Dino Runner – A 2D Endless-Runner Game using C++ and SFML

Employment Skill Development
Game using cpp
(BCSCC0301)

In
BACHELOR OF TECHNOLOGY
CSE(DATA SCIENCE) – Semester 3

SUBMITTED BY:

AMAN SINGH BHADORIYA (Roll No.401331540045)
ARMAAN PRAKASH (Roll No. 2401331540065)

Under the Supervision of

Mr. Ritesh Kumar Tripathi
Assistant professor
Training Department

Department of Computer Science & Engineering
NOIDA INSTITUTE OF ENGINEERING AND TECHNOLOGY,
GREATER NOIDA
(An Autonomous Institute)
Affiliated to
DR. APJ. ABDUL KALAM TECHNICAL UNIVERSITY, LUCKNOW

STUDENT'S DECLARATION

We hereby declare that the work presented in this report entitled **"Dino Runner – A 2D Endless-Runner Game using C++ and SFML"** was carried out by us. We have not submitted the matter embodied in this report for the award of any other degree or diploma of any other University or Institute. Due credit has been given to original authors/sources for all words, ideas, diagrams, graphics, computer programs, experiments, and results that are not our original contribution. We affirm that no portion of our work is plagiarized and the results are not manipulated..

Name : AMAN SINGH BHADORIYA

Roll Number : 2401331540045

(Candidate Signature)

Name : ARMAAN PRAKASH

Roll Number : 2401331540065

(Candidate Signature)

CERTIFICATE

This is to Certify that Aman Singh Bhadoriya (Roll No: 2401331540045), Armaan Prakash (Roll No: 2401331540065), have carried out the project work presented in this report entitled "Dino Runner – A 2D Endless-Runner Game using C++ and SFML" in partial fulfilment of the requirements for the award of the Bachelor of Technology in Computer Science & Engineering from Noida Institute Of Engineering And Technology affiliated from Dr. APJ Abdul Kalam Technical University, Lucknow under our supervision**.**

Signature

Mr. Ritesh Kumar Tripathi
Assistant Professor
Department Of Data Science NIET,
Greater Noida

Date:

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our guide, Mr. Ritesh Kumar Tripathi, for continuous guidance, feedback, and encouragement throughout this mini-project. We also thank HOD, CSE and Dy. HOD, CSE for their motivation and support. Finally, we acknowledge our peers and family members for their help in testing and improving the game.

Aman Singh Bhadoriya (2401331540045)
Armaan Prakash (2401331540065)
B.Tech in Data Science
Noida Institute Of Engineering And Technology

ABSTRACT

This mini-project presents Dino Runner, a lightweight 2D endless-runner game developed in C++ using the SFML multimedia library. The player controls a dinosaur that continuously runs on a scrolling ground and must jump or duck to avoid cacti and falling meteorites. The game also includes collectible power-ups such as Shield, Invincibility, and Speed Boost, each offering temporary advantages. The system features smooth sprite animations, accurate collision handling, dynamic difficulty levels, score and speed tracking, and clear audio feedback for every action. The project uses a clean, modular structure covering rendering, input handling, physics, UI design, sound management, and object-oriented programming. Overall, the game demonstrates how real-time updates, event handling, and game-loop logic can be combined to create an engaging desktop gaming experience.

TABLE OF CONTENTS

CHAPTER 1 – INTRODUCTION

1.1 Overview of the Project -

Endless-runner games are ideal for learning real-time rendering, timing, and collision mechanics. Using SFML, we built a desktop game that mirrors the fast feedback loop of popular runners: obstacles spawn, the ground scrolls, and the character's animation cycles synchronize with physics. Our implementation focuses on readable OOP structure and extendable gameplay systems

Key Components:-

- Game Loop & Timing: Fixed framerate (target 60 FPS) with sf::Clock based delta time.
- Rendering: Sprites, textures, and text using SFML modules.
- Input: Space for jump; Down for duck.
- Gameplay Systems: Obstacles, meteorites, power-ups, scoring, difficulty scaling, and restart..

1.2 Identified Issues / Gaps-

- Many student projects lack modular separation of concerns; we introduce distinct classes for each subsystem.
- Limited moment-to-moment variety in typical runners; we add power-ups and a size-growth mechanic from meteorite hits.
- Often absent telemetry/UI; we include FPS, level, speed, size, and power-up timers.

1.3 Objectives and Scope

- Build a stable, responsive 2D runner with clean OOP design.
- Implement collision handling with conditional effects (shield, invincibility).
- Design dynamic difficulty tied to score progression.
- Provide user feedback via sounds, HUD texts, and progress bars.
- Scope: Windows desktop using C++17 and SFML; single-player; keyboard input.

1.4 Project Report Organization

- Chapter 2 details requirements, feasibility, and planning.
- Chapter 3 explains architecture, SDLC choice, UML, and flow.
- Chapter 4 covers implementation highlights and results.
- Chapter 5 summarizes contributions and future enhancements.

CHAPTER 2 – REQUIREMENT AND ANALYSIS

2.1 Requirement Specification

Functional Requirements

1. Render ground, clouds, dino, obstacles, meteorites, power-ups.
2. Capture input (jump/duck).
3. Spawn obstacles/power-ups/meteorites at controlled intervals.
4. Detect collisions and apply effects.
5. Track score, level, speed, and power-up durations.
6. Restart game on death via button click or Space key.

Non-Functional Requirements

- Performance: Smooth gameplay at ~60 FPS on typical student laptops.
- Usability: Clear HUD and audio feedback.
- Maintainability: Modular classes; minimal coupling.
- Assets: Local textures, fonts, and audio files.

2.2 System Study / Feasibility

- Technical Feasibility: SFML provides simple APIs for windowing, graphics, input, and audio.
- Operational Feasibility: Keyboard-only controls; quick learning curve.
- Economic Feasibility: Free, open-source library; no runtime licensing.

2.3 Planning and Scheduling

- **Week 1–2:** Core loop, rendering, input, ground scroll.
- **Week 3:** Obstacles & collisions.
- **Week 4:** Power-ups & meteorites.
- **Week 5:** HUD, sounds, polish.
- **Week 6:** Testing, balancing, documentation.

2.4 Hardware and Software Requirements

- **Hardware:** Intel/AMD CPU, 8 GB RAM, iGPU/dGPU capable of OpenGL 2.1+, 200 MB disk.
- **Software:** Windows 10/11, C++17 compiler (MinGW/Clang/MSVC), **SFML 2.x**, CMake/Make (optional), code editor/IDE.

2.5 Preliminary Product Description

A single-window desktop game featuring a dinosaur character that runs on a scrolling ground. The player jumps/ducks to avoid cacti and falling meteorites, collects power-ups, and aims for a high score as speed and difficulty increase.

