Project Documentation

F1 Racing Game

Course: CL1002 – Programming Fundamentals Lab

Name: Muhammad Saleh

Instructor: Muhammad Behram Shah

Section: BCS – 1D

Table of Contents

Executive Summary

This report presents an in-depth academic and technical analysis of a complete authentication system and racing game implemented in the C programming language. The system integrates secure password masking, persistent user data storage, dynamic ASCII-based animation, obstacle generation, real-time rendering, fuel consumption logic, and state-based game transitions. The purpose of this project is to demonstrate a blend of theoretical PF concepts with practical software engineering practices including modular design, console manipulation, and algorithmic thinking.

Introduction

The purpose of this semester's final PF project is to build a fully functional console application that integrates both a login system and an interactive racing game. The project demonstrates procedural programming paradigms, use of the Windows Console API, file handling, and dynamic runtime behavior. The system is designed to mimic real-world login workflows and game loops.

Objectives

The objectives of the project include:

- Implement a secure user registration and login system.
- Demonstrate file handling using persistent storage.
- Implement ASCII-based real-time game rendering.
- Apply PF concepts: loops, conditions, arrays, strings, pointer use, and functions.
- Maintain score logs and retrieve top players.
- Provide clean separation of responsibilities through modules.

System Overview

The entire system is divided into two primary modules: the Authentication Module and the Game Module. The Authentication Module handles user creation, login, credential verification, and secure password input. The Game Module handles rendering, scoring, collision detection, and dynamic obstacle behavior. This layered design improves readability, maintainability, and debugging ease.

Overall Architecture

The system follows procedural architecture with modules interacting through shared variables. Authentication must succeed before the game loop is unlocked. The game engine updates state variables such as fuel, score, and obstacle positions in real time, while rendering to the console using cursor manipulation. The architecture includes:

- Input subsystem
- File subsystem
- Game loop engine
- Rendering subsystem
- Collision engine
- Event handling system

Module Documentation

This section documents the core modules including g_pass(), signup(), login(), menu controller, and rendering utilities. Each function serves a distinct role, allowing modular analysis and incremental debugging. This structure reflects Programming Fundamentals subject's best practices.

I. Header Files

Following are the header files used in program for different purposes.

- stdio.h – Input/output & file handling
- stdlib.h – Memory, exit(), rand()
- conio.h – Keyboard handling
- windows.h – Console control, Sleep, Beep
- time.h – Random seed using time()

1. stdio.h

- **Purpose:** Standard Input Output Library
- Used for input and output operations.
    - printf() – displays output on the screen
    - scanf() – reads input from the user
- It allows communication between the program and the user.

2. stdlib.h

- **Purpose:** Standard library for general utilities.
- Provides functions for:
    - Dynamic memory management: malloc(), free().
    - Program control: exit().
    - Random numbers: rand(), srand().
- Randomly spawning obstacles and power-ups.

3. string.h

- **Purpose:** String handling library.
- Provides functions to manipulate strings:
  - strcpy() → Copy string.
  - strcmp() → Compare strings.
  - strlen() → Get string length.
  - strcat() → Concatenate strings.
- Comparing usernames and passwords.
- Storing and validating player inputs.

4. conio.h

- **Purpose:** Console Input/Output library (non-standard, mostly Windows).
- Provides functions for real-time keyboard handling, like:
  - getch() → Get a character without waiting for Enter.
  - getche() → Get a character and display it immediately.
  - kbhit() → Check if a key is pressed.

5. windows.h

- **Purpose:** Windows API header for platform-specific functions.
- Provides functions for:
  - Console manipulation: SetConsoleCursorPosition(), SetConsoleTextAttribute().
  - Sleep and Beep: Sleep(), Beep().
- Controlling cursor visibility and colors.

6. time.h

- **Purpose:** Time library for handling date and time operations.
- time(NULL) → Import actual current time.

II. Console Control Concept

Windows Console API allows cursor movement and color manipulation. This enables smooth animation and UI effects in console games.

