SNAKE GAME DEVELOPMENT USING C

UCS3212 – Fundamentals and Practice of Software Development

A PROJECT REPORT

Submitted By

1. Dravid Ranjan A M – 3122245001046
2. Gopinath S – 3122245001048
3. Guhanesh M L – 3122245001049

Department of Computer Science and Engineering
Sri Sivasubramaniya Nadar College of Engineering
(An Autonomous Institution, Affiliated to Anna University)
Kalavakkam – 603110

April 2025

Sri Sivasubramaniya Nadar College of Engineering
(An Autonomous Institution, Affiliated to Anna University)

BONAFIDE CERTIFICATE

Certified that this project report titled **"SNAKE GAME APPLICATION"** is the bonafide work of Dravid Ranjan A M (3122245001046), Gopinath S (3122245001048), Guhanesh M L (3122245001049) who carried out the project work in the UCS2265 – Fundamentals and Practice of Software Development during the academic year 2024-25.

Internal Examiner External Examiner

Date:

Abstract

The Snake game is a console-based application developed in the C programming language, drawing its inspiration from the classic Snake game that many are familiar with. Being console-based means it operates entirely within a text interface like the terminal or command prompt, without the use of any graphical interface libraries. This minimal environment emphasizes the logic and control mechanisms behind the game rather than visual presentation. The primary goal of this project is not just to recreate the classic gameplay, but also to serve as an educational tool that demonstrates key programming principles in a structured and interactive way.

The development of this game focuses on applying fundamental concepts of C programming. One major aspect is the use of structures, which allow the programmer to group related data types—such as the coordinates of the snake's body, its length, and direction—into organized units. File handling is another core component, which may be used to store high scores or game progress, allowing the game to read from and write to files efficiently. Another crucial part of the implementation is handling real-time user input, which ensures that the snake responds instantly to keystrokes during gameplay. This is particularly challenging in C, especially in a console environment, because standard input methods like 'scanf' or 'getchar' usually block program execution.

The game itself revolves around a snake that moves continuously within a predefined boundary or playing field. As the player guides the snake, it consumes food items that appear randomly within the area. Each time the snake eats a piece of food, it grows longer, making it progressively harder to navigate without running into obstacles. A critical aspect of the gameplay is avoiding collisions—either with the boundaries of the field or with the snake's own growing body. A collision results in a game over, which adds pressure and excitement as the game progresses.

To increase the difficulty level, the speed of the snake is designed to rise as the player advances, typically as more food is consumed or more points are earned. This mechanic ensures that the game becomes more challenging over time, maintaining the player's engagement. Moreover, a penalty system is introduced for attempting to reverse the snake's direction directly—such as moving from left to right and then immediately trying to go left again. This not only prevents the snake from running into itself by logic but also forces the player to think more strategically, enhancing both the entertainment and skill-based aspects of the game.

Several features make the game feel responsive and enjoyable. Smooth movement control is achieved by detecting keyboard input in real-time and updating the snake's position on the screen without noticeable lag. The score is tracked continuously and displayed as the player progresses, offering immediate feedback and motivation. In addition, the game includes a robust game-over detection mechanism that instantly ends the session upon any fatal collision, making the game more polished and complete.

Finally, the project exemplifies good programming practices such as structured programming and modular design. Each part of the game—like rendering the screen, checking for collisions, handling input, and updating the snake's position—is likely divided into separate functions or modules. This approach makes the code cleaner, easier to debug, and more maintainable. User interaction is also handled efficiently, allowing for a smooth and responsive gameplay experience. Overall, this game not only serves as a fun project but also as an excellent demonstration of fundamental programming techniques in C.

Contents

1 Introduction

The Snake Eat Control game is a revitalized and enhanced version of the original Snake game, which first gained popularity in the late 1970s on arcade machines and later became a cultural icon when it was featured on Nokia mobile phones in the early 2000s. The classic gameplay—where a snake moves across the screen eating food and growing longer—has been reimagined in this project using the C programming language. Beyond simply mimicking the old mechanics, this project seeks to improve upon them using modern programming techniques such as user-defined data types (like struct in C for organizing data), real-time input handling (for smooth and responsive controls), and modular programming (for clean, maintainable code).[3] The primary goal is both to entertain and to serve as an educational resource for programmers learning foundational concepts in software development.

Project Components:

Game Logic
The game logic forms the core of the Snake Eat Control project, governing how the game behaves and responds to events. This includes managing the snake's movement, detecting collisions with walls or its own body, determining how the snake grows when food is consumed, and implementing any penalties or changes in difficulty over time (such as increasing speed). The game logic must be carefully designed to ensure that each update cycle correctly reflects the current state of the game and that transitions between game states (like playing, paused, or game over) are smooth and bug-free. This component demonstrates logical structuring, conditional branching, and looping constructs, all of which are essential in C programming.

Object Generation
The object generation component is responsible for dynamically creating game elements such as food, bomb and obstacles . In the traditional game, food appears randomly within the playable area, and the snake must navigate to reach it. In this project, object generation ensures that food is placed at random but valid locations—meaning it must not spawn on top of the snake or outside the boundary. This requires the use of random number generation, coordinate validation, and potentially collision checking at the moment of object creation. This component showcases the use of functions, randomization techniques, and spatial data handling in C. In addition to this, we have created the generation of four different types of powerups which grant powers like ghost, speed, double point and invincibility.

User Input Handling
Real-time user input handling is one of the most crucial aspects of the game, allowing the player to control the snake's direction using keyboard keys (arrow keys and WASD). In console-based C programs, capturing real-time input without pausing the game loop can be challenging, as traditional input functions like scanf() or getchar() are blocking. This project overcomes that by using built-in functions in Raylib such as IsKeyPressed() to detect keystrokes as they happen. This ensures responsive and fluid gameplay, and introduces learners to event-driven programming and real-time input processing in C.

Score Tracking System
The score tracking system keeps count of how many food items the snake has consumed, which typically translates into the player's score. Each time the snake eats, the score increases and is displayed on the screen. This component may also involve storing high scores using file I/O operations, introducing the player to the basics of file handling in C. Score tracking teaches how to use variables, formatted output, and persistent storage efficiently. It also plays a key role in providing feedback and motivation to the player, which is essential in game design.

Game Over Conditions
The game over conditions define when the game should end. This occurs if the snake collides with the wall (boundary of the play area) or with its own body or even obstacles. Detecting these events involves tracking the snake's head position and comparing it to both the boundaries and the coordinates of its own body segments. When such a condition is met, the game transitions to a game-over state, halting gameplay and typically displaying a final score. This component illustrates state management and conditional logic, which are important for creating stable and user-friendly programs.

Graphical Representation
The Graphical Representation component enhances the Snake Eat Control game by using the Raylib graphics library to create 2D animations with spritesheets. Instead of basic text output, the snake and other game elements are visually animated using sprite frames, offering smoother and more engaging visuals. Spritesheets, referenced from [6], allow the game to display directional movement and animation effects by cycling through image frames. Raylib handles loading textures, slicing frames, and rendering them in sync with the game logic. This component introduces learners to 2D rendering, animation, and graphical asset integration, bridging the gap between basic console programs and visually rich game development.

Overall, the Snake Eat Control game project is not just about creating a fun retro game — it's a comprehensive learning tool. It teaches core concepts of memory management, especially when dynamically updating the snake's length using pointers or arrays. The use of modular programming helps students understand how to break down a large problem into manageable functions or modules, making the code more readable and maintainable. Most importantly, it provides hands-on experience with game development fundamentals such as game loops, rendering, state handling, and interaction, all of which are transferable to more complex programming tasks in the future.

2 Limitations of Existing Solutions

Obstacles:
The game lacks dynamic challenges beyond simple self-collision and wall detection.No enemy snakes or intelligent behaviors to increase difficulty.

Fixed Game Arena Size:
The game grid is predefined and does not adjust dynamically to different screen sizes. No support for levels or expanding the play area as the game progresses.

No Save or Resume Feature:

Players cannot save their progress for later sessions. Every game session starts from scratch.

No Multiplayer Mode:

The game is single-player only, with no options for playing with friends. Lack of network or local multiplayer support.

3 Problem Statement

Develop a Snake Game using C programming that simulates the popular snake eat control game. The game includes several features to enhance the gameplay experience. First, users can choose the mode of gameplay among classic, multiplayer, survival, and maze. The snake's movement is controlled using arrow keys. Food is randomly generated for the snake to consume and grow in length. Additionally, the game includes randomly generated obstacles such as bombs and barriers. Powerups are also generated at regular intervals, providing special abilities to the snake. A scoring system keeps track of the current score and the highest score achieved. Collision detection ensures that the snake does not collide with the walls or its own body, thereby maintaining the challenge and integrity of the game.

4 Exploration of Problem Statement

Game rules:

The snake moves at a constant speed in one direction, and we can change its direction using the movement keys. However, the snake can only turn 90° at a time.Turning in opposite direction(180°) will result in self consumption and the score and length of snake is reduced.The objective is to eat as many generated food as possible, which will make the snake to grow in length.The game ends if the snake collides with the boundary,obstacle or its own body.The snake body and score will be decreased upon bomb collision. Collision with powerups grant special abilities for limited time.