CHAPTER 3 – SYSTEM DESIGN

3.1 SDLC Model Used

Incremental (Agile-inspired): Features were built and tested in small iterations (rendering → obstacles → power-ups → HUD), enabling quick playtesting and balance tweaks.

3.2 System Architecture

Modules & Responsibilities

- GameState: Orchestrates subsystems, updates/draws everything, manages restart.
- Dino: Player sprite, animation, jump/duck physics, power-up states, collision bounds.
- Obstacles/Obstacle: Cacti, meteorites, and power-ups spawning/movement/collision.
- Ground: Infinite scroll via texture offset.
- Clouds: Parallax-like background movement.
- Scores: Score, difficulty, HUD texts, progress bars.
- SoundManager: Loads/plays jump, die, point sounds.
- Fps: FPS text overlay.
- RestartButton: Mouse bounds, click handling.

3.3 UML Diagrams (Descriptions)

- Use Case: *Play Game* (Start → Run → Jump/Duck → Collect Power-Up → Avoid Obstacles → View Score → Game Over → Restart).
- Class Diagram (Overview): Classes listed above with associations: `GameState` aggregates `Dino`, `Ground`, `Obstacles`, `Scores`, `Clouds`, `RestartButton`, `Fps`. `Obstacles` contains many `Obstacle`, `PowerUp`, and meteorite shapes.
- Activity Diagram: Game loop cycles: Poll events → Update subsystems → Check collisions → Render frame.
- Sequence Diagram (Main Loop): `main` → `GameState.update()` → (`Ground.update`, `Obstacles.update(dino)`, `Dino.update(obstacles)`, `Clouds.update`, `Scores.update()`) → `GameState.draw()`.

3.4 Flowchart of the System (Textual)

Start → Init Window & Assets → While Window Open:
Poll Events → If Close → Exit
Update: Ground | Obstacles (spawn/move/collide) | Dino (input/physics/collide) | Clouds | Scores/HUD
If Dead & Restart Triggered → Reset State
Render: Background → Ground → Obstacles/Power-Ups/Meteorites → Dino → HUD/FPS → Display.

CHAPTER 4 – IMPLEMENTATION AND RESULTS

4.1 System Architecture (Monolithic OOP Model)

The Dino Runner game follows a **monolithic object-oriented architecture**, where all major components are implemented as classes within a single executable program. The architecture includes the following main modules:

- **GameState** – Central controller that manages updates, rendering, and game reset.
- **Dino** – Handles player animation, movement, power-ups, physics, and collision logic.
- **Obstacles** – Manages cacti, meteorites, spawn timing, movement, and collision detection.
- **Ground** – Manages scrolling ground animation.
- **Clouds** – Handles background cloud movement.
- **Scores** – Tracks score, high score, level, speed, and updates HUD elements.
- **SoundManager** – Loads and controls all game audio.
- **RestartButton** – Detects mouse interaction and handles restart operations.

All classes communicate directly with **GameState**, making the structure monolithic but modular internally for easier maintenance and debugging.

4.2 Data Flow Diagram / Use Case Diagram

Use Case Diagram (Textual Description)

Actors:

- **Player**

Use Cases:

1. Start Game
2. Control Dino (Jump / Duck)
3. Avoid Obstacles
4. Collect Power-Ups
5. Increase Score
6. View Game Over Screen

7. Restart Game

Data Flow Explanation

- Player input → Dino movement → Collision checking → Score update → Final render.
- Internal game data (speed, timers, positions) flows between modules through the GameState class.

4.3 Class Diagram / Component Diagram

Major Classes (Text Description of UML)

1. **GameState**
   - Aggregates Dino, Ground, Obstacles, Clouds, Scores, RestartButton, and Fps.
   - Controls update and draw cycles.
2. **Dino**
   - Attributes: sprite, textures, position, motion, animation frames, power-up flags.
   - Methods: update(), walk(), reset().
3. **Obstacles**
   - Contains arrays of Obstacle, PowerUp, and Meteorite objects.
   - Methods: update(), draw(), reset().
4. **Scores**
   - Handles score, difficulty, text objects, and UI bars.
5. **SoundManager**
   - Loads jump, die, and point sounds.
6. **Ground**, **Clouds**, **RestartButton**, **Fps**
   - Each manages its own display and update behavior.

This forms a **component-based OOP structure**, but all components run within a single application module.

4.4 Algorithm and Flowchart

Main Game Algorithm

1. Start the game and load all textures, fonts, and audio assets.
2. Enter the main game loop:
    - Read input from keyboard/mouse.
    - Update Dino movement and animations.
    - Spawn and update obstacles, power-ups, and meteorites.
    - Perform collision detection.
    - Update HUD: score, power-up timers, levels, and speed.
    - Draw all elements to the window.
3. If collision ends the game → show Game Over screen.
4. If player presses Space or clicks Restart → reset all modules.
5. Continue game loop until window is closed.

Flowchart (Textual Representation)

Start → Load Assets → Create Window →

Game Loop Begin →

Read Input → Update Dino → Update Obstacles/Power-Ups → Update Score →

Collision Check →

If Dead? → Yes → Show Game Over → Restart? → Yes → Reset Game → Loop

Else → Continue Updating → Draw Frame →

Game Loop End → Exit.

4.5 Code Implementation Details

Technologies Used

- **C++17**
- **SFML 2.5+** (Graphics, Window, System, Audio modules)
- **MinGW/Make** for compilation

Key Implementations

- **Game Loop:** sf::RenderWindow with 60 FPS lock and delta-time updates.

- **Player Control:** Jump/duck handled via sf::Keyboard.
- **Animation:** Uses sprite sheet and frame counters for walking animation.
- **Collision:** Based on intersection of global bounds with slight boundary reduction.
- **Dynamic Difficulty:** Speed increases after every 500 points.
- **Power-Ups:** Shield, Invincibility, Speed Boost managed by timers and states.
- **Restart Logic:** Resets all modules including obstacles, speed, power-ups, and score.

4.6 Module Description

1. Dino Module

- Manages player sprite, motion, animation, power-ups, and collisions.
- Contains logic for jump, duck, gravity, and size growth/shrink.

2. Obstacles Module

- Spawns cacti at regular intervals.
- Generates meteorites with diagonal falling motion.
- Handles collision with player and deletes off-screen objects.

3. Power-Up Module

- Generates Shield, Invincibility, and Speed Boost items.
- Applies effects to Dino on collision.

4. Ground Module

- Creates continuous scrolling ground using texture offsets.

5. Clouds Module

- Moves background clouds for a parallax effect.

6. Score Module

- Calculates score, difficulty, speed, and displays them on HUD.
- Renders power-up bars and Dino size indicator.

7. Audio Module

- Plays jump, death, and score sounds.

8. Restart Module

- Displays restart button.
- Detects click or Spacebar press to restart game

CHAPTER 5 – TESTING AND RESULTS

5.1 Test Plan and Test Cases

The testing phase focused on evaluating the game's functionality, stability, user interaction, and real-time responsiveness. Both **manual testing** and **scenario-based testing** were performed to ensure that all modules work as expected.

