Jaypee Institute of Information Technology, Sector - 62, Noida

B.Tech CSE II Semester

SDF PBL Report

Car Dodge Game: A Raylib-Based Arcade Experience

Submitted to
Dr. Sulabh Tyagi, Dr. Anuja Shukla,
Dr. Amarjeet Prajapati

Submitted by

Mukul Aggarwal	2401030239
Ananya Srivastava	2401030230
Aastha Malik	2401030251

Letter of Transmittal

Dr. Sulabh Tyagi & Dr. Amarjeet Prajapati & Dr. Anuja Shukla
Department of Physics and Materials Science and Engineering
Jaypee Institute of Information Technology
Sector - 62, Noida

Subject: Submission of SDF PBL Report on "Car Dodge Game"

Dear Dr. Sulabh Tyagi & Dr. Amarjeet Prajapati & Dr. Anuja Shukla

We are pleased to submit our report titled *Car Dodge Game: A Raylib-Based Arcade Experience* as part of our B.Tech CSE II Semester coursework. This report provides an in-depth exploration of the development of a 2D arcade game using C++ and the Raylib library, focusing on its design, implementation, and innovative features such as dynamic gameplay mechanics, power-up systems (Shield, Magnet, Nitro), and a persistent leaderboard. It includes detailed technical discussions on collision detection algorithms, asset management strategies, user interface design, and the optimization techniques employed during the project.

We have strived to comprehensively document the development process, including the challenges faced and the solutions implemented. We hope this report meets your expectations.

Thank you for your guidance and for the opportunity to work on this project.

Sincerely,
Mukul Aggarwal (2401030239)

Ananya Srivastava (2401030230)
Aastha Malik (2401030251)

Contents

1 Introduction

The *Car Dodge* game is a 2D arcade-style experience developed using C++ and the Raylib graphics library as part of the Software Development Fundamentals (SDF) curriculum. Inspired by classic arcade games, *Car Dodge* challenges players to control a car navigating a busy highway, dodging oncoming enemy vehicles, collecting coins to boost their score, and utilizing power-ups to gain temporary advantages. The game's core objective is to survive as long as possible, with difficulty increasing over time through faster enemy spawns and tighter navigation spaces. A persistent high-score leaderboard adds a competitive edge, encouraging players to refine their skills and aim for the top spot.

This project served as a practical platform to apply fundamental programming concepts, including object-oriented design, real-time event handling, and file input/output operations. Beyond technical implementation, the development process emphasized collaboration, iterative testing, and creative problem-solving. By integrating engaging gameplay mechanics with a polished user interface, *Car Dodge* aims to deliver both entertainment and educational value. This report details the tools, technologies, and methodologies employed, alongside a comprehensive analysis of the game's design, implementation, challenges, and potential for future enhancements.

2 Tools and Technologies Used

The development of *Car Dodge* leveraged a carefully selected stack of tools and technologies to ensure efficiency, portability, and ease of implementation. Below is a detailed overview:

- **Programming Language: C++**
  C++ was chosen for its performance, control over system resources, and robust support for object-oriented programming. Its versatility allowed us to implement complex game logic, manage memory efficiently, and interface seamlessly with the Raylib library. Features like classes, pointers, and STL containers were instrumental in structuring the codebase.

- **Graphics Library: Raylib**
  Raylib, a lightweight C/C++ library, provided an accessible framework for 2D graphics rendering, input handling, and audio integration. Its simplicity and comprehensive documentation enabled rapid prototyping and development, making it ideal for a semester-long project. Raylib's cross-platform compatibility ensured the game could run on various operating systems with minimal modifications.

- **Development Environment: Visual Studio Code with MinGW**
  Visual Studio Code (VSCode) served as the primary integrated development environment (IDE), offering a lightweight yet powerful interface for coding, debugging, and version control. The MinGW compiler facilitated C++ compilation on Windows, ensuring compatibility with Raylib's requirements. VSCode extensions for

C++ syntax highlighting and Git integration streamlined the development workflow.