1. HANDLE Console = GetStdHandle(STD_OUTPUT_HANDLE);

- HANDLE is a data type used by Windows to represent system objects.
- GetStdHandle(STD_OUTPUT_HANDLE) gets a handle to the console output screen.
- This line allows the program to control the console window (cursor, text, etc.).

2. COORD cursorpos = {0, 0};

- COORD is a structure that stores X and Y coordinates.
- {0,0} represents the top-left corner of the console screen.
- Used for setting or moving the cursor position.

3. CONSOLE_CURSOR_INFO cursorinfo;

- CONSOLE_CURSOR_INFO is a structure that stores cursor information.
- It contains:
- dwSize → cursor size
- bVisible → cursor visibility (true/false)
- This variable will store current cursor settings.

4. GetConsoleCursorInfo (Console, &cursorinfo) ;

- Retrieves the current cursor settings of the console.
- Stores them in cursorinfo.
- Needed so we can modify the cursor properties safely.

5. cursorinfo.bVisible = 0;

- bVisible = 0 means cursor is hidden.
- 1 would mean visible.
- This hides the blinking cursor from the console.

6. SetConsoleCursorInfo (Console, &cursorinfo);

- Apply the modified cursor settings to the console.
- Use the console handle and updated cursor information.
- Final step that hides the cursor.

III. Game Initialization Logic

All flags, counters, and arrays are reset before the race starts to ensure a clean game state.

1. int road_l = 10;

- Length of the road (number of rows or blocks).
- Used to control how long the road appears on the screen.

2. int road_w[5] = {60, 66, 72, 78, 84};

- An array storing road widths or lane X-positions.
- Each value likely represents the center position of a lane.
- Total 5 lanes.

3. int c_lane = 2;

- Current lane of the car.
- Lane index starts from 0 to 4, so 2 means middle lane.

4. int fuel = 100;

- Fuel level of the car.
- Decreases as the game runs.

5. int speed;

- Speed of the car.
- Controls how fast the game updates or road moves.

6. int obs = 4;

- Total number of obstacles in the game.

7. int obs_r[obs], obs_l[obs];

- Arrays storing row and lane positions of obstacles.
- obs_r → obstacle row position
- obs_l → obstacle lane position

8. int p_obs, w_obs;

- p_obs → position of obstacle

    - w_obs → which obstacle (index) is being used

9. **int pwr_r = -1, pwr_l = 0, t_pwr = 0;**
    - Power-up position and status:
        - pwr_r → power-up row (−1 means inactive)
        - pwr_l → power-up lane
        - t_pwr → timer for power-up duration
10. **int s_pwr_r = -1, s_pwr_l = 0, s_t_pwr = 0;**
    - Secondary or special power-up variables:
        - s_pwr_r → special power-up row
        - s_pwr_l → special power-up lane
        - s_t_pwr → special power-up timer
11. **int menu, mode;**
    - menu → menu selection
    - mode → game mode (easy, medium, hard, etc.)
12. **int s_flag = 0, m_flag = 0;**
    - Flags used as Boolean variables:
        - s_flag → sound flag
        - m_flag → menu or mode flag
13. **int pits = 3;**
    - Number of lives or pit stops available.
14. **int rl_keyd;**
    - Stores keyboard input (arrow keys, etc.).
15. **int spawn_t;**
    - Spawn timer for obstacles or power-ups.
16. **int score = 0;**
    - Player score.
17. **int crash, end;**
    - crash → checks if collision happened
    - end → checks if game is over
18. **int i_laps, i_load, i_exit, i_space1, i_space2, i_frame = 0, i_beep, i_obs;**
    - Loop control variables:
        - i_laps → laps counter
        - i_load → loading animation loop
        - i_exit → exit loop
        - i_space1, i_space2 → spacing for drawing
        - i_frame → frame counter
        - i_beep → sound loop
        - i_obs → obstacle loop

19. srand((unsigned) time(NULL));

- Initializes the random number generator.
- Ensures different obstacle positions each time the game runs.
- Set the current seed.

20. int choice, logged_in = 0;

- choice → menu choice selected by the user
- logged_in → login status (0 = not logged in, 1 = logged in)

IV. Rendering & Animation

Nested loops simulate rows and columns of the road. Frame-based rendering creates motion illusion.