Project Components:

Game Logic:

The game logic forms the backbone of the Snake Eat Control game. It determines how the snake moves, grows, interacts with food, and checks for collisions. The movement is typically grid-based, where the snake moves one step at a time in a specific direction. The direction can be updated in real-time based on user input, but the underlying logic ensures continuous and smooth movement. This system must consider not only position updates but also how to handle the growing tail and avoid immediate collisions.

A crucial part of the game logic is managing the snake's body structure, often implemented using a list or array of coordinates. Each segment of the snake must follow the head, creating a trailing effect that mimics the appearance of a real snake. As the snake consumes food, its body grows, which is reflected by extending the array or list. The logic for body movement must shift every coordinate accordingly to simulate movement.

Another vital component of game logic is collision detection. This involves checking whether the snake has run into a wall or its own body. These checks are performed on every iteration of the game loop, ensuring timely responses such as ending the game when a collision is detected. The challenge lies in optimizing these checks to run quickly, even as the snake grows longer.

Finally, the game logic includes the main game loop, which continuously updates the game state. It controls the timing of movement, the generation of food, score updates, and redraws the console screen. This loop must be efficient and responsive, which is achieved through careful time management using system functions and conditional logic. Together, these logical components make the game dynamic and engaging.

Object Generation

The object generation component is responsible for creating dynamic in-game elements, particularly the food items that the snake consumes to grow. These objects are typically generated at random positions within the boundaries of the game field, ensuring they do not overlap with the snake's body or appear outside the playable area. Random number generation functions in C (such as rand()) are used for this purpose, combined with logic to avoid invalid positions.

This system enhances gameplay by introducing an element of unpredictability. Each time the snake eats a piece of food, a new one must be generated instantly, keeping the game continuous and challenging. The object generation function must ensure that the new food is not placed inside the snake's body, which requires scanning the current body positions before confirming the new location.

Advanced implementations of object generation may include different types of food, such as special food items that give extra points or alter the speed of the game. This adds variety and excitement, making each game session unique. Implementing such enhancements requires extending the object structure to include additional attributes like type, value, or lifetime.

In addition, object generation also plays a role in initializing the snake's position at the start of the game. Placing the snake at a central location and spawning the first food item are part of the setup routine. These initial object placements lay the foundation for the rest of the gameplay, making this component essential for both the beginning and the ongoing dynamics of the game.

User Input Handling:

User input handling is a key feature that allows players to control the direction of the snake using keyboard keys. In a console-based game developed in C, capturing real-time input is non-trivial because standard input functions like scanf or getchar are blocking. To solve this, non-blocking input mechanisms must be implemented, often using platform-specific functions such as kbhit() and getch() on Windows systems, or termios on Unix-based systems.

This component continuously listens for user keystrokes without stopping the game loop. When a valid key (like an arrow key or WASD) is pressed, the direction of the snake is updated immediately. Input handling must include safeguards to prevent illegal movements, such as reversing direction 180 degrees, which would cause instant self-collision.

In addition to movement controls, the input handling system may support pause, restart, or exit commands, allowing more interactivity and flexibility for the player. These commands are usually bound to specific keys (like 'P' for pause or 'Q' for quit) and are

checked in each iteration of the game loop.

Furthermore, this component ensures responsive gameplay. The challenge lies in balancing speed control and input responsiveness so that the snake's motion feels smooth while accurately reflecting the player's commands. Efficient input handling directly affects the quality and playability of the game.

Score Tracking System:

The score tracking system monitors the player's progress by keeping count of how many food items the snake has consumed. Each time the snake eats food, the score increases, and the updated score is displayed on the screen. This gives players real-time feedback and a sense of achievement, which is crucial for engagement and replay value.

The score is typically stored as an integer variable that is incremented with every successful consumption of food. For a more advanced system, different food types could yield varying point values, and bonus challenges could be introduced to multiply scores based on speed or combos. Displaying the score involves updating the screen frequently, usually alongside the game field.

In more advanced implementations, the game may use file handling to store high scores. This allows players to compete against past performances or friends. When a game ends, the current score is compared with stored high scores and saved if it surpasses the previous best. This feature involves reading from and writing to files using C's file I/O functions like fopen(), fprintf(), and fscanf().

Finally, the score system may be tied into difficulty adjustments, such as increasing the snake's speed after certain thresholds. This makes the game more dynamic and adds a layer of strategy and pressure as players strive to achieve higher scores under increasing difficulty.

Game Over Conditions:

The game over conditions define when and how the game ends. These conditions are vital for controlling the game's flow and signaling to the player that they've either failed or reached a terminal state. In Snake, the game typically ends when the snake collides with the walls or its own body. These checks are performed continuously during gameplay.

Collision with a wall is detected by comparing the snake's head position with the game boundary. If the head moves beyond the allowed coordinate range, the game registers a fatal collision. Similarly, self-collision occurs when the head's position matches any of the body segments, which requires iterating through the body coordinates and checking for overlaps.

Once a game-over condition is met, the program transitions to a different state — the game loop halts, and a "Game Over" message is displayed, along with the final score. The system may also prompt the player to restart or exit. Implementing this transition smoothly requires managing program states and controlling what the screen displays during and after the game.

In extended versions of the game, game-over events could trigger additional logic such as saving the score, updating a high-score list, or logging game stats. By implementing these conditions carefully, the game maintains fairness, provides closure to each session, and creates opportunities for competitive or repeat play.

Graphical Representation: The Graphical Representation component transforms the

traditional console-based Snake game into a visually engaging 2D animated experience by integrating the Raylib graphics library. Raylib is a simple and powerful C library designed for game development, offering built-in support for rendering shapes, images, text, and handling input and audio. In this project, Raylib is used to animate the snake and other game elements in a 2D space, enhancing the overall gameplay and user experience far beyond plain text output.

A key technique employed here is the use of spritesheets, which are image files containing multiple animation frames arranged in a grid. These are typically used to animate game characters or objects by quickly switching between frames to simulate movement. In the case of the Snake Eat Control game, the snake is visually represented by animated sprites—each segment of the snake (head, body, tail) may use a different sprite, and the movement of the snake is animated frame by frame, making it feel more dynamic and lively. The spritesheets referenced from [6] provide a consistent and appealing art style, adding polish and personality to the game.

Implementing this graphical layer involves loading image textures, slicing them into individual frames, and drawing the correct frame based on the snake's direction and state. The animation logic must sync with the game's timing and input so that the visual representation matches the game logic. This component introduces essential concepts in 2D rendering, texture management, frame animation, and graphical asset integration using C and Raylib.

By incorporating Raylib, the project moves beyond a purely educational exercise and starts to bridge into real-world game development, where visual feedback and aesthetics play a crucial role. It also teaches how to blend logic-driven code with visual presentation, helping learners understand the basics of game graphics pipelines, rendering loops, and animation timing.

