draw(RenderWindow& window, Sprite& sprite) override {
        sprite.setPosition(position); // Set sprite position
        window.draw(sprite);          // Draw platform sprite
    }
};
```

- Platforms move downward relative to the player's vertical movement (`move(float dy)`).
- They reset to the top when they move off-screen.

- **Game (Class):**

```
class Game {
private:
    RenderWindow window;
    Texture t1, t2, t3;
    Sprite sBackground, sPlat, sPers;
    Font font;
    Text scoreText, gameOverText, retryText;

    Player player;
    std::vector<Platform> platforms;
    int score;
    bool gameOver;

    // Sound-related variables
    SoundBuffer jumpBuffer;
    Sound jumpSound;

    void resetGame() {
        player.reset();
        score = 0;
        gameOver = false;

        platforms.clear();
        for (int i = 0; i < 10; i++) {
            platforms.emplace_back(rand() % 500, rand() % 700);
        }   // Adjusted spawn range for platforms

        updateScoreText();
    }

    void updateScoreText() {
        std::ostringstream ss;
        ss << "Score: " << score;
        scoreText.setString(ss.str());
    }

    void handleInput() {
        if (Keyboard::isKeyPressed(Keyboard::Left)) {
            player.moveLeft();
        }
        if (Keyboard::isKeyPressed(Keyboard::Right)) {
            player.moveRight();
        }
    }

    void handleCollisions() {
        for (auto& platform : platforms) {
            Vector2f pPos = platform.getPosition();
            Vector2f plPos = player.getPosition();

     if ((plPos.x + 50 > pPos.x) && (plPos.x + 20 < pPos.x + 68) &&
       (plPos.y + 70 > pPos.y) && (plPos.y + 70 < pPos.y + 14) &&
                (player.getVelocityY() > 0)) {
                player.jump(); // Boost the player up when they collide
// with the platform
```

```
                jumpSound.play();      // Play the jump sound
            }
        }
    }

    void updatePlatforms() {
        for (auto& platform : platforms) {
            platform.move(player.getVelocityY());

            // Check if platform is off-screen and reset it
            if (platform.getPosition().y > 700) { // Adjusted for new
window height
                platform.setPosition(rand() % 500, 0); // Adjusted for
new window width
                score++; // Increment score when a platform resets
            }
        }

        updateScoreText(); // Update score text after platform movement
    }

    void draw() {
        window.clear();

        if (gameOver) {
            window.draw(sBackground);
            window.draw(gameOverText);
            window.draw(retryText);
        } else {
            window.draw(sBackground);
            player.draw(window, sPers);
            for (auto& platform : platforms) {
                platform.draw(window, sPlat);
            }
            window.draw(scoreText);
        }

        window.display();
    }

public:                         // Adjusted window size
    Game() : window(VideoMode(500, 700), "Doodle Game!"), score(0),
gameOver(false) {
        srand(time(0));
        window.setFramerateLimit(60);

        // Load textures and font
        t1.loadFromFile("images/sea.png");
        t2.loadFromFile("images/platform.png");
        t3.loadFromFile("images/character.png");

        sBackground.setTexture(t1);
        sPlat.setTexture(t2);
        sPers.setTexture(t3);

        if (!font.loadFromFile("fonts/DoodleJumpBold_v2.ttf")) {
            std::cerr << "Failed to load font" << std::endl;
```

```
            return;
        }

        // Load sound
        if (!jumpBuffer.loadFromFile("sounds/sound_jump.wav")) {
            std::cerr << "Failed to load sound_jump.wav" << std::endl;
            return;
        }
        jumpSound.setBuffer(jumpBuffer);

        scoreText.setFont(font);
        scoreText.setCharacterSize(30);
        scoreText.setFillColor(Color::White);
        scoreText.setPosition(10, 10);

        gameOverText.setFont(font);
        gameOverText.setCharacterSize(48);
        gameOverText.setFillColor(Color::Red);
        gameOverText.setString("Game Over!");
        gameOverText.setPosition(150, 300);

