Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report

on

"VELOCITY CLASH – AI BATTLE ARENA"

[Code No.: COMP 202]

(For partial fulfillment of Year I / Semester II in Computer Engineering)

Submitted by

Ankit Jha (70)

Submitted to

Er. Sagar Acharya

Department of Computer Science and Engineering

February 26 2026

This project work on

"VELOCITY CLASH – AI BATTLE ARENA"

is the bonafide work of

Ankit Jha (70)

who carried out the project work under my supervision.

Project Supervisor
Er. Sagar Acharya
Teaching Assistant
Department of Computer Science and Engineering

Date: **February 26 2026**

Acknowledgement

I would like to express my sincere gratitude to the **Department of Computer Science and Engineering** at **Kathmandu University** for providing me with the opportunity and resources to undertake this project. This platform has allowed me to bridge theoretical knowledge with practical application, resulting in meaningful learning and growth.

My deepest appreciation goes to my supervisor, **Er. Sagar Acharya**, whose guidance, encouragement, and constructive feedback were instrumental throughout the development of this project. His mentorship helped me navigate complex technical challenges and consistently elevated the quality of my work.

I also extend my heartfelt thanks to all faculty members and staff of the Department for fostering a positive and intellectually stimulating environment.

This project has provided valuable hands-on experience in game development, object-oriented programming, AI logic, real-time rendering, and software architecture — all of which have greatly enhanced my technical skill set and understanding of computer engineering fundamentals.

Ankit Jha (70)

Abstract

Velocity Clash – AI Battle Arena is a desktop-based 2D arcade game developed using C++ and the raylib graphics library. It presents a structured and enhanced version of the classic Pong concept, featuring AI-controlled opponent movement, dynamic speed escalation, multi-state game management, real-time collision detection, and a complete scoring system.

The system is built around four core classes — GameBall, Paddle, PlayerBar, and ComputerBar — following object-oriented programming principles. The AI paddle tracks the ball's vertical position and adjusts its movement using difficulty-based speed logic to ensure balanced gameplay. Ball–paddle collisions progressively increase ball speed during rallies, gradually raising the game's difficulty.

The game operates through four states: MENU, PLAYING, PAUSED, and GAMEOVER, ensuring structured transitions and smooth user interaction. It was developed using raylib 4.2.0 for 2D rendering, compiled with g++ under the C++14 standard, and built using a custom Makefile supporting debug and release modes.

Keywords: *Velocity Clash, AI Battle Arena, C++, raylib, Pong, Game Development, Collision Detection, Object-Oriented Programming, Game Loop, Artificial Intelligence*

Table of Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Expansion
AI	Artificial Intelligence
C++	C Plus Plus Programming Language
CPU	Central Processing Unit
EXE	Executable File
FPS	Frames Per Second
g++	GNU C++ Compiler
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
OOP	Object-Oriented Programming
OS	Operating System
RAM	Random Access Memory
2D	Two-Dimensional

Chapter 1
Introduction

1.1 Background

Video games have long served as an important domain for applying computer science concepts, ranging from real-time rendering and physics simulation to artificial intelligence and structured software architecture. Among classic game designs, the Pong game — originally introduced in the 1970s — remains one of the most iconic examples for demonstrating game loop mechanics, collision detection, player input handling, and opponent AI behaviour in a compact and well-understood format.

Modern implementations of Pong-style games allow developers to explore contemporary software engineering principles including object-oriented design, modular architecture, event-driven programming, and hardware-accelerated 2D rendering. Libraries such as raylib provide a lightweight, cross-platform foundation for building such games in C++ without the overhead of large game engines, making them well-suited for academic projects and rapid prototyping.

Despite the simplicity of the Pong concept, implementing a complete and polished version requires careful attention to system design. Issues such as ball-speed escalation, AI difficulty tuning, smooth frame rendering, and clean state management all present real engineering challenges. This project — Velocity Clash: AI Battle Arena — addresses these challenges through a structured, class-based architecture and a clear game state machine.