5 System Architecture

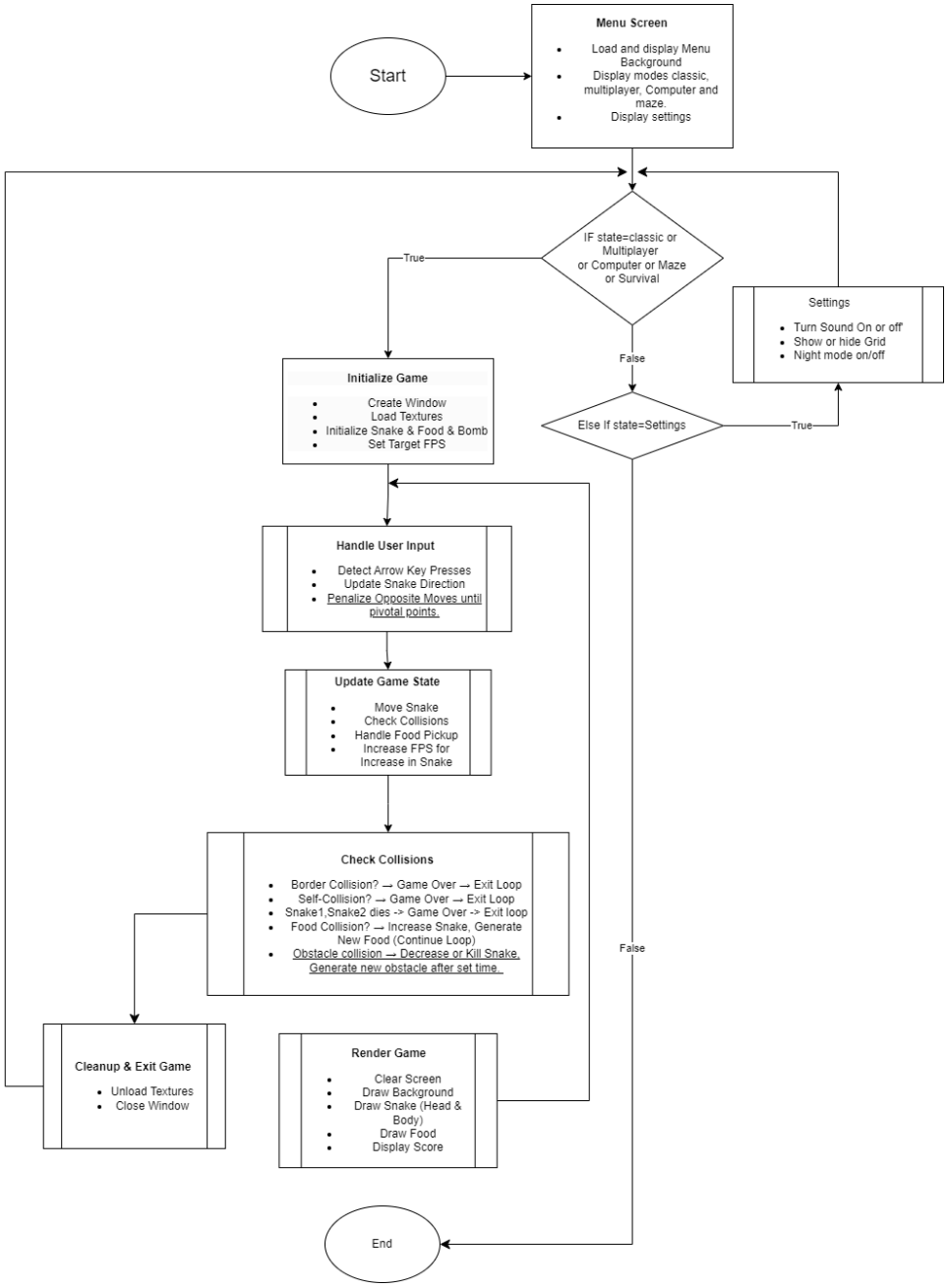

Figure 1: System Architecture

The architecture diagram presents a comprehensive overview of the Snake game's system flow, beginning from initialization to final cleanup. The system follows a structured state-based approach that clearly separates different components of the game while maintaining logical connections between them.

Menu Screen

The game begins with the **Menu Screen** which serves as the entry point, handling initial graphics rendering and mode selection. This screen loads and displays the menu background texture while presenting various gameplay options including classic mode, multiplayer, computer opponent, and maze variations. The menu also provides access to game settings, establishing the foundation for user customization before gameplay begins.

Initialization Phase

Initialization forms the critical setup phase where the system creates the game window, loads all necessary textures (snake segments, food items, bombs), and configures the game environment. This includes setting up the snake's initial position, spawning the first food items, and establishing the target frame rate for smooth gameplay. The initialization process carefully allocates resources and prepares all game objects for the main loop.

Core Gameplay Loop

The core **gameplay loop** activates when a valid game state (classic, multiplayer, computer, maze, or survival) is selected. This loop continuously processes three main operations: handling user input, updating game state, and rendering the scene. Input handling captures arrow key presses and translates them into snake movement directions, while implementing the game's unique reversal penalty system that shrinks the snake when making opposite-direction moves.

Game State Updates

Game state updates form the computational heart of the system, moving the snake according to player input and checking for various collision scenarios. The collision detection system evaluates multiple conditions including border impacts (which end the game), self-collisions (also fatal), inter-snake collisions in multiplayer mode, and beneficial food collisions that grow the snake. The system also manages obstacle interactions that may decrease snake length or trigger game over conditions based on game mode rules.

Rendering System

Rendering operations maintain visual fidelity by clearing and redrawing the entire scene each frame. This includes displaying the background, all snake segments (with distinct head graphics), food items at their current positions, and the constantly updating score display.

Cleanup Phase

The architecture concludes with a dedicated **cleanup phase** that properly releases all allocated resources when the game ends. This includes unloading textures from GPU memory and closing the game window, preventing memory leaks and ensuring system resources are returned to the operating system.

6 Functions Flowchart

The flowchart illustrates the main game loop:

1. Initialize game state
2. Process user input
3. Update snake position
4. Check for collisions
5. Handle food consumption
6. Render graphics
7. Repeat until game over

6.1 Initialization Process

The following flowchart represents the `initGameObjects()` function, detailing the initialization process for the game environment and entities. The process begins with setting up the game window and proceeds to initialize Snake1, assigning its direction, length, position, and other related attributes. If the game mode is set to Multiplayer or Computer, Snake2 is also initialized similarly. Next, the function resets the state of core game entities including foods, bombs, and powerups, ensuring they are marked inactive. Timers such as `gameTimer`, `difficultyTimer`, `bombTimer`, `powerupTimer`, and `obstacleTimer` are reset to `0.0f`. Then, initial gameplay variables like `foodCount`, `bombCount`, and `obstacleCount` are set based on the selected game mode—Survival mode activates a bomb, while other modes do not. Following this, food items are generated, and if bombs are required, they are also initialized. If the game mode is Maze, specific obstacles are generated to challenge the player. Finally, the level is set to 1, concluding the setup and returning control to the main game loop. This structured process ensures a consistent and dynamic starting state for each game session.