- **Asset Design: Custom and Free Resources**
  The game's visual assets, including car sprites, coins, power-up icons, and the road background, were sourced from free online repositories and customized using graphic design tools like GIMP. This approach balanced quality and accessibility while adhering to licensing constraints. Audio effects, such as coin collection and collision sounds, were similarly sourced and integrated using Raylib's audio module.

- **Version Control: Git**
  Git was employed for collaborative development, enabling version tracking, conflict resolution, and seamless integration of contributions from all team members. A private GitHub repository hosted the project, with regular commits documenting progress and facilitating rollback when needed. Branching strategies ensured stable development and testing cycles.

3 Features

Car Dodge incorporates a rich set of features designed to enhance gameplay, challenge players, and demonstrate technical proficiency. These include:

- **Player-Controlled Car:** The player navigates a car using left and right arrow keys, with smooth movement constrained within the screen's boundaries to prevent edge-clipping.

- **Enemy Cars:** Oncoming vehicles spawn at the top of the screen and move downward at varying speeds, increasing in frequency and velocity as the game progresses.

- **Coin Collection System:** Coins appear randomly across the highway, awarding points upon collection and contributing to the player's score, displayed in real-time.

- **Power-Ups:** Three unique power-ups—Shield (temporary invincibility), Magnet (automatic coin attraction), and Nitro (speed boost)—spawn periodically, adding strategic depth and excitement. Each power-up has a timed duration and distinct visual cues.

- **Dynamic Difficulty:** The game adapts to the player's survival time by increasing enemy spawn rates and movement speeds, ensuring a progressively challenging experience.

- **Timer-Based Gameplay:** A session timer tracks survival duration, displayed alongside the score to provide immediate feedback on performance.

- **Persistent Leaderboard:** High scores are saved to a file, allowing players to compete across sessions. The leaderboard displays the top five scores with player initials.

These features collectively create an immersive and replayable arcade experience, balancing accessibility for beginners with challenges for seasoned players.

4 Game Design and Implementation

4.1 Game Mechanics

The core gameplay revolves around the player controlling a car constrained to horizontal movement along the bottom of the screen. Enemy cars spawn at random x-positions at the top and descend at speeds that scale with game time, requiring quick reflexes to avoid collisions. Coins and power-ups appear at randomized intervals, incentivizing exploration and risk-taking. The game ends upon collision with an enemy car (unless shielded), at which point the final score is saved if it ranks among the top scores.

To enhance engagement, we implemented a dynamic difficulty system that adjusts enemy spawn frequency and speed based on a time-based scaling factor. For example, after 30 seconds, the spawn interval decreases by 10%, and enemy velocity increases by 5%, creating a smooth difficulty curve. Power-ups like the Magnet attract coins within a defined radius, while Nitro temporarily doubles the player's movement speed, requiring careful navigation to avoid overshooting.

4.2 Physics and Collision Detection

Collision detection employs axis-aligned bounding box (AABB) checks, a computationally efficient method suitable for 2D games. Each game object—player car, enemy cars, coins, and power-ups—is represented by a rectangular hitbox. The Raylib function `CheckCollisionRecs` is used as follows:

```
if (CheckCollisionRecs(playerRect, enemyRect)) {
    if (!player.isShielded) gameOver = true;
}
```

For coins and power-ups, similar checks trigger score increments or ability activations. To optimize performance, collision checks are performed only for active objects, reducing unnecessary computations. We also fine-tuned hitbox sizes to ensure fairness, slightly shrinking enemy hitboxes to account for visual overlap and improve player experience.