1.2 Objectives

The primary objectives of this project are:

- Develop a fully functional 2D Pong-style arcade game in C++ using the raylib library, applying real-time game development principles with a modular, object-oriented design (Ball, Player Paddle, AI Paddle) to ensure code clarity, reusability, and maintainability.
- Implement an AI-controlled paddle with configurable difficulty, dynamic collision detection, and progressive ball speed escalation to create competitive and engaging gameplay.
- Design a structured game state system (MENU, PLAYING, PAUSED, GAMEOVER) for smooth transitions and complete user experience, supported by a professional Makefile build system with debug/release modes and a portable Windows executable.

1.3 Motivation and Significance

The motivation for developing Velocity Clash arose from the desire to apply core computer science fundamentals — OOP, real-time systems, AI logic, and graphics programming — in the context of an interactive and visually engaging application. Game development offers an ideal environment for learning because it requires the integration of multiple disciplines simultaneously: software architecture, physics, input handling, rendering, and user experience.

From an academic perspective, this project is significant in several ways. First, it provides hands-on experience with the raylib library, a powerful yet accessible tool for 2D game development in C++. Second, the implementation of AI paddle logic — even at the level of simple ball-tracking — introduces the fundamental concept of autonomous agent behaviour, which underlies more complex AI systems. Third, the structured Makefile build system introduces professional-grade software development practices, including compilation flags, build modes, and dependency management.

The project also addresses a gap in many beginner-level game implementations: the lack of structured game state management. By implementing a clear state machine, Velocity Clash demonstrates how game applications should handle transitions between distinct operational modes — a pattern applicable to far more complex game and software systems. In summary, this project serves as a practical demonstration of how academic knowledge in C++ programming, object-oriented design, and algorithm implementation can be applied to create a real, interactive software product.

Chapter 2
Literature Review

2.1 Introduction

The development of Velocity Clash – AI Battle Arena draws upon a body of work spanning classic arcade game implementations, modern game development frameworks, AI movement logic in competitive games, and software engineering best practices. Reviewing these sources helps identify the strengths and weaknesses of prior approaches and provides the foundation for the design decisions made in this project.

2.2 Previous Works

S. S. Ghosh (2020) developed a C++ console-based cinema ticket booking system demonstrating structured C++ application design and file-based data management. While not directly related to game development, this work highlights the importance of modular code structure and clear separation of concerns in C++ applications — principles adopted in the architecture of Velocity Clash.

The raylib library (Raylib Documentation, 2024) provides a simple and easy-to-use C library for game programming. It offers hardware-accelerated 2D and 3D rendering, input handling, audio management, and window creation without requiring external dependencies. Its minimal API surface makes it particularly suited for academic projects, while its cross-platform support ensures portability across Windows, Linux, and macOS.

Robert Nystrom's Game Programming Patterns (2014) provides an extensive analysis of software design patterns specifically tailored to game development. Patterns such as the Game Loop, Component, and State Machine are directly relevant to the architecture of Velocity Clash.

Alkaison (2025) implemented a Java-based movie ticket booking management system using SQLite for persistence. Although in a different domain, this work demonstrates effective object-oriented design and separation of layers — a principle also applied in this project's class hierarchy.

Shabrawy (2025) developed an airline reservation system in C++ using Qt and MySQL, demonstrating complex GUI development and database integration in C++. Velocity Clash takes a lighter approach suited to real-time game development rather than data-driven application design.

2.3 Comparison of Existing Systems

Table 2.1 summarises the relevant previous works, highlighting their platforms, features, and limitations in the context of this project.

Table 2.1: Comparison of Previous Works

Source	Platform	Features	Limitations
S. S. Ghosh (2020)	C++ Console	Ticket booking, seat availability, file storage	No GUI; basic console; internet needed for updates
raylib (2024)	C Library	Hardware-accelerated 2D/3D rendering, cross-platform	Requires C/C++ knowledge; no built-in UI widgets
Robert Nystrom (2014)	Textbook	Game loop patterns, OOP design, software architecture	Conceptual only; no runnable code or graphics
Alkaison (2025)	Java / SQLite	GUI-based ticket booking, booking history	Different language; no game logic or rendering
Shabrawy (2025)	C++ / Qt / MySQL	Airline reservation, GUI, database backend	Different domain; no AI logic or game loop