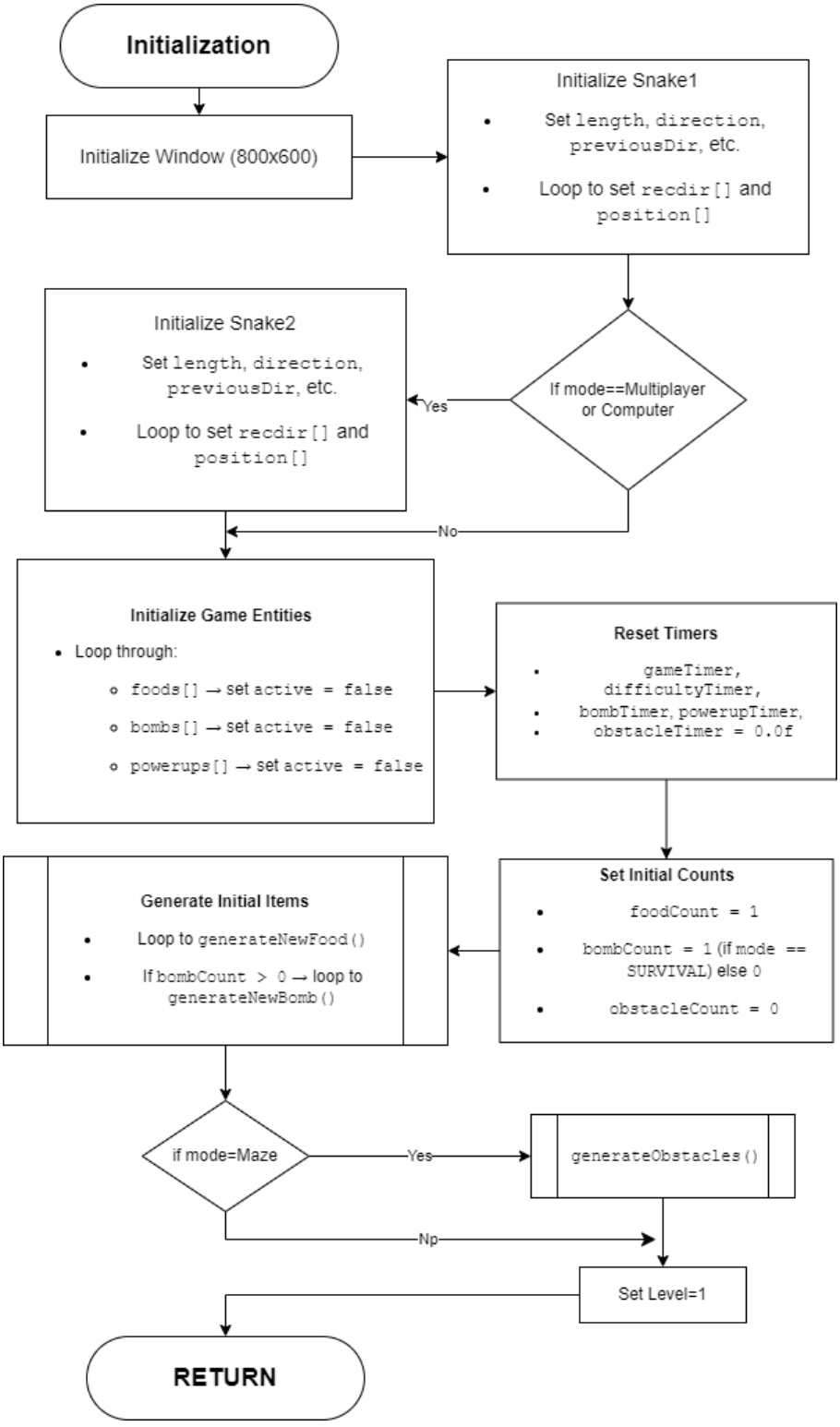

Figure 2: Flowchart depicting the initialization process

6.2 Collision Handling

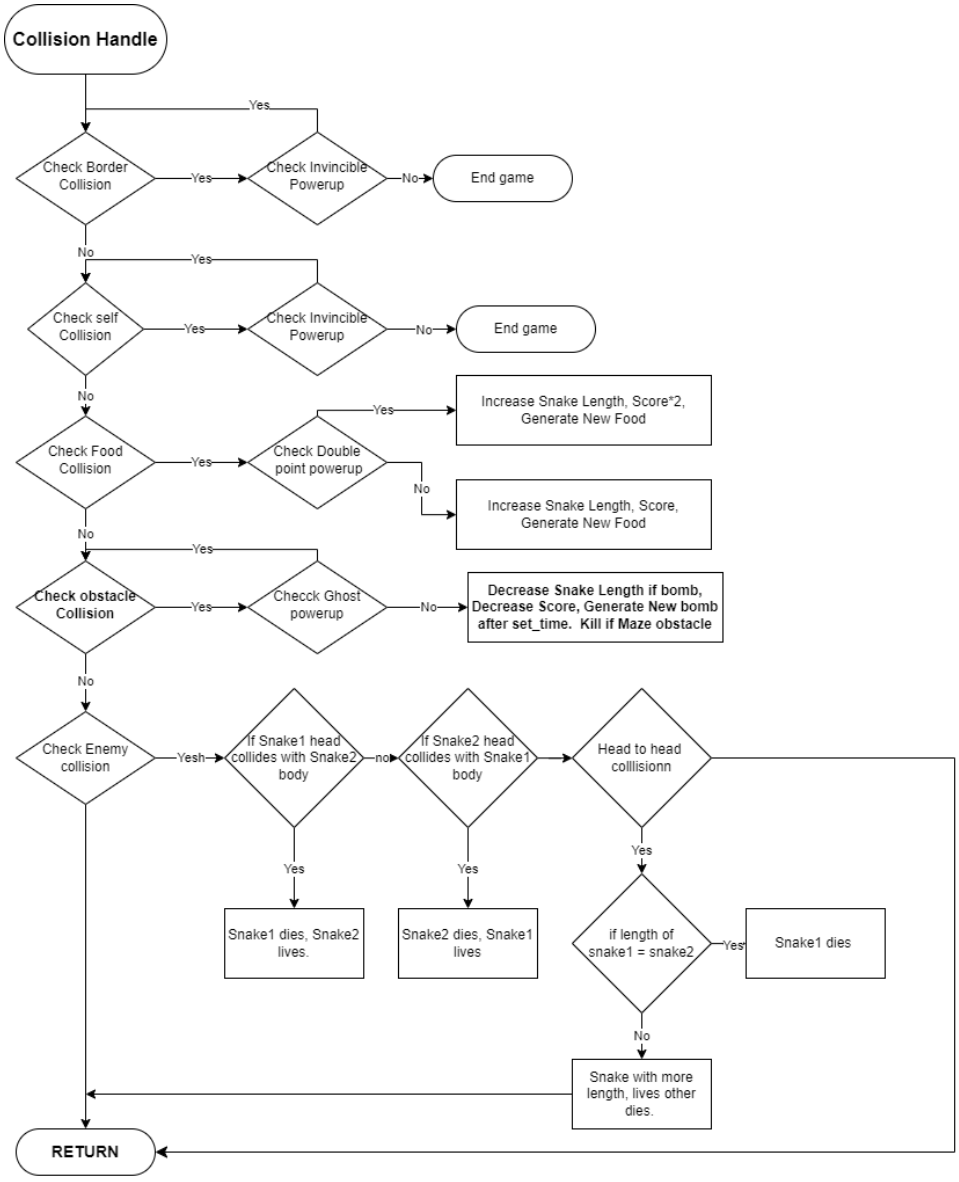

Figure 3: Diagram representing the collision handling mechanism

Collision Handling:

The above flowchart illustrates the collision handling logic, detailing how various types of collisions are processed to determine game outcomes. The process begins by checking if the snake has collided with the border. If a border collision occurs, the game checks whether the snake has an invincibility powerup; if not, the game ends. Next, it checks for self-collision, again considering the invincibility state—without it, the game ends upon self-collision.

6.3 Snake Movement

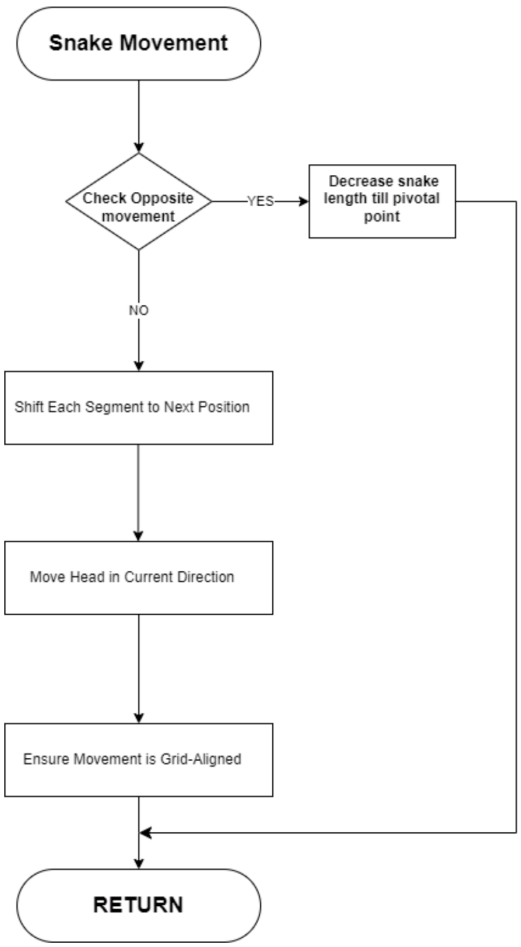

Figure 4: Illustration of the snake movement logic

Snake Movement:

The above flowchart explains the logic behind snake movement in the game. It begins by checking if the snake is trying to move in the opposite direction. If the answer is yes, the snake's length is reduced up to a pivotal point, and the rest of the movement steps are skipped. If not, each segment of the snake shifts to the position of the one ahead of it. The snake's head then moves in the current direction. Finally, the movement is adjusted to stay aligned with the grid before returning control to the main game loop. This ensures smooth and valid movement throughout the game.

6.4 Food Generation

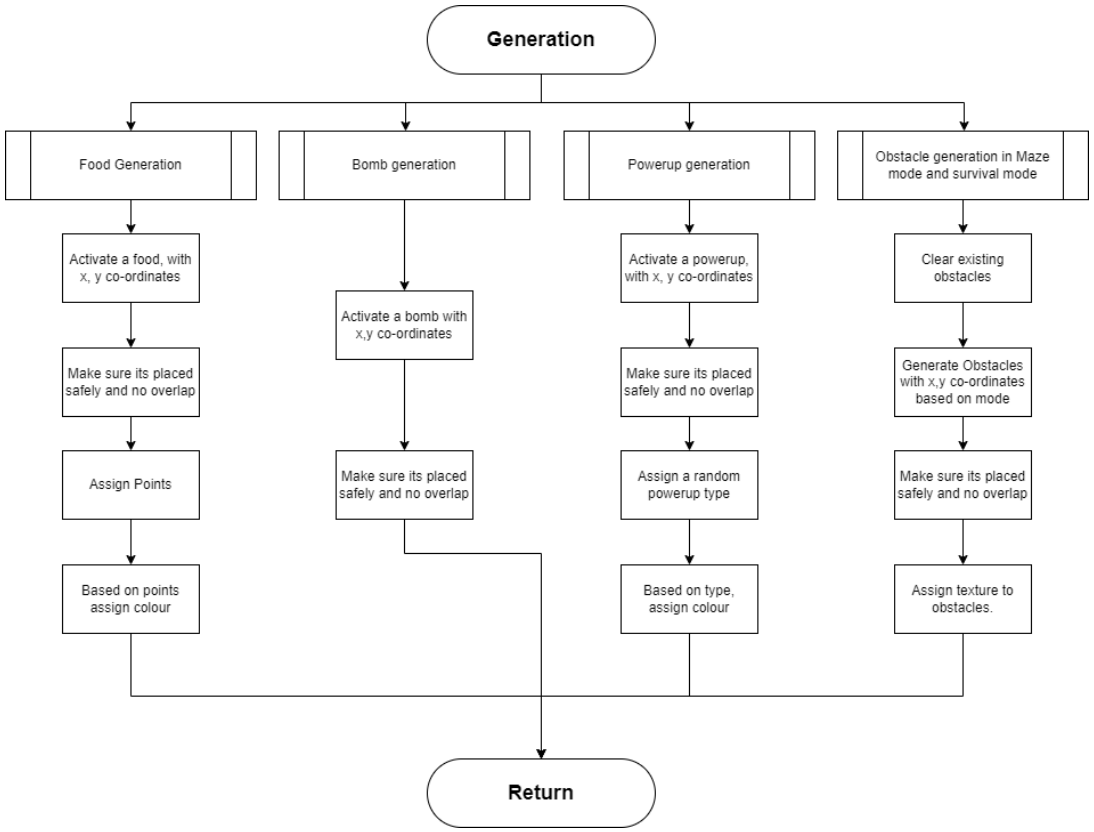

Figure 5: Diagram showing the food generation process

Food Generation:

The above flowchart shows the generation process. This process in the game involves creating new interactive elements such as food, bombs, powerups, and obstacles. When generating food, an inactive food item is activated, placed safely on the game field, assigned a point value between 1 and 3, and given a color that corresponds to its point value. For bombs, an inactive bomb is simply activated and placed in a safe position. Powerups are generated by activating an inactive powerup, assigning it a random type, setting a fixed duration, and placing it safely on the field with a color that reflects its type. Obstacle generation involves first clearing any existing obstacles, then creating new ones based on the current game mode—either as maze-style patterns or randomly placed structures—before assigning textures to all the newly created obstacles.

