End Semester Project
CS-114: Fundamentals of Programming

National University of Sciences and Technology (NUST)
College of Electrical and Mechanical Engineering (CEME)

STUDENT DECLARATION

I declare that:

- I understand what is meant by plagiarism.
- The implication of plagiarism has been explained to me by my institution.
- This Project is all my own work and I have acknowledged any use of the published and unpublished works of other people.

Sr. No.	Name	CMS ID No.	No. of Pages
1	M. Soban Hussain	505247	
2	Hamza Imran	508981	
3	Hamza Mudassir	504982	
4	Hassan Ali	512606	
Submission Due Date: 13th May, 2025		**Degree/ Syn:** CE-46-A	

CONTENTS

ABSTRACT

This code implements a 2D retro arcade fighting game using SFML, featuring two animated characters with various states like Idle, Move, Jump, Punch, and Kick. Each character has individual movement, attack, and gravity logic, with animations driven by sprite sheets. The game reads state data from a file and includes basic AI for the enemy character. Collision detection handles attack interactions and health reduction. The main loop updates game logic, animations, gravity simulation, and sprite rendering frame-by-frame.

INTRODUCTION

This project implements a 2D fighting game using the SFML (Simple and Fast Multimedia Library), designed to provide an immersive and interactive combat experience. To optimize the performance and reduce the load on the CPU, we used sprite sheets instead of individual PNG images for character animations. A sprite sheet allows for faster rendering as all the character frames are stored within a single image file, significantly lowering the resource consumption compared to loading multiple files. The game features two characters, each with a variety of animations like idle, movement, jumping, and attacking. The enemy character's sprite was modified in color to provide a visual distinction from the player.

Instead of creating separate files and headers for different components, we decided to consolidate all the game logic into a single file. This decision aimed to keep the game's files organized and manageable, ensuring that all code related to game mechanics, animation, physics, and AI behavior is housed in one place for easier debugging and testing. The game uses a state-machine system to manage the characters' actions, where each state corresponds to a specific animation and behavior. User input controls the player character's movement and attacks, while an AI system governs the enemy character's responses, providing a dynamic and unpredictable experience.

Core gameplay mechanics, such as gravity simulation and collision detection, were implemented to add realism and interaction. The gravity system controls vertical movements like jumping, ensuring that characters fall back to the ground when they are not on it. Collision detection ensures that when two characters collide, the corresponding health of the character is reduced, and the "Hurt" state interrupts their current animation. This combination of efficient asset management, real-time state updates, AI-driven enemy actions, and seamless collision handling results in a smooth and engaging 2D fighting game.

CURRENT SYSTEM

The current system has several limitations stemming from initial design choices and time constraints. One of the primary challenges was the attempt to create a custom library to manage game logic. However, due to the complexity of the game mechanics, this approach proved difficult, and we ultimately decided to integrate all the logic into a single file. This decision simplified the project but also led to certain limitations in organization and scalability. The integration of SFML into the project also proved time-consuming as connecting all the necessary static libraries was more complex than initially anticipated. Despite these difficulties, we succeeded in getting SFML working, which enabled the game to function with animations, input handling, and graphics rendering.

Despite the functionality achieved with the current system, several important features had to be sacrificed due to time constraints. For example, we were unable to implement sound effects and background music for the game, which would have enriched the player's experience. Similarly, multiplayer support, obstacle interaction, and additional character models were all planned but ultimately not included in the final version due to time limitations. Although we began with four characters, we had to reduce this to one, simplifying the gameplay but also limiting the variety and complexity of the game. These limitations are what we aim to overcome in the proposed system, which will allow for a more polished and feature-rich experience.

PROPOSED NEW SYSTEM

The proposed new system aims to address the issues faced in the current system by introducing a more modular and efficient design. Instead of relying on a single file for all the game logic, we would separate the various components of the game into different files, such as AI, physics, graphics, and game mechanics. This approach would make the code more organized and maintainable while also improving scalability for future features. Additionally, the modular design would facilitate easier debugging, as each component could be tested and updated independently. This restructuring would not only optimize performance but also make the game more flexible and adaptable to new ideas and functionalities.

In terms of features, the new system would introduce many enhancements that were not included in the current version. Sound effects and background music would be integrated to enhance the immersion and excitement of the game. Multiplayer functionality and obstacle interaction would be added to provide more dynamic gameplay. Furthermore, the new system would support multiple characters, allowing players to choose from a variety of fighters and adding depth to the gameplay. By improving the AI for enemy characters and incorporating more advanced game mechanics, the proposed system would offer a much more engaging and enjoyable experience for players.

Benefits: The proposed system would provide several key benefits over the current system. First, it would allow for greater scalability, making it easier to add new features and expand the game in the future. The modular structure would improve the maintainability and readability of the code, making it easier for team members to collaborate and for new developers to understand the system. Enhanced features like multiplayer support and diverse character options would make the game more enjoyable for players, adding replayability and variety. The integration of sound effects and background music would further immerse players in the game, creating a more engaging experience. Overall, the new system would be more polished, flexible, and feature-rich, providing both developers and players with a much better gaming experience.

- Removing the dependency on a separate custom library and instead managing everything efficiently in a single file.
- Successfully setting up and linking SFML with all static libraries, resolving initial configuration issues.
- Making the codebase cleaner and easier to debug by avoiding unnecessary complexity.

Benefits:

- Faster setup time and fewer build errors.
- Simpler structure makes it easy for beginners to understand.
- Stable and functional execution without external file complications.