2.4 Current Work

Velocity Clash – AI Battle Arena is a C++ desktop game built with the raylib library that addresses the key limitations identified in the works above. Specifically, the proposed system:

- Provides a real-time, hardware-accelerated graphical interface using raylib, unlike console-based or non-game applications.
- Implements structured AI logic for the opponent paddle without requiring external AI frameworks.
- Uses a clean, object-oriented architecture with distinct, reusable classes for each game entity.
- Operates entirely offline with no database dependency, keeping the system lightweight and portable.
- Supports both debug and release builds via a custom Makefile, reflecting professional software development practices.

2.5 Overview of Existing Systems

2D game development has a long history dating back to early arcade systems. Modern implementations of classic games like Pong are widely used in computer science education to teach core programming concepts. Libraries such as SDL2, SFML, and raylib have made it accessible to implement such games in C/C++ without requiring complex engine infrastructure. Among these, raylib stands out for its simplicity, minimal dependencies, and active documentation.

AI in classic games typically falls into two categories: reactive AI, where the opponent responds directly to game state, and predictive AI, which anticipates future positions. For Pong-style games, reactive tracking AI is standard and provides effective competition without excessive complexity.

2.6 Technologies Used in Previous Works

Most previous implementations of Pong-style games in academic settings used either C++ with SDL or SFML for graphics, or higher-level languages such as Python with Pygame. Console-based C++ implementations exist but lack the interactive graphical experience expected of a modern game. The use of raylib in Velocity Clash represents a practical middle ground — providing full 2D rendering capabilities while keeping the API simple and the build process straightforward.

2.7 Drawbacks and Gaps in Existing Systems

Existing simple Pong implementations reviewed during research suffer from several common drawbacks. Many console-based versions lack graphical feedback, making them less engaging and harder to use. Graphical implementations often use deprecated or heavyweight libraries. AI implementations are frequently either too simplistic or unrealistic, leaving room for a balanced, difficulty-configurable approach. Additionally, few academic implementations include proper game state management — most programs use simple boolean flags rather than a structured state machine.

2.8 Summary

The review of existing literature confirms that while Pong-style games are a common educational exercise, implementations combining structured OOP design, configurable AI difficulty, dynamic speed escalation, proper state machine architecture, and a professional build system are less common in academic projects. Velocity Clash addresses these gaps and builds upon the foundations established by the reviewed works to produce a complete, polished, and educationally valuable implementation.

Chapter 3
Methodology

3.1 System Overview

Velocity Clash – AI Battle Arena is a desktop-based 2D real-time arcade game built using C++ and the raylib graphics library. The game is designed around a structured class hierarchy and a finite state machine governing gameplay transitions. The system runs at a fixed target of 60 frames per second, ensuring smooth and consistent rendering across supported hardware configurations.

The game involves two paddles and a ball on a 1200×700 pixel arena. The player controls the right paddle using the W and S keyboard keys. The AI controls the left paddle, tracking the ball's vertical position and adjusting movement speed based on the configured difficulty level. A match ends when either side reaches five points, at which point the game transitions to the GAMEOVER state.

The source code is organised into a single file (main.cpp) comprising global configuration constants, colour definitions, the game state enum, global score variables, and four classes: GameBall, Paddle, PlayerBar, and ComputerBar. The main() function initialises raylib, creates game objects, and runs the primary game loop.

3.2 Gameplay Overview

Upon launch, the game displays a MENU screen with the title and an instruction to press ENTER. Once the game starts, the ball is launched from the centre in a random direction. The player moves the right paddle up and down to deflect the ball while the AI paddle on the left automatically tracks and deflects it. Each time the ball passes a paddle and exits the screen boundary, the opposing side earns a point. The game pauses and resumes with the P key, and the first to score five points wins.

A key gameplay mechanic is progressive speed escalation: each time the ball collides with either paddle, its horizontal speed is multiplied by 1.1 and its vertical speed by 1.05. This ensures that rallies become increasingly fast and challenging, preventing indefinite play and increasing excitement as a match progresses.