        retryText.setFont(font);
        retryText.setCharacterSize(28);
        retryText.setFillColor(Color::Black);
        retryText.setString("Press R to Retry or Esc to Exit");
        retryText.setPosition(120, 400);

        resetGame();
    }

    void run() {
        while (window.isOpen()) {
            Event e;
            while (window.pollEvent(e)) {
                if (e.type == Event::Closed) {
                    window.close();
                }

                if (gameOver) {
                    if (Keyboard::isKeyPressed(Keyboard::R)) {
                        resetGame();
                    } else if
(Keyboard::isKeyPressed(Keyboard::Escape)) {
                        window.close();
                    }
                }
            }

            if (!gameOver) {
                handleInput();
                player.update();

                if (player.getPosition().y > 700) { // Adjusted for new
window height
                    gameOver = true;
                }
```

```
                if (player.getPosition().y < 300) { // Adjusted gravity
offset for larger window
                    float offset = 300 - player.getPosition().y;
                    player.setPosition(player.getPosition().x, 300);
                    for (auto& platform : platforms) {
                        platform.move(-offset);
                    }
                }

                updatePlatforms();
                handleCollisions();
            }

            draw();
        }
    }
};
```

Game Logic:

- **Gravity:**

  Applied to the player via Player::applyGravity(), causing continuous downward motion (dy).

```
void applyGravity() {
        dy += 0.2f; // Increase gravity
        position.y += dy;
    }
```

- **Collision Detection**:

  Checks if the player's position overlaps with a platform. If true, the player is "boosted" upwards with a negative dy (jump).

```
void handleCollisions() {

        for (auto& platform : platforms) {

            Vector2f pPos = platform.getPosition();

            Vector2f plPos = player.getPosition();

     if ((plPos.x + 50 > pPos.x) && (plPos.x + 20 < pPos.x + 68) &&

       (plPos.y + 70 > pPos.y) && (plPos.y + 70 < pPos.y + 14) &&

                (player.getVelocityY() > 0)) {

                player.jump(); // Boost the player up when they collide
with the platform
```

```
                jumpSound.play();      // Play the jump sound
            }
        }
    }
```

- **Score**:

  Increases when a platform moves off the bottom of the screen and is repositioned at the top.

```
void updateScoreText() {
        std::ostringstream ss;
        ss << "Score: " << score;
        scoreText.setString(ss.str());
    }
```

- **Game Over**:

  Triggered when the player falls below the screen (y > 700).

```
void run() {
        while (window.isOpen()) {
            Event e;
            while (window.pollEvent(e)) {
                if (e.type == Event::Closed) {
                    window.close();
                }
                if (gameOver) {
                    if (Keyboard::isKeyPressed(Keyboard::R)) {
                        resetGame();
                    } else if
(Keyboard::isKeyPressed(Keyboard::Escape)) {
```

```
                        window.close();
                    }
                }
            }
            if (!gameOver) {
                handleInput();
                player.update();
    if (player.getPosition().y > 700) { // Adjusted for new window height
                    gameOver = true;
                }
     if (player.getPosition().y < 300) { // Adjusted gravity offset for
                                larger window
                    float offset = 300 - player.getPosition().y;
                    player.setPosition(player.getPosition().x, 300);
                    for (auto& platform : platforms) {
                        platform.move(-offset);
                    }
                }
                updatePlatforms();
                handleCollisions();
            }
            draw();
        }
    }
```

GRAPHICS AND SOUND:

- Textures are used for the background, platforms, and player sprite.

```
// Load textures and font
        t1.loadFromFile("images/sea.png");
        t2.loadFromFile("images/platform.png");
        t3.loadFromFile("images/character.png");

        sBackground.setTexture(t1);
        sPlat.setTexture(t2);
        sPers.setTexture(t3);
```

- A jump sound effect (sound_jump.wav) plays whenever the player jumps.
  Sound Variables:

```
// Sound-related variables
    SoundBuffer jumpBuffer;
    Sound jumpSound;

// Load sound
        if (!jumpBuffer.loadFromFile("sounds/sound_jump.wav")) {
            std::cerr << "Failed to load sound_jump.wav" << std::endl;
            return;
        }
jumpSound.play();      // Play the jump sound
```

- Text objects (scoreText, gameOverText, retryText) display relevant information like the score and game state.
  Score Text:

```
scoreText.setFont(font);
        scoreText.setCharacterSize(30);
        scoreText.setFillColor(Color::White);
        scoreText.setPosition(10, 10);
```

Game Over Text:

```
gameOverText.setFont(font);

  gameOverText.setCharacterSize(48);

  gameOverText.setFillColor(Color::Red);

  gameOverText.setString("Game Over!");

  gameOverText.setPosition(150, 300);
```

Retry Text:

```
retryText.setFont(font);

  retryText.setCharacterSize(28);

  retryText.setFillColor(Color::Black);

  retryText.setString("Press R to Retry or Esc to Exit");
```

```
retryText.setPosition(120, 400);

resetGame();
```

GAME LOOP (MAIN FUNCTIONALITY):

- **Event Handling**:
  - Checks for Keyboard inputs to control the player or reset/exit the game after a Game Over.
    Player Control:

```
void handleInput() {

        if (Keyboard::isKeyPressed(Keyboard::Left)) {

            player.moveLeft();

        }

        if (Keyboard::isKeyPressed(Keyboard::Right)) {

            player.moveRight();

        }

    }
```

Reset/exit:

```
if (gameOver) {

                    if (Keyboard::isKeyPressed(Keyboard::R)) {

                        resetGame();

                    } else if
(Keyboard::isKeyPressed(Keyboard::Escape)) {

                        window.close();

                    }

                }
```

- **Game Updates**:
  - Player's movement and gravity are updated.

Player Movement:

```
void moveLeft() {
        position.x -= 5.0f;
    }
    void moveRight() {
        position.x += 5.0f;
    }
```

Gravity:

```
void applyGravity() {
        dy += 0.2f; // Increase gravity
        position.y += dy;
    }
void update() override {
        applyGravity();
    }
```

- Platforms move relative to the player's motion, and their positions are reset when they go off-screen.

```
void updatePlatforms() {
        for (auto& platform : platforms) {
            platform.move(player.getVelocityY());
            // Check if platform is off-screen and reset it
            if (platform.getPosition().y > 700) { // Adjusted for new window height
                platform.setPosition(rand() % 500, 0); // Adjusted for new window width
                score++; // Increment score when a platform resets
            }
```

```
        }
        updateScoreText(); // Update score text after platform movement
    }
```

- Collisions between the player and platforms are handled to trigger jumps.

```
void handleCollisions() {
        for (auto& platform : platforms) {
            Vector2f pPos = platform.getPosition();
            Vector2f plPos = player.getPosition();
     if ((plPos.x + 50 > pPos.x) && (plPos.x + 20 < pPos.x + 68) &&
       (plPos.y + 70 > pPos.y) && (plPos.y + 70 < pPos.y + 14) &&
                (player.getVelocityY() > 0)) {
               player.jump(); // Boost the player up when they collide
with the platform
                jumpSound.play();      // Play the jump sound
            }
        }
    }
```

- **Rendering**:
  - Clears the window and redraws all elements (background, player, platforms, and text).

```
void draw() {
        window.clear();
        if (gameOver) {
            window.draw(sBackground);
            window.draw(gameOverText);
            window.draw(retryText);
        } else {
```

```
            window.draw(sBackground);

            player.draw(window, sPers);

            for (auto& platform : platforms) {

                platform.draw(window, sPlat);

            }

            window.draw(scoreText);

        }

        window.display();

    }
```

OOP CONCEPTS IMPLEMENTED:

- **ENCAPSULATION:**
  Encapsulation is basically grouping different data types, methods and variables in a class, and implementing access specifiers for data hiding. Here is a general example of an encapsulated class:

```
// Base class for all game objects
class GameObject {
protected:
    Vector2f position;

public:
    GameObject(float x, float y) : position(x, y) {}

    virtual void update() = 0; // Pure virtual function for
polymorphism
    virtual void draw(RenderWindow& window, Sprite& sprite) = 0;

    Vector2f getPosition() const {
        return position;
    }

    void setPosition(float x, float y) {
        position = {x, y};
    }
};
```

- **ABSTRACTION:**
  Using Pure Virtual Function:

```
virtual void update() = 0; // Pure virtual function for polymorphism
virtual void draw(RenderWindow& window, Sprite& sprite) = 0;
```

- **POLYMORPHISM:**
  Examples Code:

```
void update() override {
        applyGravity();
    }

void draw(RenderWindow& window, Sprite& sprite) override {
        sprite.setPosition(position);
        window.draw(sprite);
    }
```

- **INHERITANCE:**
  Example Code of inheritance:

```
class Platform : public GameObject {
// body of class
}

class Player : public GameObject {
// body of class
}
```

OVERALL GAME CODE:

main.cpp

```cpp
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include <time.h>
#include <sstream>
#include <iostream>
#include <vector>

using namespace sf;

// Base class for all game objects
class GameObject {
protected:
    Vector2f position;

public:
    GameObject(float x, float y) : position(x, y) {}

    virtual void update() = 0; // Pure virtual function for polymorphism
    virtual void draw(RenderWindow& window, Sprite& sprite) = 0;

    Vector2f getPosition() const {
        return position;
    }

    void setPosition(float x, float y) {
        position = {x, y};
    }
};

```

```
class Platform : public GameObject {
public:
    Platform(float x, float y) : GameObject(x, y) {}

    void update() override {
        // Platforms only move vertically when the player jumps above a threshold
    }

    void move(float dy) {
        position.y -= dy; // Move platform based on player gravity
    }

    void draw(RenderWindow& window, Sprite& sprite) override {
        sprite.setPosition(position);
        window.draw(sprite);
    }
};

class Player : public GameObject {
private:
    float dx, dy;

public:
    Player() : GameObject(200, 200), dx(0.0f), dy(0.0f) {}

    void moveLeft() {
        position.x -= 5.0f;
    }

    void moveRight() {
        position.x += 5.0f;
    }

    void applyGravity() {
        dy += 0.2f; // Increase gravity
        position.y += dy;
    }

    void jump() {
        dy = -8.0f; // Boost player upwards when colliding with a platform
    }

    void reset() {
        position = {200, 200}; // Reset player position
        dx = 0;
        dy = 0;
    }

    float getVelocityY() const {
        return dy;
    }

    void update() override {
        applyGravity();
    }

    void draw(RenderWindow& window, Sprite& sprite) override {
        sprite.setPosition(position);
        window.draw(sprite);
    }
};
```

```
class Game {
private:
    RenderWindow window;
    Texture t1, t2, t3;
    Sprite sBackground, sPlat, sPers;
    Font font;
    Text scoreText, gameOverText, retryText;

    Player player;
    std::vector<Platform> platforms;
    int score;
    bool gameOver;

    // Sound-related variables
    SoundBuffer jumpBuffer;
    Sound jumpSound;

    void resetGame() {
        player.reset();
        score = 0;
        gameOver = false;

        platforms.clear();
        for (int i = 0; i < 10; i++) {
            platforms.emplace_back(rand() % 500, rand() % 700);
        }   // Adjusted spawn range for platforms

        updateScoreText();
    }
```

```
    void updateScoreText() {
        std::ostringstream ss;
        ss << "Score: " << score;
        scoreText.setString(ss.str());
    }

    void handleInput() {
        if (Keyboard::isKeyPressed(Keyboard::Left)) {
            player.moveLeft();
        }
        if (Keyboard::isKeyPressed(Keyboard::Right)) {
            player.moveRight();
        }
    }

    void handleCollisions() {
        for (auto& platform : platforms) {
            Vector2f pPos = platform.getPosition();
            Vector2f plPos = player.getPosition();