PROBLEM DESCRIPTION

The problems that we faced were:

- Creating a separate custom library proved to be very difficult. Although we managed to make one, we discarded it since our course focuses on C++ and not OOP, so we continued with a single main file.

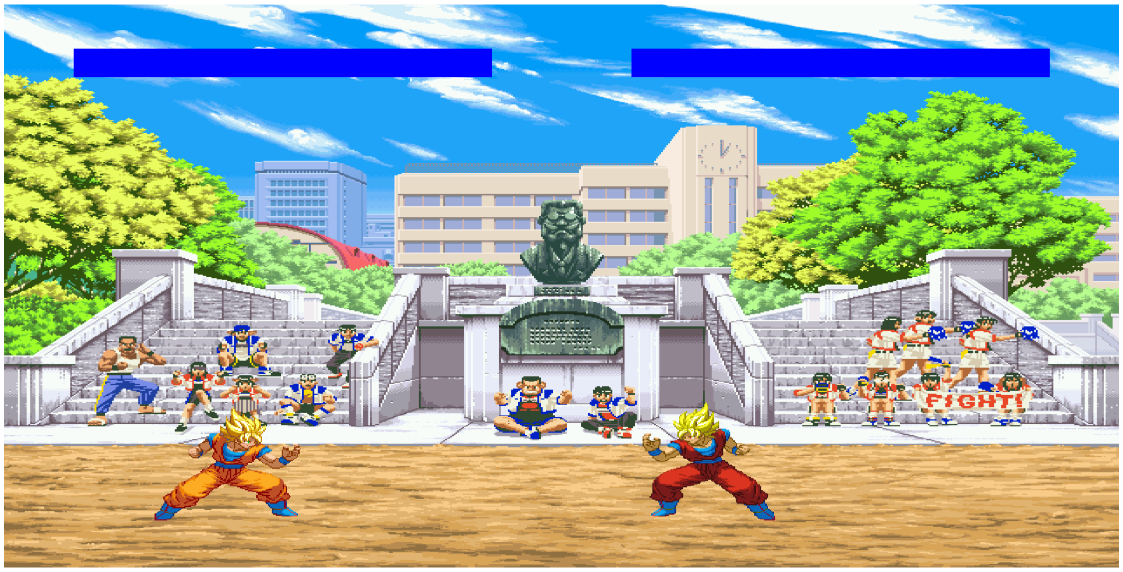

Fig. 1. Retro Arcade Game Interface

- Implementing SFML and linking all static libraries consumed a lot of our time. It was challenging, but we eventually succeeded.
- Due to lack of time, we couldn't add features like:
  - Music and fighting sounds
  - Obstacle interaction
  - Multiplayer mode
  - Multiple characters (we made 4 initially but skipped them due to time limits)

HARDWARE / SOFTWARE REQUIREMENTS

- IDE: Visual Studio

I. SYSTEM SPECIFICATION / FUNCTION MODULES

A. 1. Character Class

- **Represents the player and enemy characters.**
- **Attributes:**
  - Health, speed, gravity, position, texture, sprite, and animation states.
- **Functions:**
  - `update()`: Updates the character's actions based on input (move, attack, etc.).
  - `animate()`: Manages the animation of the character.
  - `move()`: Handles character movement (left, right, jump).
  - `flip()`: Flips the character's sprite for movement direction.
  - `loadStates()`: Loads the different animation states from a file.

B. 2. Gravity Simulation

- **Function: `SimulateGravity()`**
  - Simulates the gravity effect for characters to make them fall when in the air.
  - Ensures the character lands when reaching the ground.

C. 3. Collision Detection

- **Function: `collides()`**
  - Detects if two characters' sprites intersect (for collision detection).

D. 4. Enemy AI

- **Function: `updateEnemy()`**
  - Controls the enemy character's behavior, including movement and attack based on distance from the player and random decisions.
  - The enemy performs random actions like punch, kick, block, or movement towards the player.

E. 5. State Management

- Character states like `Idle`, `Punch`, `Kick`, `Jump`, etc., are managed by the `State_Cords` structure.
- The state transitions based on user input or AI decisions (such as attacking or moving).
- `setState()` and `loadStates()` manage the character's state and load animations.

F. 6. Main Game Loop

- Handles window creation, event processing, and game logic:
  - Updates player and enemy characters.
  - Checks for input and updates character states.
  - Detects collisions and updates health.
  - Draws characters to the screen.

G. 7. Input Handling

- Captures user input from the keyboard (e.g., `W`, `S`, `A`, `D`, `Space`, `J`, `K`, etc.) to control character actions.
- Controls both movement and attacks.

H. 8. Graphics

- Uses SFML (Simple and Fast Multimedia Library) for rendering sprites and handling graphics.
- Manages sprite animations for each character based on their current state.
- Handles the rendering of characters on the screen based on their position and state.

This modular approach allows for easy integration of new features such as additional characters, levels, or even multiplayer modes by updating or adding to these modules.

Summary

This project showcases the design and implementation of a 2D fighting game using object-oriented programming principles and graphical rendering via SFML. Core modules such as character control, enemy AI, collision detection, and animation systems work together to provide an interactive and responsive gameplay experience. The modular structure ensures scalability for future upgrades or feature additions.

Acknowledgements

We express our profound gratitude to our respected course instructor for their continuous guidance, encouragement, and insightful feedback throughout the duration of this project. Their expertise greatly contributed to our understanding and successful completion of this work. We are also thankful to the department for providing the necessary resources and facilities. Lastly, we acknowledge the contributions of open source communities whose tools and libraries were integral to the development of this project.

References

We did not use external references or resources in the development of this project. All work, code, and design were created independently by the project team.