5.1.1 Test Plan Objectives

- Verify correct functioning of player controls.
- Ensure obstacles, meteorites, and power-ups spawn accurately.
- Test collision detection and game-over mechanics.
- Validate scoring, difficulty scaling, and HUD updates.
- Check performance at 60 FPS under normal load.
- Confirm proper reset and restart functionality.

5.1.2 Test Environment

- **OS:** Windows 10/11
- **Compiler:** MinGW / MSVC
- **Framework:** SFML 2.5+
- **Hardware:** 8 GB RAM, Intel i5/Ryzen CPU

5.1.3 Test Cases

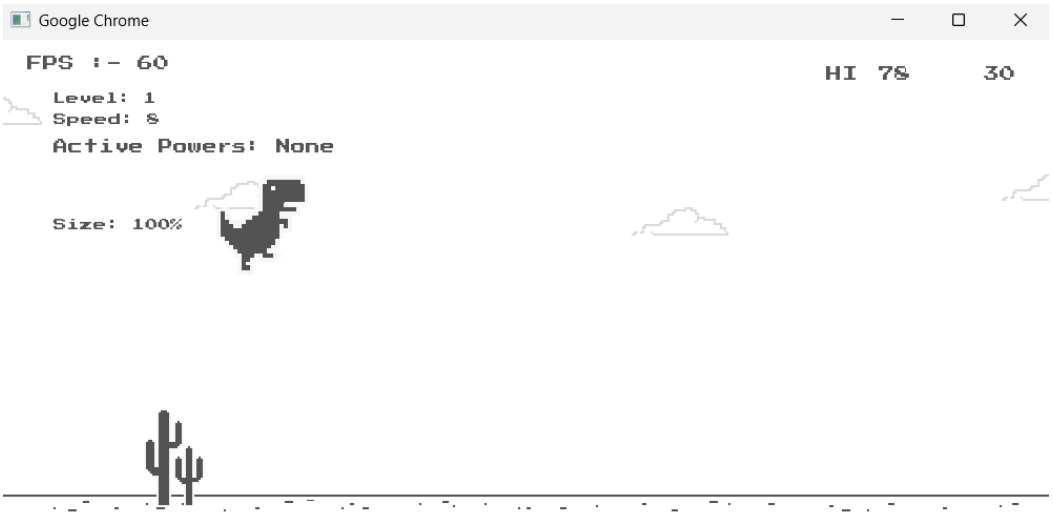
Test Case ID	Test Scenario	Input / Action	Expected Output	Status
TC01	Start Game	Launch program	Window opens successfully	Pass
TC02	Jump Action	Press **Spacebar**	Dino jumps upward	Pass
TC03	Duck Action	Press **Down Arrow**	Dino switches to duck animation	Pass
TC04	Obstacle Spawn	Wait for 1–2 sec	Cacti spawn at right side	Pass
TC05	Meteorite Spawn	Play for ~10 sec	Meteorites appear randomly	Pass
TC06	Power-Up Spawn	Play for 10–12 sec	Shield/Invincibility/Speed Boost appear	Pass
TC07	Obstacle Collision	Collide Dino with cactus	Player dies if no power-up is active	Pass
TC08	Power-Up Collision	Collect power-up	Correct effect is applied (Shield/Invincibility/Speed)	Pass
TC09	Score Increment	Keep game running	Score increases continuously	Pass
TC10	Difficulty Increment	Reach new milestones	Speed increases after every 500 points	Pass
TC11	Restart Button	Click Restart button	Game resets and restarts	Pass
TC12	Space Restart	Press **Space** on Game Over	Game restarts without issues	Pass
TC13	FPS Counter	Observe FPS text	Constant ~60 FPS	Pass
TC14	Off-Screen Cleanup	Wait for objects to leave screen	Obstacles & meteorites get removed	Pass

5.3 Output Screens and Observations

Figure 5.1 – Main Gameplay Screen

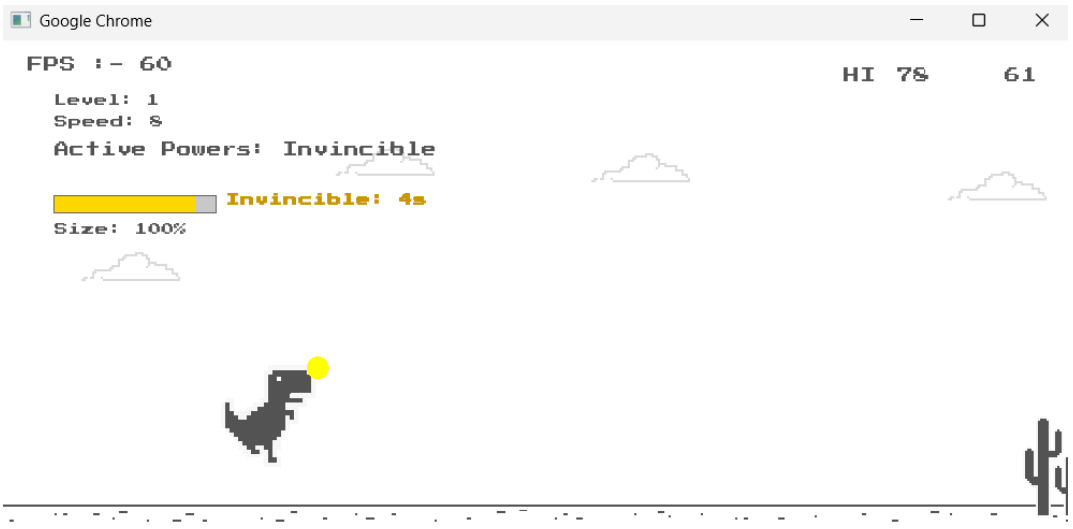
- Shows Dino running on the ground with clouds in the background.
- Score, previous score, level, and speed are displayed on the top-right corner.
- Ground scrolls smoothly.