6.5 Complete Game Flow

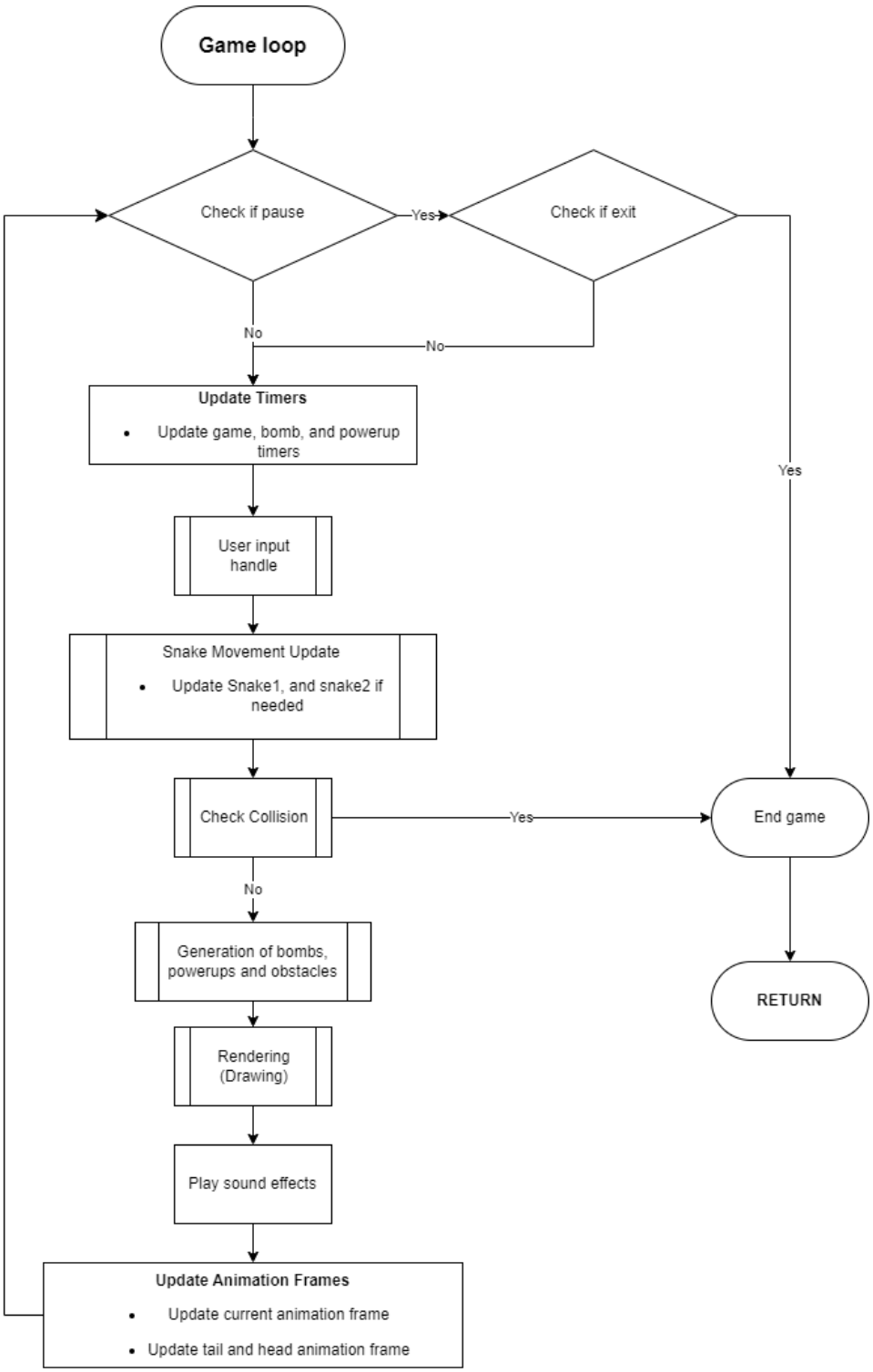

Figure 6: Flowchart presenting the complete gameplay loop

6.6 Rendering Process

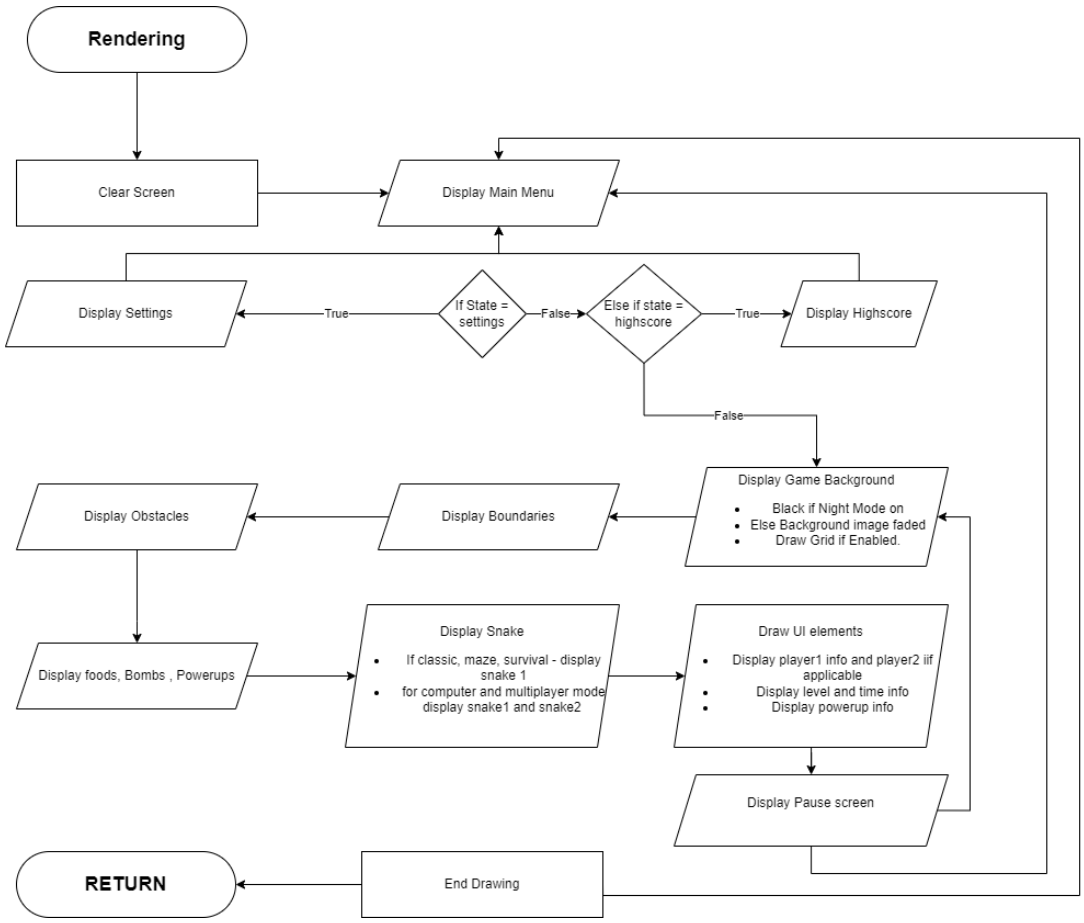

Figure 7: Visualization of the rendering process

Rendering process:

The above flowchart represents the rendering process. It begins with clearing the screen and then displaying the main menu. From there, the rendering pathway branches based on the current state of the game. If the state is set to "settings", it displays the settings menu; if it is "highscore", the highscore screen is shown. If neither, the flow proceeds to render the actual gameplay environment. Next, the boundaries and obstacles are drawn, followed by the rendering of food items, bombs, and powerups. Finally, the pause screen is displayed if triggered, and the rendering ends with the final drawing before returning from the routine.

6.7 User Input Handling

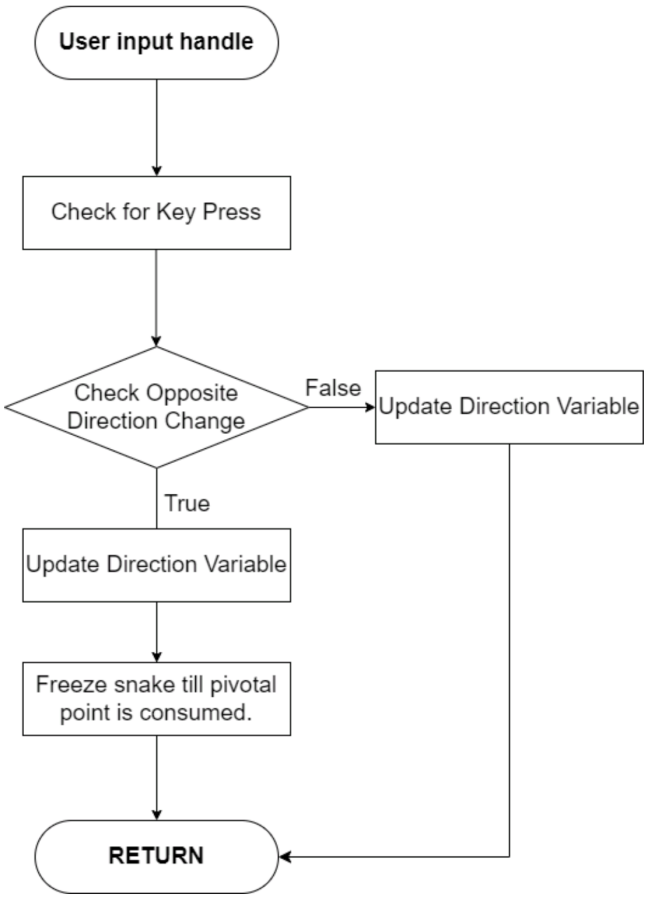

Figure 8: Flowchart demonstrating User Input Handling

User Input Handling:

The above flowchart represents the user input handling system in the snake game. The process begins with checking for any key press by the player. Once a key press is detected, it checks if the new direction is opposite to the current one. If the direction is not opposite, the direction variable is updated immediately. If the direction is opposite, the game still updates the direction but adds a freeze mechanism, which prevents the snake from instantly reversing until the pivotal point is consumed. This mechanism helps maintain logical movement and prevents self-collision bugs that can occur from sudden reverse inputs. Once the input is processed, the system returns control to the main loop.