            if ((plPos.x + 50 > pPos.x) && (plPos.x + 20 < pPos.x + 68) &&
                (plPos.y + 70 > pPos.y) && (plPos.y + 70 < pPos.y + 14) &&
                (player.getVelocityY() > 0)) {
                player.jump(); // Boost the player up when they collide with the platform
                jumpSound.play();      // Play the jump sound
            }
        }
    }
```

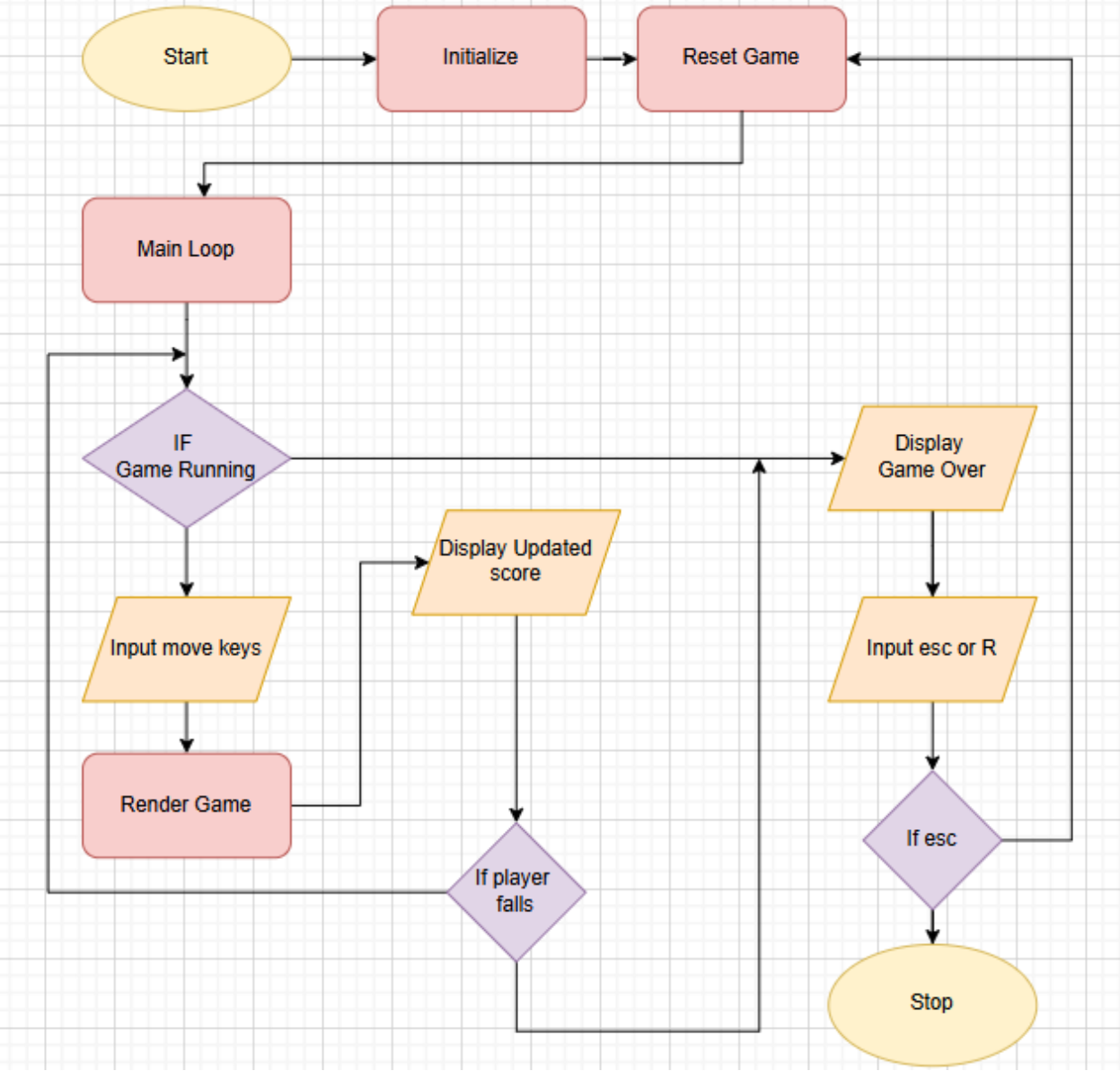
```
151     void updatePlatforms() {
152         for (auto& platform : platforms) {
153             platform.move(player.getVelocityY());
154 
155             // Check if platform is off-screen and reset it
156             if (platform.getPosition().y > 700) { // Adjusted for new window height
157                 platform.setPosition(rand() % 500, 0); // Adjusted for new window width
158                 score++; // Increment score when a platform resets
159             }
160         }
161 
162         updateScoreText(); // Update score text after platform movement
163     }
164 
165     void draw() {
166         window.clear();
167 
168         if (gameOver) {
169             window.draw(sBackground);
170             window.draw(gameOverText);
171             window.draw(retryText);
172         } else {
173             window.draw(sBackground);
174             player.draw(window, sPers);
175             for (auto& platform : platforms) {
176                 platform.draw(window, sPlat);
177             }
178             window.draw(scoreText);
179         }
180 
181         window.display();
182     }
183 
```