Figure 5.2 – Jump Mechanic



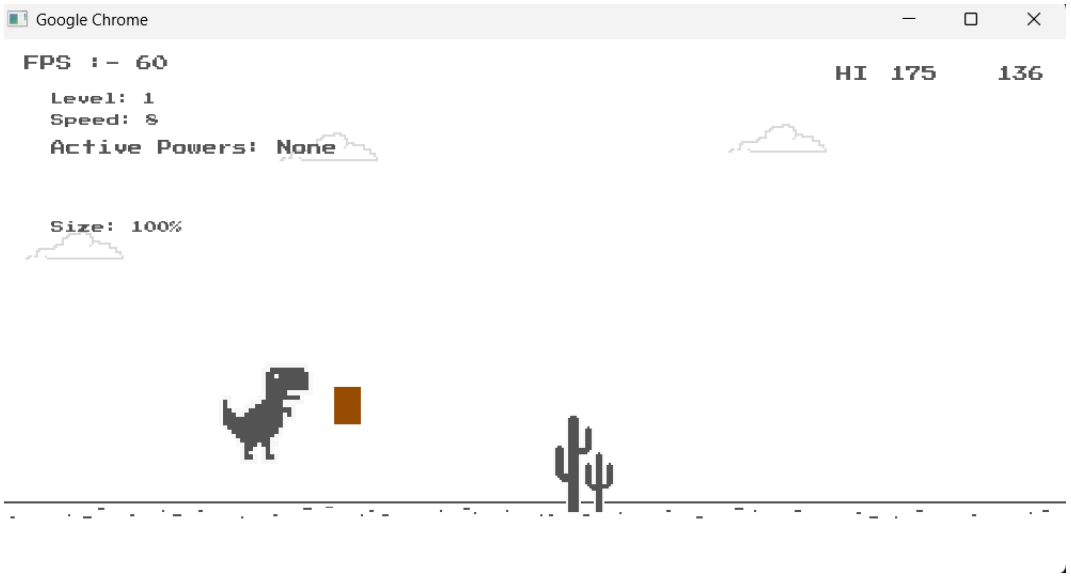
- Captures Dino in jump motion while avoiding cactus.
- Jump sound plays.
- Animation switches correctly to the jump frame.

Figure 5.3 – Power-Up Collection



- Highlight of Dino collecting Shield/Invincibility/Speed power-up.
- HUD displays active power-up and timer bars.
- Shield bar turns blue; Invincibility bar turns yellow, speed bar turns red.

Figure 5.4 – Meteorite Collision



- Shows Dino hit by meteorite.
- Dino grows or shrinks depending on the current state, 100% to 150% then 150% to 100%.
- Sound effect plays.

Figure 5.5 – Game Over Screen

- Displays “Game Over” text prominently.
- Restart button is visible at the center.
- Clicking Restart or pressing Space restarts gameplay instantly.

CHAPTER 6 – CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

The Dino Runner game successfully demonstrates how object-oriented programming, real-time rendering, event handling, and graphical animation can be implemented together to create an interactive 2D game. Using C++ and SFML, the project delivers smooth character movement, responsive controls, obstacle generation, power-up systems, scoring, and dynamic difficulty adjustment.

The game architecture is modular and easy to maintain, with separate classes handling gameplay, visuals, audio, UI, and system updates. All major objectives—such as implementing jumping and ducking mechanics, collision detection, power-up effects, meteorite interactions, HUD display, and restart functionality—have been successfully achieved.

Overall, this mini-project helped us understand practical game development concepts like the game loop, frame updates, animation timing, asset management, and real-time user interaction.

6.2 Future Enhancements

Although the core game is fully functional, several improvements can be added in future versions**:**

Gameplay Enhancements

- Addition of **new types of obstacles**, enemies, and power-ups for more variety.
- Multiple **biomes or themes** (desert, forest, night mode).
- Increasing complexity using **moving obstacles** or **flying enemies**.

User Experience Improvements

- A **main menu**, **pause menu**, and **settings page** (sound, difficulty, key bindings).
- Improved character animations and particle effects for jumps or collisions.
- Background music and more advanced sound effects.

Performance and Technical Extensions

- Implementing a **save system** for storing high scores and player progress.
- Porting the game to **Linux, macOS, or mobile platforms**.
- Refactoring into a more scalable architecture such as **ECS (Entity-Component-System)**.
- Adding **AI-based obstacle generation** for dynamic gameplay.

Accessibility Features

- Color-blind friendly power-up indicators.
- Adjustable speed modes for easier gameplay.