1. Menus Rendering:

- Displays Login Menu and Game Menu using formatted printf.
- Options are numbered for user interaction (Sign Up, Log In, Start Game, Pit Stop, View Stats, High Scores, Game History, Log Out).
- Uses loops and input checks to ensure valid choices.

2. Animation & Loading Effects:

- Simulates engine loading using a progress bar: Loading Engine...[#####].
- Uses Sleep() to control animation speed.
- Beep() function is used to enhance feedback for actions like starting the game, crashing, or pit stops.

3. Dynamic Console Updates (Rendering Game Scene):

- Road and lanes rendered using loops for rows (road_l) and columns (road_w[]).
- Obstacles (XXX) and power-ups (+++) dynamically appear at random positions.
- Player's car (|/^\|) displayed in the correct lane.
- Updates the display every frame (i_frame) for animation effect.

4. Collision & Power-Up Animation:

- Detects collision with obstacles and triggers crash animation with sound (Beep).
- Detects collection of fuel or pit power-ups and animates their effect.

5. Real-Time Input Handling:

- Uses kbhit() and getch() to capture arrow keys and WASD for lane movement.
- Updates car position and fuel dynamically while the game loop runs.
- Allows pause/exit with ESC key.

6. Score & Fuel Animation:

- Displays score, fuel, and remaining pits in real time.
- Changes color of fuel display based on level (SetConsoleTextAttribute).

7. Menu System Navigation Animation:

- Smooth transitions between menus using system("cls") and delays (Sleep).
- Press ESC to return to the previous menu, ensuring fluid user experience.

8. High Score & History Rendering:

- Reads from games_log.txt to display sorted top scores with player names.
- Updates dynamically with current user's game history.

9. Overall Design:

- Combines text-based animation with real-time updates for interactive gameplay.
- Uses loops, timing, cursor control, and sound to create an engaging console menu and animation system.

V. Obstacle & Power-up System

Random spawning using rand() and timers. Power-ups refill fuel when collected. The system uses timers to control the spawning of obstacles and power-ups, preventing them from appearing too frequently.

- Obstacles are activated only after a fixed delay to maintain balanced gameplay difficulty.
- Each obstacle is assigned a random lane using rand()%5, which adds unpredictability to the game.
- Obstacles start just above the visible road and move downward frame by frame, creating a realistic motion effect.
- Once an obstacle reaches the end of the road, it is deactivated and reused, improving performance and memory efficiency.
- A normal power-up system spawns power-ups at regular intervals to reward the player.
- Power-ups also appear in random lanes, encouraging lane switching and strategic movement.
- If a power-up is not collected, it disappears after leaving the road area.
- A special power-up system uses a longer timer, making special power-ups rare and valuable.
- Obstacles and power-ups are rendered using different symbols (XXX for obstacles and +++ for power-ups).
- Different console colors are used to clearly distinguish hazards from rewards.
- The entire system works on frame-based rendering, ensuring smooth animation and continuous gameplay.
- Overall, this system increases challenge interaction and replay value in the console-based racing game.

VI. Collision Detection

If obstacle row equals car row and lane matches, crash is triggered.

- A horizontal line is printed to visually separate the game status area from the road.
- The variable crash is initialized to 0, assuming no collision at the start of the frame.
- A loop runs through all obstacles to check for a possible collision.
- The condition checks whether an obstacle's row position matches the car's row (road_l - 1).
- It also checks whether the obstacle's lane matches the current lane of the car (c_lane).
- If both row and lane match, a collision is detected and crash is set to 1.
- Once a collision is detected, the loop breaks immediately to save processing time.
- On collision, sound effects (Beep) are played to indicate a crash.
- A Game Over message and the player's score are displayed on the screen.
- The player's score is saved to a file (games_log.txt) for record keeping.
- All game variables such as score, speed, fuel, pits, and flags are reset to initial values.
- The loop breaks, ending the current game session and returning control to the menu.

VII. Fuel & Speed Algorithm

Fuel reduces over time and movement. Speed increases difficulty.