7 Modules with Constraints

7.1 Initialization Module

The **Initialization Module** is responsible for setting up the game environment, loading resources, and preparing all necessary components before gameplay begins. The game window is initialized with a fixed size of **1200×680 pixels** (defined by `WIDTH` and `HEIGHT`), ensuring sufficient space for the snake, obstacles, and UI elements. The game uses a grid-based system where each cell is **20 pixels** (`TILE_SIZE`) in its logical representation but **40 pixels** (`SCALED_TILE_SIZE`) when rendered, allowing for smooth movement while maintaining a structured grid.

Before gameplay starts, the module loads all required assets, including textures for the snake's body, head, and tail, as well as food, bombs, power-ups, obstacles, and background images. Sound effects (such as eating, crashing, and power-up activation) and background music are also loaded here. The snake is initialized with a starting length of **5 segments**, positioned at opposite ends of the screen in multiplayer mode, and given an initial movement speed (`moveInterval = 0.1 seconds`). The module ensures that textures are successfully loaded before rendering begins and prevents the snake from exceeding the maximum allowed length (`MAX_SNAKE_LENGTH = 500`).

Additionally, this module includes a **safe position generation system** that ensures newly spawned food, bombs, and power-ups do not overlap with obstacles or the snake itself. If no valid position is found after multiple attempts, it defaults to a central location to prevent infinite loops.

7.2 Input Handling Module

The **Input Handling Module** processes player controls and ensures smooth, responsive movement while enforcing game rules. Player 1 controls the snake using the **arrow keys (UP, DOWN, LEFT, RIGHT)**, while Player 2 (in multiplayer mode) uses the **WASD keys**. The module prevents the snake from making abrupt reversals—if the player tries to move in the exact opposite direction, the snake enters a **reversal state**, where it shrinks in length as a penalty.

Only **one key press is processed per frame**, preventing erratic movement and ensuring the snake moves in a predictable manner. The module also handles menu navigation, where players use **UP/DOWN keys** to select options and **ENTER** to confirm. Special keys like **P (pause)** and **M (return to menu)** are processed separately to manage game states without interfering with gameplay.

7.3 Game State Update Module

The **Game State Update Module** manages the core gameplay mechanics, including snake movement, AI behavior, and dynamic difficulty scaling. The snake moves in **discrete grid steps (40 pixels)** and cannot move diagonally—only in four directions (up, down, left, right). Movement is controlled by a timer (`moveInterval`), ensuring the snake advances at a consistent speed.

In **vs. Computer mode**, the AI-controlled snake intelligently tracks the nearest food while avoiding obstacles and walls. It calculates the shortest path and adjusts its direction to prevent collisions. The game dynamically increases difficulty as the player

progresses—each level reduces the snake's movement interval (making it faster), spawns more food and bombs, and introduces new obstacles in **Survival and Maze modes**.

The module also handles **power-up effects**, such as temporary speed boosts, invincibility, and ghost mode (allowing the snake to pass through walls). These effects are tracked with timers and visually indicated by flashing animations.

7.4 Collision Handling Module

The **Collision Handling Module** enforces game rules related to interactions between the snake, environment, and other objects. If the snake collides with **screen borders**, it dies unless it has the **ghost power-up**, which allows it to wrap around the screen. Similarly, hitting an **obstacle** results in death unless the snake is invincible.

Self-collision (the snake's head touching its body) is fatal unless the player is reversing (where the snake instead loses segments). When the snake collides with **food**, it grows in length, and the player earns points (doubled if a power-up is active). **Bombs** reduce the snake's length by two segments and deduct points, while **power-ups** grant temporary abilities.

In **multiplayer mode**, the module checks for **snake-to-snake collisions**, where the head of one snake touching the other's body results in death (unless invincible). The game ends when all snakes die, and the winner is determined by the highest score.

Each module operates independently but interacts seamlessly to create a polished and challenging Snake game experience. The strict separation of concerns ensures maintainability, while the dynamic difficulty and power-up systems add depth to the classic gameplay.

8 Implementation

8.1 Data Organization

The snake game's data organization follows a structured approach that efficiently manages game entities, state, and rendering components. The implementation leverages Raylib's built-in data types while introducing custom structures to represent game-specific elements.

8.1.1 Snake Data Structure

The snake is implemented as a composite structure containing multiple members that collectively define its state and behavior. The primary structure includes an array of Vector2 positions to track each segment's location on the grid, with the head always at index 0. Movement is controlled through direction vectors (current and previous) that enforce the four possible orthogonal directions (up, down, left, right). The structure also contains timing mechanisms (moveTimer and moveInterval) that regulate movement speed, allowing for smooth, frame-rate independent updates. Additional state flags track whether the snake is alive, reversing direction, or under power-up effects (ghost mode, invincibility). The visual representation is handled through separate texture references for the head, body segments, and tail, with color properties for customization between players.

8.1.2 Game Board Representation

Rather than using a traditional 2D array, the game implements the play area through a combination of boundary definitions and obstacle structures. The board boundaries are fixed at 40 pixels from each screen edge, creating an effective play area of (WIDTH-80, HEIGHT-80). Obstacles are stored as an array of rectangle structures, each containing position, dimensions, and texture references. This approach allows for dynamic obstacle generation in different game modes (particularly in Survival and Maze modes) without requiring a full grid representation. Collision detection operates directly against these geometric boundaries rather than grid cells, providing more flexibility in obstacle shapes and sizes.

8.1.3 Food Items Management

Food items are organized as an array of structures, each containing position (Vector2), active status, point value, and visual properties. The game supports multiple concurrent food items (up to MAX_FOODS) with varying point values (1-3) indicated by different colors. Each food item maintains its own texture reference, though in practice they share a common texture with color modulation. The active flag allows for efficient reuse of food instances - when eaten, the flag is simply set to false, and the generation system finds the next available inactive slot when spawning new food. Position validation ensures foods spawn in valid locations through the safe position generation system that checks against obstacles and snake segments.

8.1.4 Game State Management

The game state structure encompasses all meta-information about the current gameplay session. This includes player scores (tracked separately for multiplayer), current level, and various timers for difficulty progression and power-up durations. Status flags control the overall game flow (MAIN_MENU, GAMEPLAY, PAUSE, etc.) and are used to determine which systems should be updated and rendered. The state also manages game progression through level increments that trigger difficulty increases - faster movement, more obstacles, and additional simultaneous food items. High score information is maintained separately but linked to the game state for transitions between menu and gameplay screens. The structure serves as the central coordination point between all other data components, ensuring proper synchronization of game rules and visual feedback.

8.1.5 Supplemental Data Structures

Additional supporting data structures enhance the game's functionality. Power-ups follow a similar pattern to food items but include type identifiers (speed, invincibility, etc.) and duration timers. Bombs are simpler structures containing just position and active status. The high score system uses an array of structures with name, score, and game mode information, persisted to disk between sessions. All these elements are carefully sized with MAX_ constants (like MAX_BOMBS, MAX_POWERUPS) to ensure memory safety while providing enough capacity for gameplay needs. The organization reflects a balance between flexibility (dynamic spawning) and performance (pre-allocated arrays), with careful attention to data locality for rendering efficiency.