SOURCE CODE

```
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include <iostream>
#include <array>
#include <vector>
#include <random>

const unsigned int windowSize_x = 1000;
const unsigned int windowSize_y = 500;
const unsigned int groundLevel = 350; // Y position where dino stands
const unsigned int groundOffset = groundLevel; // Keep compatibility
int gameSpeed = 8;
bool playerDead = false;
bool playDeadSound = false;

enum PowerUpType { SHIELD, INVINCIBILITY, SPEED_BOOST };

struct PowerUp
{
   sf::CircleShape shape;
   PowerUpType type;
   sf::Vector2f position;

   PowerUp(float x, float y, PowerUpType t)
   :type(t)
   {
      shape.setRadius(15.f);
      position = sf::Vector2f(x, y);
      shape.setPosition(position);

      if(type == SHIELD)
         shape.setFillColor(sf::Color::Blue);
      else if(type == INVINCIBILITY)
         shape.setFillColor(sf::Color::Yellow);
      else
         shape.setFillColor(sf::Color::Red);
```

```
    }
};

struct Fps_s
{
    sf::Font font;
    sf::Text text;
    sf::Clock clock;
    int Frame = 0;
    int fps = 0;

    Fps_s() : text(font) {}
};
class Fps
{
    Fps_s fps;
    public:
        Fps()
        {
            if(fps.font.openFromFile("rsrc/Fonts/font.ttf"))
            {
                fps.text.setFont(fps.font);
            }
            fps.text.setCharacterSize(15);
            fps.text.setPosition(sf::Vector2f(fps.text.getCharacterSize() + 10.f, fps.text.getCharacterSize()));
            fps.text.setFillColor(sf::Color(83, 83, 83));
        }
        void update()
        {
            if(fps.clock.getElapsedTime().asSeconds() >= 1.f)
            {
                fps.fps = fps.Frame; fps.Frame = 0; fps.clock.restart();
            }
            fps.Frame++;
            fps.text.setString("FPS :- " + std::to_string(fps.fps));
        }
        void drawTo(sf::RenderWindow& window)
        {
            window.draw(fps.text);
        }
```

```
};

class SoundManager
{
   public:
      sf::SoundBuffer dieBuffer;
      sf::SoundBuffer jumpBuffer;
      sf::SoundBuffer pointBuffer;
      sf::Sound dieSound;
      sf::Sound jumpSound;
      sf::Sound pointSound;

      SoundManager()
      :dieSound(dieBuffer), jumpSound(jumpBuffer), pointSound(pointBuffer)
      {
         if(!dieBuffer.loadFromFile("rsrc/Sounds/die.wav")) std::cout << "Error loading die sound\n";
         if(!jumpBuffer.loadFromFile("rsrc/Sounds/jump.wav")) std::cout << "Error loading jump
sound\n";
         if(!pointBuffer.loadFromFile("rsrc/Sounds/point.wav")) std::cout << "Error loading point
sound\n";
      }
};

class Ground
{
   public:
   sf::Texture groundTexture;
   sf::Sprite groundSprite;
   int offset{0};
   Ground() : groundSprite(groundTexture)
   {
      if(groundTexture.loadFromFile("rsrc/Images/GroundImage.png"))
      {
         groundSprite.setTexture(groundTexture);
         groundSprite.setPosition(sf::Vector2f(0.f, windowSize_y - groundTexture.getSize().y - 50));
      }
   }

   void updateGround()
```

```
    {
        if(playerDead == false)
        {
            if(offset > (int)groundTexture.getSize().x - (int)windowSize_x)
                offset = 0;

            offset += gameSpeed;
            groundSprite.setTextureRect(sf::IntRect(sf::Vector2i(offset, 0), sf::Vector2i(windowSize_x,
windowSize_y)));
        }

        if(playerDead == true)
            groundSprite.setTextureRect(sf::IntRect(sf::Vector2i(offset, 0), sf::Vector2i(windowSize_x,
windowSize_y)));

    }
    void reset()
    {
        offset = 0;
        groundSprite.setTextureRect(sf::IntRect(sf::Vector2i(0, 0), sf::Vector2i(windowSize_x,
windowSize_y)));
    }

};

// Forward declaration
class Dino;

class Obstacle
{
    public:
        sf::Sprite obstacleSprite;
        sf::FloatRect obstacleBounds;
        Obstacle(sf::Texture& texture)
        :obstacleSprite(texture), obstacleBounds(sf::Vector2f(0.f, 0.f), sf::Vector2f(0.f, 0.f))
        {
            obstacleSprite.setPosition(sf::Vector2f(windowSize_x, groundLevel));
        }
};
```

```
class Obstacles
{
   public:
      std::vector<Obstacle> obstacles;
      std::vector<PowerUp> powerUps;
      std::vector<sf::RectangleShape> meteorites;

      sf::Time spawnTimer;
      sf::Time powerUpTimer;
      sf::Time meteoriteTimer;
      sf::Time lastObstacleTime;
      sf::Texture obstacleTexture_1;
      sf::Texture obstacleTexture_2;
      sf::Texture obstacleTexture_3;
      int randomNumber{0};
      float minSafeDistance{300.f};

      Obstacles()
      :spawnTimer(sf::Time::Zero), powerUpTimer(sf::Time::Zero), meteoriteTimer(sf::Time::Zero),
lastObstacleTime(sf::Time::Zero)
      {
         obstacles.reserve(5);
         powerUps.reserve(3);
         meteorites.reserve(5);

         if(obstacleTexture_1.loadFromFile("rsrc/Images/Cactus1.png"))
         {
            std::cout << "loaded cactus Image 1 " << std::endl;
         }

         if(obstacleTexture_2.loadFromFile("rsrc/Images/Cactus2.png"))
         {
            std::cout << "Loaded cactus Image 2" << std::endl;
         }

         if(obstacleTexture_3.loadFromFile("rsrc/Images/Cactus3.png"))
         {
            std::cout << "Loaded cactus Image 3" << std::endl;
         }
      }
```

```
    bool isSafeToSpawn()
    {
        for(auto& meteorite : meteorites)
        {
            float meteoriteX = meteorite.getPosition().x;
            float meteoriteY = meteorite.getPosition().y;

            if(meteoriteX > windowSize_x - 200.f && meteoriteX < windowSize_x + 300.f &&
meteoriteY < groundLevel - 50.f)
            {
                return false;
            }
        }
        return true;
    }

    void update(sf::Time& deltaTime, Dino& dino);

    void drawTo(sf::RenderWindow& window)
    {
        for(auto& obs : obstacles)
        {
            window.draw(obs.obstacleSprite);
        }
        for(auto& powerUp : powerUps)
        {
            window.draw(powerUp.shape);
        }
        for(auto& meteorite : meteorites)
        {
            window.draw(meteorite);
        }
    }

    void reset()
    {
        obstacles.erase(obstacles.begin(), obstacles.end());
        powerUps.erase(powerUps.begin(), powerUps.end());
        meteorites.erase(meteorites.begin(), meteorites.end());
```

```
        lastObstacleTime = sf::Time::Zero;
    }
};


class Dino
{
  public:
    sf::Sprite dino;
    sf::Vector2f dinoPos{0.f, 0.f};
    sf::Vector2f dinoMotion{0.f, 0.f};
    sf::Texture dinoTex;
    sf::FloatRect dinoBounds;
    SoundManager soundManager;
    std::array<sf::IntRect, 6> frames;
    sf::Time timeTracker;
    int animationCounter{0};
    bool isDucking{false};
    float normalHeight{95.f};
    float duckHeight{50.f};

    // Power-up states
    bool hasShield{false};
    bool isInvincible{false};
    sf::Time invincibilityTimer;
    sf::Time shieldTimer;
    float invincibilityDuration{5.f};
    float shieldDuration{8.f};

    // Size growth mechanic