4.3 Game Loop and Timing

The game operates on a standard real-time game loop, running at 60 frames per second (FPS) to ensure smooth visuals and responsive controls. The loop handles:

- **Input Detection:** Keyboard inputs (left/right arrow keys) update the player's position, with bounds checking to prevent off-screen movement.
- **Object Updates:** Enemy cars, coins, and power-ups move or spawn based on predefined timers and random intervals.
- **Collision Checks:** All active objects are checked for collisions in each frame, with outcomes like score updates or game-over states processed immediately.
- **Score and Timer Updates:** The score increments with coin collections, and the survival timer advances, influencing difficulty parameters.
- **Power-Up Management:** Active power-ups have timers that decrement each frame, deactivating effects upon expiration and resetting visuals.

Timing is managed using Raylib's `GetFrameTime` to ensure frame-rate independence, allowing consistent gameplay across different hardware. For example, object velocities are scaled by the frame delta time:

```
enemy.position.y += enemy.speed * GetFrameTime();
```

4.4 Assets and Graphics

The game's visual assets create an immersive highway environment:

- **Background:** A scrolling road image provides a sense of motion, tiled seamlessly using Raylib's texture rendering.
- **Sprites:** Player and enemy cars feature distinct designs, with the player's car in blue and enemies in varied colors to aid recognition. Coins and power-up icons (Shield, Magnet, Nitro) use bright, recognizable graphics.
- **UI Elements:** Score, timer, and power-up status are displayed using Raylib's `DrawText` in a clear, readable font.

Assets are loaded at initialization using `LoadTexture` and unloaded at program exit to prevent memory leaks. Audio effects, such as coin collection jingles and crash sounds, enhance feedback, loaded via `LoadSound` and played on specific events. Asset management was optimized by reusing textures for similar objects, reducing memory usage.

The codebase is modular, organized into separate files for maintainability and scalability:

Classes use encapsulation to manage state, with public methods for interactions and private members for data integrity. This structure facilitated debugging and allowed team members to work on separate components concurrently.

5 Challenges Faced

The development process presented several challenges, each requiring creative solutions:

- **Managing Multiple Objects:** Handling dozens of enemies, coins, and power-ups simultaneously risked performance degradation. We addressed this by using dynamic arrays (C++ `std::vector`) and removing off-screen objects promptly.

- **Power-Up Timing:** Ensuring smooth activation and expiration of power-ups was complex, especially with overlapping effects. We implemented a state machine for the player, tracking active power-ups and their remaining durations.

- **File I/O for Scores:** Persistent leaderboard storage required robust file handling. Initial issues with corrupted files were resolved by validating data before writing and using a simple text-based format.

- **Balancing Difficulty:** Early versions were either too easy or impossibly hard. We iterated on the difficulty curve, testing spawn rates and speed increments to achieve a satisfying progression.

- **Team Coordination:** Aligning contributions from multiple team members required clear communication. Regular Git commits and team meetings ensured consistency and resolved merge conflicts promptly.

Each challenge strengthened our problem-solving skills and deepened our understanding of software development workflows.

6 Conclusion

The *Car Dodge* project was a rewarding exploration of game development, blending technical rigor with creative design. By leveraging C++ and Raylib, we implemented a polished arcade game that demonstrates proficiency in graphics programming, object-oriented design, and real-time systems. The project taught us to manage memory efficiently, handle user inputs responsively, and maintain persistent data across sessions. Collaborative tools like Git and modular code design enhanced our teamwork and project management skills.

Reflecting on the experience, we recognize the value of iterative testing and user feedback in refining gameplay. Looking ahead, we envision expanding *Car Dodge* with additional levels, varied enemy types, enhanced visuals, and multiplayer modes. Integrating soundtracks and animated sprites could further elevate the experience. This project has laid a strong foundation for future endeavors in software and game development, equipping us with both technical expertise and a passion for innovation.

7 References

- Raylib Official Website: `https://www.raylib.com/`
- C++ Reference: `https://cplusplus.com/`
- Raylib Cheatsheet: `https://github.com/raysan5/raylib-cheatsheet`
- Game Programming Patterns, Robert Nystrom: `https://gameprogrammingpatterns.com/`
- OpenGameArt for Assets: `https://opengameart.org/`