8.2 Library Used

raylib.h - The Core Game Development Library

The game heavily relies on raylib.h, a simple and easy-to-use library for game programming that provides comprehensive functionality for cross-platform graphics rendering and input handling. This header file gives the game access to powerful 2D rendering capabilities through an abstraction layer that works across Windows, Linux, and macOS. The library handles all low-level graphics operations including texture loading (via LoadTexture()), drawing (DrawTexturePro(), DrawRectangle(), etc.), and color management. For input handling, raylib provides direct access to keyboard states (IsKeyPressed()) and gamepad input, enabling responsive controls for both single-player and multiplayer modes. The library's audio subsystem (accessed through LoadSound() and PlaySound()) manages sound effects and background music playback. Additionally, raylib's window management system (InitWindow()) creates and maintains the game window, while its timing functions help regulate the game's frame rate for smooth animation and movement.

stdlib.h - Fundamental Utility Functions

The stdlib.h header provides essential general-purpose functions that form the backbone of many game systems. Most critically, it offers memory allocation routines (malloc(), free()) though these aren't explicitly used in this implementation due to the game's static allocation approach. The library's random number generation (rand()) works in conjunction with time.h to create varied gameplay elements, while exit() provides a clean way to terminate the program. Conversion functions like atoi() could be used for parsing high scores or configuration data. The header also contains constants like NULL and EXIT_SUCCESS/FAILURE that help write more portable and robust code. While not all of stdlib's capabilities are utilized in this game, it remains an essential inclusion for core C functionality.[2]

time.h - Randomness and Timing Operations

The time.h library serves two primary purposes in this implementation. First and foremost, it provides time() which seeds the random number generator (srand()) at program startup, ensuring different sequences of random numbers across game sessions - crucial for generating random food positions, power-up types, and obstacle layouts. The header also contains clock() and related timing functions that could be used for more precise frame timing, though the game primarily uses raylib's timing functions instead. The time-related structures (like tm) aren't used here but could be helpful for adding timestamps to high scores or implementing time-based gameplay mechanics in future expansions.

stdio.h - File Handling and Data Persistence

Through stdio.h, the game implements persistent storage of high scores using basic file operations. The fopen(), fread(), fwrite(), and fclose() functions handle the binary reading and writing of high score data to disk, allowing player achievements to persist between game sessions. While the implementation uses simple binary I/O for efficiency, stdio.h also provides formatted I/O functions (fprintf(), fscanf()) that could be used for more human-readable score files. The header's file positioning functions (fseek(), ftell()) might

be useful for more advanced score management systems. Error handling for file operations could be enhanced using ferror() and feof(), though the current implementation keeps it simple.[1]

string.h - String Manipulation Utilities

The string.h library plays a crucial role in handling player names and string operations throughout the game. Its most used function here is strcpy() for copying player names into the high score structure, ensuring proper memory management when storing textual data. The header also provides strcmp() which could be used for sorting high scores alphabetically, though the current implementation sorts by score value only. String concatenation (strcat()) and length measurement (strlen()) functions are available for any future needs involving more complex string manipulation. While the game currently uses simple string operations, string.h offers a full suite of functions for case conversion, searching, and comparison that could be valuable for adding features like player name validation or more sophisticated menu systems.

math.h - Mathematical Operations and Calculations

The math.h library provides essential mathematical functions that support several game mechanics. Most prominently, sqrt() enables distance calculations through the Pythagorean theorem (used in vector2distance()), which is fundamental for collision detection between game objects. The fmax() function helps ensure scores never go negative when deducting points for bomb collisions. While not used in the current implementation, math.h offers a wide range of other potentially useful functions: trigonometric functions (sin(), cos()) for advanced movement patterns, rounding functions (floor(), ceil()) for precise positioning, and power functions (pow()) for more complex calculations. The library's absolute value function (fabs()) could be helpful for certain collision or AI calculations, though the current implementation uses simpler comparisons where possible.

8.3 User Interface Designs

Menu Screen:
The following figure shows the menu screen of the game. The menu screen serves as the primary interface where players interact with the game before starting or resuming gameplay. It typically includes options such as **Start Game**, **Settings**, **High Scores**, and **Exit**. When the game launches, the menu screen is displayed, allowing the player to navigate through different selections using keyboard inputs (e.g., arrow keys) or mouse clicks.

The menu screen operates using a **state-based system**, where each option is associated with a specific action. For instance, selecting **Start Game** transitions the game to the initialization phase, where game objects like the snake, food, and obstacles are set up. The **Settings** option allows players to adjust configurations such as difficulty level, controls, or audio preferences. The **High Scores** section displays the top player rankings, often retrieved from a saved file or database. Finally, the **Exit** option closes the application.

To enhance user experience, the menu screen may include visual feedback, such as highlighting the currently selected option or playing sound effects when navigating. Some games also incorporate animations or background themes to make the menu more engaging. The implementation typically involves event handling, where user inputs trigger corresponding functions to load different game states or modify settings.

Figure 9: Game Menu

Game Screen:

The gameplay screen in Classic Mode is the core interactive interface where the player controls a snake to collect food and grow in length while avoiding collisions. The screen consists of a bordered rectangular play area, typically rendered with a grid-based layout, where the snake moves in four primary directions: up, down, left, and right. The player controls the snake using keyboard inputs (usually arrow keys or WASD), and each movement updates the snake's position in real-time.

At the start of the game, the snake is initialized with a default length (often three segments) and placed at a predefined position, such as the center of the screen. Food items, represented as small objects (egg), spawn randomly within the playable area. When the snake's head collides with food, the snake grows longer, and the player earns points. The food then respawns at a new random location, ensuring continuous gameplay.

The primary challenge in Classic Mode is avoiding collisions. If the snake's head hits the boundaries of the play area or its own body, the game ends. The screen typically displays the current score, high score, and sometimes a timer or level indicator. Some variations may introduce obstacles or speed increments as the player progresses, increasing difficulty.

The game loop continuously checks for collisions, updates the snake's position, and refreshes the display. If a game-over condition is met, the screen transitions to a summary display showing the final score and an option to restart or return to the menu. The Classic Mode gameplay screen is designed to be simple yet engaging, relying on smooth controls, responsive mechanics, and clear visual feedback to maintain player immersion.The following figure displays the classic mode game screen.

Figure 10: Game Screen

Game Over Screen:

The Game Over screen in Classic Mode appears when the player's snake collides with either the boundaries of the play area or its own body. This screen serves as the termination point of the current gameplay session, providing feedback to the player about their performance while offering options for subsequent actions. When triggered, the game immediately freezes all moving entities and displays the Game Over screen overlay on top of the final game state.

The screen prominently displays the player's final score, typically in large, bold text to emphasize the achievement. Alongside the current score, many implementations also show the player's personal best or all-time high score for comparison. This comparative scoring system helps motivate players to improve their performance in subsequent attempts. The visual design often includes a darkened or semi-transparent overlay that partially obscures the game board beneath while keeping it visible as context for the ending scenario.

Below the score display, the Game Over screen presents the player with several interactive options. The most common choices include a **Restart** button that immediately begins a new game with fresh initialization, and a **Main Menu** button that returns the player to the game's primary navigation interface.

The screen also frequently incorporates visual and auditory feedback elements to enhance the player experience. A distinctive game over sound effect often plays upon screen appearance, and the text may feature animation effects such as fading in or growing from small to large size. Some implementations include particle effects or other visual flourishes

to mark the end of the game session. These elements combine to create a satisfying conclusion to the gameplay experience while maintaining engagement for subsequent plays.

Figure 11: Game Over Screen

Multiplayer:

The multiplayer game screen provides a shared gameplay environment where two players can compete simultaneously on the same display. This screen extends the basic functionality of the classic single-player mode while introducing additional mechanics to support competitive interaction. When initialized, the screen divides its attention between two snake entities, each with independent controls and scoring systems. The visual design typically employs color differentiation to help players distinguish their snake from their opponent's, with common color pairings i.e. red vs green.

Player controls are mapped to separate keyboard sections to allow simultaneous input. Player 1 traditionally uses the **WASD** keys for movement, while Player 2 utilizes the **arrow keys**. The game engine processes these inputs independently, updating each snake's position within the same game cycle. This simultaneous processing creates a dynamic competitive environment where both players must navigate the shared space while interfering with each other's progress. The play area remains a single unified grid, meaning both snakes interact with the same food items and obstacles.

The scoring system displays both players' current scores prominently at the top or sides of the screen. In most implementations, each snake collects food independently, with collisions between snakes resulting in a game over for the colliding player. Some variations introduce special power-ups that can temporarily affect the opponent's snake, such as speed reductions or control reversals. The game continues until one player loses, at which point the screen typically declares the surviving player as the winner while offering a rematch option. The following figure shows the multiplayer mode gameplay.

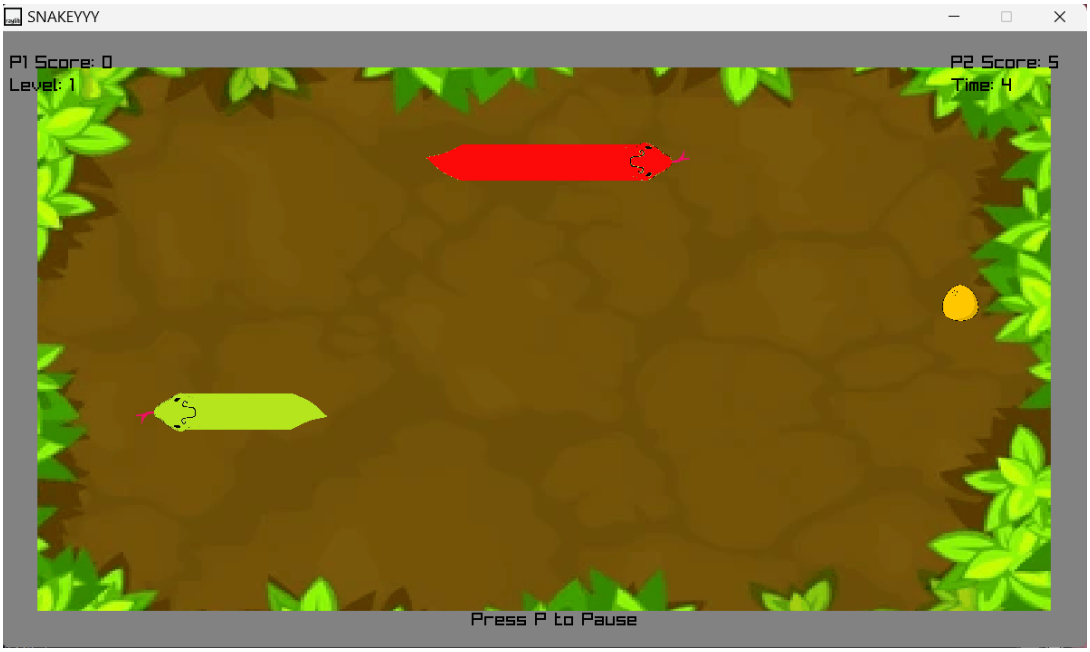

Figure 12: Multiplayer Mode

Survival Mode:

The survival mode implementation focuses on three core systems that modify the base gameplay through specific programmed behaviors and algorithms.