    float currentScale{1.0f};
    float minScale{1.0f};
    float maxScale{1.5f};
    bool isGrowing{true};

    // Sound control
    bool deathSoundPlayed{false};

    Dino()
    :dino(dinoTex), soundManager()
```

```
{
    if(dinoTex.loadFromFile("rsrc/Images/PlayerSpriteSheet.png"))
    {
        dino.setTexture(dinoTex);
        for(int i = 0; i < frames.size(); i++)
            frames[i] = sf::IntRect(sf::Vector2i(i * 90, 0), sf::Vector2i(90, 95));
        dino.setTextureRect(frames[0]);
        dino.setOrigin(sf::Vector2f(45.f, 47.5f));
        dinoPos = dino.getPosition();
    }
    else
    {
        std::cout << "Error loading the PlayerSprite texture" << std::endl;
    }
}

void update(sf::Time& deltaTime, std::vector<Obstacle>& obstacles)
{
    dinoPos = dino.getPosition();
    dinoBounds = dino.getGlobalBounds();
    dinoBounds.size.y -= 15.f * currentScale;
    dinoBounds.size.x -= 10.f * currentScale;
    timeTracker += deltaTime;

    dino.setScale(sf::Vector2f(currentScale, currentScale));

    if(isInvincible)
    {
        invincibilityTimer += deltaTime;
        if(invincibilityTimer.asSeconds() > invincibilityDuration)
            isInvincible = false;
    }

    if(hasShield)
    {
        shieldTimer += deltaTime;
        if(shieldTimer.asSeconds() > shieldDuration)
            hasShield = false;
    }
```

```
        for(auto& obs: obstacles)
        {
            if(dinoBounds.findIntersection(obs.obstacleBounds))
            {
                if(isInvincible)
                {
                    soundManager.jumpSound.play();
                }
                else if(hasShield)
                {
                    hasShield = false;
                    soundManager.jumpSound.play();
                }
                else
                {
                    playerDead = true;
                }
            }
        }

        if(!playerDead)
        {
            if(sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Down) && dinoPos.y >= groundLevel)
            {
                isDucking = true;
                dino.setTextureRect(sf::IntRect(sf::Vector2i(0, 95), sf::Vector2i(110, duckHeight)));
                dino.setPosition(sf::Vector2f(dinoPos.x, groundLevel + (normalHeight - duckHeight)));
            }
            else
            {
                isDucking = false;
                walk();
                if(sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Space) == true && dinoPos.y >=
groundLevel)
                {
                    animationCounter = 0;
                    dinoMotion.y = -20.f;
                    dino.setTextureRect(frames[1]);
                    soundManager.jumpSound.play();
                }
```

```
            if(dinoPos.y < groundLevel)
            {
               dinoMotion.y += 1.f;
               dino.setTextureRect(frames[1]);
            }

            if(dinoPos.y > groundLevel)
            {
               dino.setPosition(sf::Vector2f(dino.getPosition().x, groundLevel));
               dinoMotion.y = 0.f;
            }

            dino.move(dinoMotion);
         }
      }
      if(playerDead == true)
      {
         dinoMotion.y = 0.f;
         dino.setTextureRect(frames[3]);

         if(!deathSoundPlayed)
         {
            soundManager.dieSound.play();
            deathSoundPlayed = true;
         }
      }

}

void walk()
{
   for(int i = 0; i < frames.size() - 3; i++)
      if(animationCounter == (i + 1) * 3)
         dino.setTextureRect(frames[i]);

   if(animationCounter >= (frames.size() - 2) * 3)
      animationCounter = 0;

   animationCounter++;
```

```
    }
    void reset()
    {
        dinoMotion.y = 0;
        dino.setPosition(sf::Vector2f(dino.getPosition().x, groundLevel));
        dino.setTextureRect(frames[0]);
        hasShield = false;
        isInvincible = false;
        invincibilityTimer = sf::Time::Zero;
        shieldTimer = sf::Time::Zero;
        currentScale = 1.0f;
        isGrowing = true;
        dino.setScale(sf::Vector2f(1.0f, 1.0f));
        deathSoundPlayed = false;
    }

};

void Obstacles::update(sf::Time& deltaTime, Dino& dino)
{
   spawnTimer += deltaTime;
   powerUpTimer += deltaTime;
   meteoriteTimer += deltaTime;
   lastObstacleTime += deltaTime;

   if(spawnTimer.asSeconds() > 1.5f + gameSpeed/8 && isSafeToSpawn())
   {
      randomNumber = (rand() % 3) + 1;
      if(randomNumber == 1)
      {
         obstacles.emplace_back(Obstacle(obstacleTexture_1));
      }
      if(randomNumber == 2)
      {
         obstacles.emplace_back(Obstacle(obstacleTexture_2));
      }
      if(randomNumber == 3)
      {
         obstacles.emplace_back(Obstacle(obstacleTexture_2));
      }
```

```
    spawnTimer = sf::Time::Zero;
    lastObstacleTime = sf::Time::Zero;
}

if(powerUpTimer.asSeconds() > 10.f + gameSpeed/3)
{
    int powerType = rand() % 3;
    float yPosition = groundLevel - 80.f - (rand() % 40);
    powerUps.emplace_back(PowerUp(windowSize_x + 50.f, yPosition, (PowerUpType)powerType));
    powerUpTimer = sf::Time::Zero;
}

if(meteoriteTimer.asSeconds() > 10.f + gameSpeed/4 && lastObstacleTime.asSeconds() > 2.f)
{
    sf::RectangleShape meteorite(sf::Vector2f(25.f, 35.f));
    meteorite.setFillColor(sf::Color(150, 75, 0));
    float xOffset = 100.f + rand() % 150;
    meteorite.setPosition(sf::Vector2f(windowSize_x + xOffset, 50.f));
    meteorites.emplace_back(meteorite);
    meteoriteTimer = sf::Time::Zero;
}

if(playerDead == false)
{
    for(int i = 0; i < obstacles.size(); i++)
    {
        obstacles[i].obstacleBounds = obstacles[i].obstacleSprite.getGlobalBounds();
        obstacles[i].obstacleBounds.size.x -= 10.f;
        obstacles[i].obstacleSprite.move(sf::Vector2f(-1*gameSpeed, 0.f));
        if(obstacles[i].obstacleSprite.getPosition().x < -150.f)
        {
            obstacles.erase(obstacles.begin() + i);
        }
    }

    for(int i = 0; i < powerUps.size(); i++)
    {
        powerUps[i].shape.move(sf::Vector2f(-1*gameSpeed, 0.f));

        sf::FloatRect powerUpBounds = powerUps[i].shape.getGlobalBounds();
```

```
    if(dino.dinoBounds.findIntersection(powerUpBounds))
    {
        if(powerUps[i].type == SHIELD)
        {
            dino.hasShield = true;
            dino.shieldTimer = sf::Time::Zero;
        }
        else if(powerUps[i].type == INVINCIBILITY)
        {
            dino.isInvincible = true;
            dino.invincibilityTimer = sf::Time::Zero;
        }
        else if(powerUps[i].type == SPEED_BOOST)
        {
            gameSpeed += 2;
        }
        powerUps.erase(powerUps.begin() + i);
        dino.soundManager.pointSound.play();
    }
    else if(powerUps[i].shape.getPosition().x < -50.f)
    {
        powerUps.erase(powerUps.begin() + i);
    }
}

for(int i = 0; i < meteorites.size(); i++)
{
    meteorites[i].move(sf::Vector2f(-1*gameSpeed, 2.5f));

    sf::FloatRect meteoriteBounds = meteorites[i].getGlobalBounds();