```
184 public:             // Adjusted window size
185     Game() : window(VideoMode(500, 700), "Doodle Game!"), score(0), gameOver(false) {
186         srand(time(0));
187         window.setFramerateLimit(60);
188 
189         // Load textures and font
190         t1.loadFromFile("images/sea.png");
191         t2.loadFromFile("images/platform.png");
192         t3.loadFromFile("images/character.png");
193 
194         sBackground.setTexture(t1);
195         sPlat.setTexture(t2);
196         sPers.setTexture(t3);
197 
198         if (!font.loadFromFile("fonts/DoodleJumpBold_v2.ttf")) {
199             std::cerr << "Failed to load font" << std::endl;
200             return;
201         }
202 
203         // Load sound
204         if (!jumpBuffer.loadFromFile("sounds/sound_jump.wav")) {
205             std::cerr << "Failed to load sound_jump.wav" << std::endl;
206             return;
207         }
208         jumpSound.setBuffer(jumpBuffer);
209 
210         scoreText.setFont(font);
211         scoreText.setCharacterSize(30);
212         scoreText.setFillColor(Color::Black);
213         scoreText.setPosition(10, 10);
```

```
        scoreText.setPosition(10, 10);

        gameOverText.setFont(font);
        gameOverText.setCharacterSize(48);
        gameOverText.setFillColor(Color::Red);
        gameOverText.setString("Game Over!");
        gameOverText.setPosition(150, 300);

        retryText.setFont(font);
        retryText.setCharacterSize(28);
        retryText.setFillColor(Color::Black);
        retryText.setString("Press R to Retry or Esc to Exit");
        retryText.setPosition(120, 400);

        resetGame();
    }

    void run() {
        while (window.isOpen()) {
            Event e;
            while (window.pollEvent(e)) {
                if (e.type == Event::Closed) {
                    window.close();
                }

                if (gameOver) {
                    if (Keyboard::isKeyPressed(Keyboard::R)) {
                        resetGame();
                    } else if (Keyboard::isKeyPressed(Keyboard::Escape)) {
                        window.close();
                    }
                }
            }
```

```
            if (!gameOver) {
                handleInput();
                player.update();

                if (player.getPosition().y > 700) { // Adjusted for new window height
                    gameOver = true;
                }

                if (player.getPosition().y < 300) { // Adjusted gravity offset for larger window
                    float offset = 300 - player.getPosition().y;
                    player.setPosition(player.getPosition().x, 300);
                    for (auto& platform : platforms) {
                        platform.move(-offset);
                    }
                }

                updatePlatforms();
                handleCollisions();
            }

            draw();
        }
    }
};

int main() {
    Game game;
    game.run();
    return 0;
}
```

SEQUENCE OF EVENTS IN THE FORM OF A FLOWCHART:



IMAGES USED:

CHARACTER:

PLATFORM:

OBJECT ORIENTED PROGRAMMING LAB-PROJECT

BACKGROUND:

IN-GAME OVERVIEW:

CONCLUSION:

- With this project we got introduced to a completely new library that is SFML.
- We learned how to use many different classes in SFML especially regarding the graphics.
- This project challenged us to implement the core concepts of Object Oriented Programming which we have learned in the lab.
- We got introduced to game development which is a very interesting and a vast field, and we learned a lot from it.
- Finally using the knowledge that we gained and the research that we did, and some help from online sources, we as a group were able to create our own version of the classic Doodle Jump Game.