- Fuel increases when the player collects a normal power-up (pwr_r == road_l-1 && pwr_l == c_lane).
- Special power-up increases the number of pits (lives) when collected.
- Each successful frame update increases the score, rewarding survival time.
- Lane movement (left/right) consumes fuel, encouraging careful driving.
- Invalid lane movement triggers a warning beep without changing position.
- Both WASD keys (A, D) and arrow keys (Left Arrow, Right Arrow) are supported for movement.
- Pressing ESC exits the race immediately.
- Fuel continuously decreases:
- Gradually over time (every fixed number of frames).
- Faster when the player changes lanes.
- When fuel reaches zero, warning beeps are played and the game ends.
- Game speed gradually increases difficulty by reducing the delay value over time.
- Faster speed results in quicker frame updates, making obstacle avoidance harder.
- The algorithm balances risk and reward by linking movement, speed, and fuel consumption.

VIII. File Handling & Logs

“accounts.txt” stores users and “games_log.txt” stores score history.

- The program uses file handling to store and retrieve persistent game data.
- ”accounts.txt” is used to store user account information such as usernames and login details.
- ”games_log.txt” is used to store game history, where each entry contains a username and the corresponding score.
- The file is opened in read mode ("r") to display the player’s past game scores.
- Data is read using fscanf() in a loop until the end of the file is reached.
- strcmp() is used to match the logged-in user’s name with stored records.
- Only the scores of the current user are displayed as game history.
- The file is opened in append mode ("a") to save new scores without deleting old data.
- After writing or reading, files are properly closed using fclose() to prevent data loss.
- This system provides persistent storage, allowing players to view their past performance even after restarting the game.

IX. Security & Input Validation

Password masking and bounded input are used.

- Password input is masked using getch(), so characters are not displayed on the screen.
- Bounded input is enforced using size-limited arrays (e.g., char username[51]) to prevent buffer overflow.
- Usernames are validated to disallow invalid characters such as commas (',') that could break file parsing.
- Empty username and password inputs are rejected to ensure valid credentials.
- Duplicate usernames are checked during signup to prevent account conflicts.
- Login attempts are limited to a fixed number of tries to reduce brute-force attempts.
- User input is validated in menus to handle invalid selections safely.

Limitations

Windows-only, ASCII graphics, no encryption.

- The program is Windows-only due to the use of windows.h, conio.h, Beep(), and Sleep().
- The game uses ASCII-based graphics, which limits visual quality and realism.
- Passwords are stored in plain text without encryption, posing a security risk.
- File-based storage (.txt files) is not suitable for large-scale or multi-user systems.
- Input handling relies on console behavior, which may vary across environments.

Conclusion

This project demonstrates solid C fundamentals and system programming.

- This project successfully implements a console-based racing game using the C programming language.
- It demonstrates strong fundamentals of C, including functions, loops, arrays, pointers, structures of logic, and file handling.
- A complete user authentication system is implemented with sign-up and login functionality using file storage ”accounts.txt”.
- Password masking improves basic security and enhances user experience during login and signup.
- The game uses Windows system programming features such as Beep(), Sleep(), cursor positioning, and color control to simulate animation and sound effects.
- Real-time input handling (kbhit() and getch()) enables smooth car movement and responsive gameplay.
- ASCII-based rendering and frame updates create a visual illusion of motion and a scrolling road.
- Core gameplay mechanics like collision detection, fuel management, speed progression, power-ups, and pit stops are logically designed and well-integrated.
- Difficulty levels dynamically control game speed, increasing replay value and challenge.
- Score tracking, high scores, and game history are maintained using file handling (games_log.txt).
- The project handles game state management effectively, including pause, crash handling, restart, and exit conditions.
- Input validation prevents common errors such as empty usernames, invalid characters, and incorrect menu selections.
- The overall structure reflects a mini game engine built entirely in C without external libraries.
- While limited to Windows and console graphics, the project clearly shows system-level programming concepts.
- This project is a strong academic example of combining programming logic, user interaction, and file-based data persistence in C.