    if(dino.dinoBounds.findIntersection(meteoriteBounds))
    {
        if(dino.isGrowing)
        {
            dino.currentScale += 0.05f;
            if(dino.currentScale >= dino.maxScale)
            {
                dino.currentScale = dino.maxScale;
                dino.isGrowing = false;
            }
```

```
            }
            else
            {
               dino.currentScale -= 0.025f;
               if(dino.currentScale <= dino.minScale)
               {
                  dino.currentScale = dino.minScale;
                  dino.isGrowing = true;
               }
            }

            dino.soundManager.jumpSound.play();
            meteorites.erase(meteorites.begin() + i);
         }
         else if(meteorites[i].getPosition().y > windowSize_y)
         {
            meteorites.erase(meteorites.begin() + i);
         }
      }
   }
}


class Scores
{
   public:
      sf::Font scoresFont;
      sf::Text previousScoreText;
      sf::Text HIText;
      sf::Text scoresText;
      sf::Text difficultyText;
      sf::Text speedText;
      sf::Text powerUpStatusText;
      sf::Text shieldTimerText;
      sf::Text invincibilityTimerText;
      sf::Text sizeText;
      sf::RectangleShape shieldBar;
      sf::RectangleShape invincibilityBar;
      sf::RectangleShape shieldBarBg;
      sf::RectangleShape invincibilityBarBg;
```

```
    SoundManager soundManager;
    short scores{0};
    short previousScore{0};
    short scoresIndex{0};
    short scoresDiff{0};
    short scoresInital{0};
    int difficulty{1};

    Scores()
    :previousScoreText(scoresFont), HIText(scoresFont), scoresText(scoresFont),
     difficultyText(scoresFont), speedText(scoresFont), powerUpStatusText(scoresFont),
     shieldTimerText(scoresFont), invincibilityTimerText(scoresFont), sizeText(scoresFont)
    {
       if(scoresFont.openFromFile("rsrc/Fonts/Font.ttf"))
       {
          scoresText.setFont(scoresFont);
          scoresText.setCharacterSize(15);
          scoresText.setPosition(sf::Vector2f(windowSize_x/2 + windowSize_x/4 + 185.f,
scoresText.getCharacterSize() + 10.f));
          scoresText.setFillColor(sf::Color(83, 83, 83));

          previousScoreText.setFont(scoresFont);
          previousScoreText.setCharacterSize(15);
          previousScoreText.setPosition(sf::Vector2f(scoresText.getPosition().x - 100.f,
scoresText.getPosition().y));
          previousScoreText.setFillColor(sf::Color(83, 83, 83));

          HIText.setFont(scoresFont);
          HIText.setCharacterSize(15);
          HIText.setPosition(sf::Vector2f(previousScoreText.getPosition().x - 50.f,
previousScoreText.getPosition().y));
          HIText.setFillColor(sf::Color(83, 83, 83));

          difficultyText.setFont(scoresFont);
          difficultyText.setCharacterSize(12);
          difficultyText.setPosition(sf::Vector2f(50.f, 50.f));
          difficultyText.setFillColor(sf::Color(83, 83, 83));

          speedText.setFont(scoresFont);
          speedText.setCharacterSize(12);
```

```
    speedText.setPosition(sf::Vector2f(50.f, 70.f));
    speedText.setFillColor(sf::Color(83, 83, 83));

    powerUpStatusText.setFont(scoresFont);
    powerUpStatusText.setCharacterSize(14);
    powerUpStatusText.setPosition(sf::Vector2f(50.f, 95.f));
    powerUpStatusText.setFillColor(sf::Color(83, 83, 83));

    shieldTimerText.setFont(scoresFont);
    shieldTimerText.setCharacterSize(12);
    shieldTimerText.setPosition(sf::Vector2f(210.f, 118.f));
    shieldTimerText.setFillColor(sf::Color(0, 100, 255));
    shieldTimerText.setStyle(sf::Text::Bold);

    invincibilityTimerText.setFont(scoresFont);
    invincibilityTimerText.setCharacterSize(12);
    invincibilityTimerText.setPosition(sf::Vector2f(210.f, 143.f));
    invincibilityTimerText.setFillColor(sf::Color(200, 150, 0));
    invincibilityTimerText.setStyle(sf::Text::Bold);

    sizeText.setFont(scoresFont);
    sizeText.setCharacterSize(12);
    sizeText.setPosition(sf::Vector2f(50.f, 170.f));
    sizeText.setFillColor(sf::Color(83, 83, 83));
}
HIText.setString("HI");

shieldBarBg.setSize(sf::Vector2f(150.f, 15.f));
shieldBarBg.setPosition(sf::Vector2f(50.f, 120.f));
shieldBarBg.setFillColor(sf::Color(200, 200, 200));
shieldBarBg.setOutlineThickness(1.f);
shieldBarBg.setOutlineColor(sf::Color(100, 100, 100));

shieldBar.setSize(sf::Vector2f(150.f, 15.f));
shieldBar.setPosition(sf::Vector2f(50.f, 120.f));
shieldBar.setFillColor(sf::Color(0, 100, 255));

invincibilityBarBg.setSize(sf::Vector2f(150.f, 15.f));
invincibilityBarBg.setPosition(sf::Vector2f(50.f, 145.f));
invincibilityBarBg.setFillColor(sf::Color(200, 200, 200));
```

```
    invincibilityBarBg.setOutlineThickness(1.f);
    invincibilityBarBg.setOutlineColor(sf::Color(100, 100, 100));

    invincibilityBar.setSize(sf::Vector2f(150.f, 15.f));
    invincibilityBar.setPosition(sf::Vector2f(50.f, 145.f));
    invincibilityBar.setFillColor(sf::Color(255, 215, 0));
}

void update()
{
    if(playerDead == false)
    {
        scoresIndex++;
        if(scoresIndex >= 5)
        {
            scoresIndex = 0;
            scores++;
        }
        scoresDiff = scores - scoresInital;

        int newDifficulty = 1 + (scores / 500);
        if(newDifficulty > difficulty)
        {
            difficulty = newDifficulty;
            gameSpeed += 2;
            soundManager.pointSound.play();
        }

        scoresText.setString(std::to_string(scores));
        previousScoreText.setString(std::to_string(previousScore));
        difficultyText.setString("Level: " + std::to_string(difficulty));
        speedText.setString("Speed: " + std::to_string(gameSpeed));
    }

}

void updatePowerUpStatus(const Dino& dino)
{
    std::string status = "Active Powers: ";
    bool hasPowerUp = false;
```

```
if(dino.hasShield)
{
   float remaining = dino.shieldDuration - dino.shieldTimer.asSeconds();
   status += "Shield ";
   float progress = remaining / dino.shieldDuration;
   shieldBar.setSize(sf::Vector2f(150.f * progress, 15.f));
   shieldTimerText.setString("Shield Active: " + std::to_string((int)remaining) + "s");
   hasPowerUp = true;
}
else
{
   shieldBar.setSize(sf::Vector2f(0.f, 15.f));
   shieldTimerText.setString("");
}

if(dino.isInvincible)
{
   float remaining = dino.invincibilityDuration - dino.invincibilityTimer.asSeconds();
   if(hasPowerUp) status += "| ";
   status += "Invincible";
   float progress = remaining / dino.invincibilityDuration;
   invincibilityBar.setSize(sf::Vector2f(150.f * progress, 15.f));
   invincibilityTimerText.setString("Invincible: " + std::to_string((int)remaining) + "s");
   hasPowerUp = true;
}
else
{
   invincibilityBar.setSize(sf::Vector2f(0.f, 15.f));
   invincibilityTimerText.setString("");
}

if(!hasPowerUp)
{
   status += "None";
}

powerUpStatusText.setString(status);

int sizePercent = (int)(dino.currentScale * 100);
```

```
    std::string sizeStatus = "Size: " + std::to_string(sizePercent) + "%";