The bomb spawning system operates through a managed list of active bombs and a dedicated spawn timer. When this timer reaches the current spawn interval, which initially begins at 10 seconds, the system executes several coordinated actions. First, it performs validation checks to identify suitable spawn locations that avoid existing snake segments and food items. After identifying a valid position, the system instantiates a new bomb entity at the selected coordinates and initiates the bomb's active timer with a default duration of 5 seconds. Finally, the spawn timer resets using the current interval value, preparing for the next spawn event.

Difficulty progression follows a time-based escalation pattern tied directly to the player's survival duration. Every 30 seconds of continuous gameplay triggers a difficulty increase that modifies multiple parameters simultaneously. The spawn interval between bombs decreases by 1 second with each increment, bottoming out at a minimum of 3 seconds to maintain playability. Concurrently, the active duration of each bomb decreases by 0.5 seconds per difficulty level, reaching a minimum active time of 2 seconds. Additionally, the snake's movement speed increases by 5% with each difficulty step, capping at a maximum 30% increase over the base speed to preserve control responsiveness.

The scoring implementation combines traditional food collection with time-based bonuses through a specific mathematical formula. The total score calculation incorporates both the sum of all collected food values and the elapsed survival time, with the time component serving as a multiplier. This relationship is expressed mathematically as $TotalScore = FoodPoints \times (1 + \frac{SurvivalTime}{60})$, where FoodPoints represents the cumulative value of all collected food items and SurvivalTime measures the duration in seconds. This approach rewards both successful navigation and endurance while maintaining balance between the two scoring components.In addition to this powerups that grant special

abilities are also generated. The powerups give any of the folwing powers: Speed, Ghost, Double Point and Invincibility. The following figure shows the survival mode gameplay.

Figure 13: Survival Mode

Dark Mode:

The dark mode functionality transforms the visual presentation through a systematic color scheme inversion while preserving all game mechanics. Activation triggers a comprehensive palette change that affects every visual component, from background surfaces to text elements, creating optimal contrast for low-light environments. This implementation uses centralized color variables that all interface elements reference, enabling instantaneous theme switching without gameplay interruption or layout recomputation.

Color transformations follow specific design principles to maintain usability. The game grid background shifts from light to a deep charcoal , with grid lines appearing in a subtle gray for structural clarity. Text elements invert their standard coloring, rendering in light gray on dark surfaces instead of traditional black-on-white.

A persistence layer stores the user's preference in local storage, automatically reapplying the selected mode during subsequent sessions. This system checks the stored preference during initialization before rendering begins without requiring repeated manual activation. The following figure shows the dark mode representation.

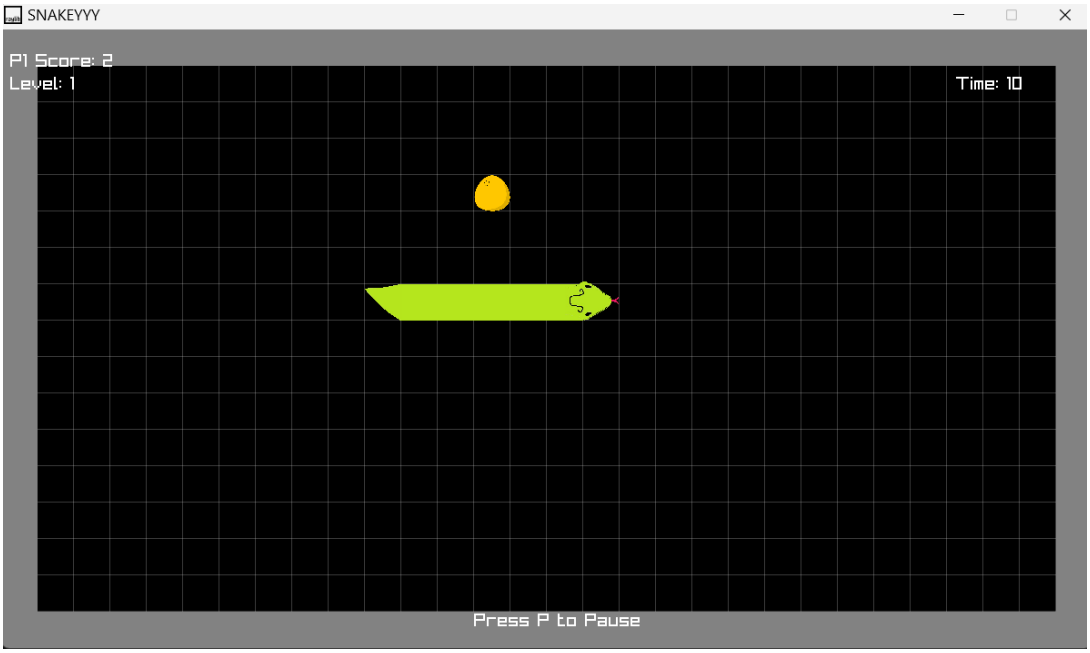

Figure 14: Dark Mode

Settings Screen:

The settings screen implementation provides a comprehensive interface for game customization through several interconnected systems. When initialized, the screen loads all available configuration options into a managed state object that preserves default values while tracking current modifications. Each setting renders as an appropriate interactive control element bound directly to these state variables, creating immediate feedback loops for user adjustments.

Audio configuration utilizes range sliders connected to the game's audio mixer subsystem, allowing real-time adjustment of background music and sound effect volume levels. These controls display percentage values and trigger immediate audio level changes when manipulated. The visual options section includes theme selection, resolution preferences, and frame rate controls, with most changes applying instantly except for resolution modifications which require explicit confirmation to handle the display mode change properly.

Control customization features a dynamic key binding interface that captures keyboard input when the user attempts to rebind actions. This system includes conflict detection to prevent duplicate bindings and visual feedback during the reassignment process. All setting modifications remain temporary until the user explicitly confirms changes, at which point the system serializes the configuration state to persistent storage and applies any required system-level adjustments.

The implementation includes robust error handling for storage operations and input validation, ensuring configuration integrity across sessions. A cancel button reverts all unconfirmed changes by reloading the last persisted state, while the apply button triggers the serialization and application process. The screen maintains clean separation between presentation logic and core game systems, interacting only through well-defined interfaces to prevent unintended side effects on active gameplay.The following figure shows the settings available in the game.

Figure 15: Settings Screen

8.4 Platform Used for Code Development

- **Development Environment:** Visual Studio Code
- **Compiler:** Notepad++ installed from [4]
- **Operating System:** Windows 11

9 Observations with Respect to Society

The game provides nostalgic value for those who played classic Snake games in Nokia phones. Simple gameplay makes it accessible to all age groups. Demonstrates how classic games can be modernized with new features. Educational value in learning game development concepts. The game was tested with other department students and received positive reviews.

10 Legal and Ethical Perspectives

This project is an original implementation inspired by a classic game concept, reimagined with a fresh approach while staying true to the charm of the original. It features simple and intuitive controls, making it easily accessible and enjoyable for players of all ages and skill levels. The game is designed with user privacy in mind—no personal data is collected at any stage, ensuring a safe and secure experience. Visual assets used in the game are credited to Pixabay.com[5], a platform known for providing high-quality, royalty-free images.

11 Limitations and Future Enhancement

Current Limitations:

- Limited graphical capabilities
- No background music
- No difficulty level for playing against computer

Future Enhancements:

- Add online multiplayer mode
- Upgrade snake with increase in level
- Add level progression with different maps

12 Learning Outcomes

- Practical experience with C programming
- Understanding of game loops and real-time systems
- Creating spritesheets for animation using pngs
- Creating animations for snake's head and tail
- Understanding collision detection by using x and y coordinates

References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1988.

[2] H. M. Deitel and P. J. Deitel, *C How to Program*, 8th ed. Upper Saddle River, NJ, USA: Pearson, 2016.

[3] "The Snake Game Algorithm Explained," *GeeksforGeeks*. [Online]. Available: `https://www.geeksforgeeks.org/snake-game-introduction/`. [Accessed: Feb. 17, 2025].

[4] "Raylib: A simple and easy-to-use library," *Raylib*. [Online]. Available: `https://www.raylib.com/`. [Accessed: Feb. 17, 2025].

[5] "Stunning Free Images & Royalty Free Stock," *Pixabay* [Online]. Available: `https://pixabay.com/`. [Accessed: Mar. 19, 2025]

[6] "Animation in Raylib," YouTube, Mar. 2, 2021. *CodeWithHarry* [Online]. Available: https://youtu.be/MeZaTMUU4Yc?si=caoXyHVYJGlW4pO0. [Accessed: Mar. 11, 2025].