    if(dino.isGrowing && dino.currentScale > 1.0f)
    {
        sizeStatus += " (Growing)";
    }
    else if(!dino.isGrowing && dino.currentScale > 1.0f)
    {
        sizeStatus += " (Shrinking)";
    }
    sizeText.setString(sizeStatus);
}

void drawPowerUpBars(sf::RenderWindow& window, const Dino& dino)
{
    if(dino.hasShield)
    {
        window.draw(shieldBarBg);
        window.draw(shieldBar);
        window.draw(shieldTimerText);
    }
    if(dino.isInvincible)
    {
        window.draw(invincibilityBarBg);
        window.draw(invincibilityBar);
        window.draw(invincibilityTimerText);
    }
}

void reset()
{
    if(scores > previousScore)
        previousScore = scores;
    if(scores < previousScore)
        previousScore = previousScore;

    previousScoreText.setString(std::to_string(previousScore));
    scores = 0;
    difficulty = 1;
}
```

```
};

class RestartButton
{
   public:
      sf::Texture restartButtonTexture;
      sf::Sprite restartButtonSprite;
      sf::FloatRect restartButtonSpriteBounds;
      sf::Vector2f mousePos;
      bool checkPressed{false};

      RestartButton()
      :restartButtonSprite(restartButtonTexture), mousePos(0.f, 0.f)
      {
         if(restartButtonTexture.loadFromFile("rsrc/Images/RestartButton.png"))
         {
            restartButtonSprite.setTexture(restartButtonTexture);
            restartButtonSprite.setPosition(sf::Vector2f(windowSize_x/2 -
restartButtonTexture.getSize().x/2, windowSize_y/2));
            restartButtonSpriteBounds = restartButtonSprite.getGlobalBounds();
         }
      }
};


class Clouds
{
   public:
      std::vector<sf::Sprite> clouds;
      sf::Time currTime;
      sf::Texture cloudTexture;
      std::random_device dev;
      std::mt19937 rng{dev()};


      Clouds()
      {
         if(cloudTexture.loadFromFile("rsrc/Images/Clouds.png"))
         {
            std::cout << "Loaded CloudTexture" << std::endl;
```

```
        }
        clouds.reserve(4);
        clouds.emplace_back(sf::Sprite(cloudTexture));
        clouds.back().setPosition(sf::Vector2f(windowSize_x, windowSize_y/2 - 40.f));
    }

    void updateClouds(sf::Time& deltaTime)
    {

        currTime += deltaTime;
        if(currTime.asSeconds() > 8.f)
        {
            clouds.emplace_back(sf::Sprite(cloudTexture));

            std::uniform_int_distribution<std::mt19937::result_type> dist6( windowSize_y/2 - 200,
windowSize_y/2 - 50);
            clouds.back().setPosition(sf::Vector2f(windowSize_x, dist6(rng)));

            currTime = sf::Time::Zero;
        }


        for(int i = 0; i < clouds.size(); i++)
        {
            if(playerDead == false)
                clouds[i].move(sf::Vector2f(-1.f, 0.f));
            if(playerDead == true)
                clouds[i].move(sf::Vector2f(-0.5f, 0.f));

            if(clouds[i].getPosition().x < 0.f - (float)cloudTexture.getSize().x)
            {
                std::vector<sf::Sprite>::iterator cloudIter = clouds.begin() + i;
                clouds.erase(cloudIter);
            }
        }
    }


    void drawTo(sf::RenderWindow& window)
    {
```

```
        for(auto& cloud: clouds)
        {
            window.draw(cloud);
        }
    }

};


class GameState
{
  public:
    Fps fps;
    Dino dino;
    Ground ground;
    Obstacles obstacles;
    Scores scores;
    Clouds clouds;
    RestartButton restartButton;
    sf::Font gameOverFont;
    sf::Text gameOverText;
    sf::Vector2f mousePos{0.f, 0.f};

    GameState()
    :gameOverText(gameOverFont)
    {
        if(gameOverFont.openFromFile("rsrc/Fonts/Font.ttf"))
        {
            gameOverText.setFont(gameOverFont);
        }
        dino.dino.setPosition(sf::Vector2f(windowSize_x/2 - windowSize_x/4, groundLevel));
        gameOverText.setString("Game Over");
        gameOverText.setPosition(sf::Vector2f(restartButton.restartButtonSprite.getPosition().x -
gameOverText.getCharacterSize(),
                                    restartButton.restartButtonSprite.getPosition().y - 50));
        gameOverText.setFillColor(sf::Color(83, 83, 83));
    }
    void setMousePos(sf::Vector2i p_mousePos)
    {
        mousePos.x = p_mousePos.x;
```

```
    mousePos.y = p_mousePos.y;
}

void update(sf::Time deltaTime)
{
    restartButton.checkPressed = sf::Mouse::isButtonPressed(sf::Mouse::Button::Left);
    if(playerDead == true && (restartButton.restartButtonSpriteBounds.contains(mousePos) &&
        restartButton.checkPressed == true || sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Space)))
    {
        ground.reset();
        obstacles.reset();
        dino.reset();
        scores.reset();
        playerDead = false;
        gameSpeed = 8;
    }
    else
    {
        ground.updateGround();
        obstacles.update(deltaTime, dino);
        dino.update(deltaTime, obstacles.obstacles);
        clouds.updateClouds(deltaTime);
        scores.update();
        scores.updatePowerUpStatus(dino);
    }
    fps.update();
}

void drawTo(sf::RenderWindow& window)
{
    if(playerDead == true)
    {
        clouds.drawTo(window);
        window.draw(ground.groundSprite);
        obstacles.drawTo(window);
        window.draw(scores.scoresText);
        window.draw(scores.previousScoreText);
        window.draw(scores.HIText);
        window.draw(scores.difficultyText);
        window.draw(scores.speedText);
```

```
            window.draw(scores.powerUpStatusText);
            window.draw(scores.sizeText);
            scores.drawPowerUpBars(window, dino);
            window.draw(dino.dino);
            window.draw(gameOverText);
            window.draw(restartButton.restartButtonSprite);
            fps.drawTo(window);
        }
        else
        {
            clouds.drawTo(window);
            window.draw(ground.groundSprite);
            obstacles.drawTo(window);
            window.draw(scores.scoresText);
            window.draw(scores.previousScoreText);
            window.draw(scores.HIText);
            window.draw(scores.difficultyText);
            window.draw(scores.speedText);
            window.draw(scores.powerUpStatusText);
            window.draw(scores.sizeText);
            scores.drawPowerUpBars(window, dino);
            window.draw(dino.dino);
            fps.drawTo(window);
        }
    }

};

int main()
{
   sf::RenderWindow window(sf::VideoMode({windowSize_x, windowSize_y}), "Google Chrome");
   window.setFramerateLimit(60);

   GameState gameState;

   sf::Clock deltaTimeClock;

   while(window.isOpen())
   {
      while(const std::optional<sf::Event> event = window.pollEvent())
```

```
        {
            if(event->is<sf::Event::Closed>())
                window.close();
        }
        gameState.setMousePos(sf::Mouse::getPosition(window));
        sf::Time deltaTime = deltaTimeClock.restart();

        gameState.update(deltaTime);

        window.clear(sf::Color::White);
        gameState.drawTo(window);
        window.display();
    }
}
```