3.3 Game State Architecture

The game is governed by a finite state machine with four states defined in the GameState enum: MENU, PLAYING, PAUSED, and GAMEOVER. Transitions are triggered by specific keyboard inputs as follows:

- MENU → PLAYING: triggered by pressing ENTER
- PLAYING → PAUSED: triggered by pressing P
- PAUSED → PLAYING: triggered by pressing P again

- PLAYING → GAMEOVER: triggered automatically when either score reaches MAX_SCORE (5)
- GAMEOVER → MENU: triggered by pressing R, which resets scores and ball position

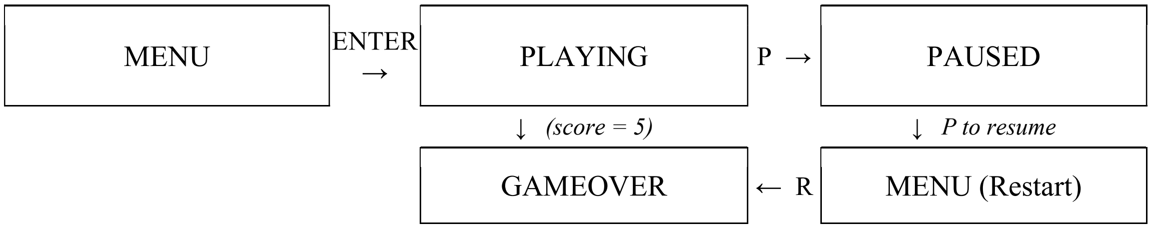

Figure 3.1: Game State Architecture Diagram

Each state determines what rendering and update logic is executed within the main game loop, ensuring clean separation of behaviour across different game phases.

3.4 System Design

3.4.1 Architectural Design

The system uses a class-based architecture where each game entity is encapsulated in its own class. All classes operate within a single main game loop hosted in the main() function. The Paddle base class provides shared geometry, drawing logic, and boundary enforcement. PlayerBar and ComputerBar inherit from Paddle and add input handling and AI logic respectively. GameBall is an independent class managing position, velocity, rendering, collision response, and score updates.

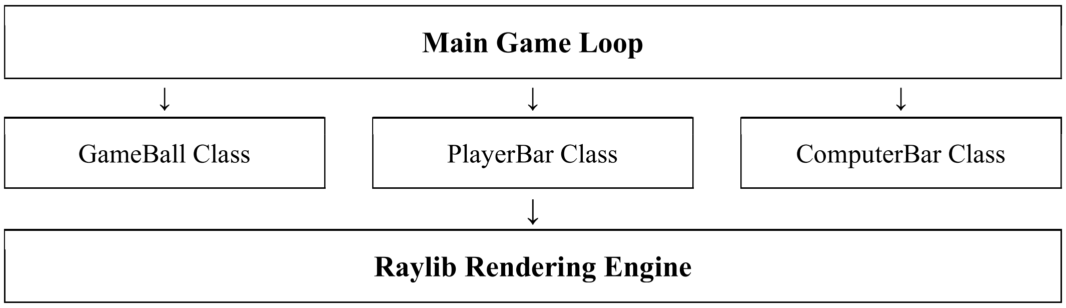

Figure 3.2: System Architecture Diagram

3.4.2 Module Design

The major modules are summarised in Table 3.1 below.

Table 3.1: Major Modules of Velocity Clash

Module / Class	Functionality
GameBall Class	Manages ball position, speed, draw, update, wall bouncing, and score-triggered reset logic
PlayerBar Class	Inherits from Paddle; handles W/S keyboard input and movement boundary enforcement
ComputerBar Class	Inherits from Paddle; implements AI tracking logic with difficulty-based speed scaling
Main Game Loop	Manages game states (MENU, PLAYING, PAUSED, GAMEOVER), rendering pipeline, and collision detection

3.5 GUI Overview

The game provides three distinct screens corresponding to its game states, each rendered using raylib's 2D drawing functions within the main game loop.

3.5.1 Menu Screen

The menu screen is displayed on launch, showing the game title and an instruction to press ENTER to begin. It serves as the entry point into the game.

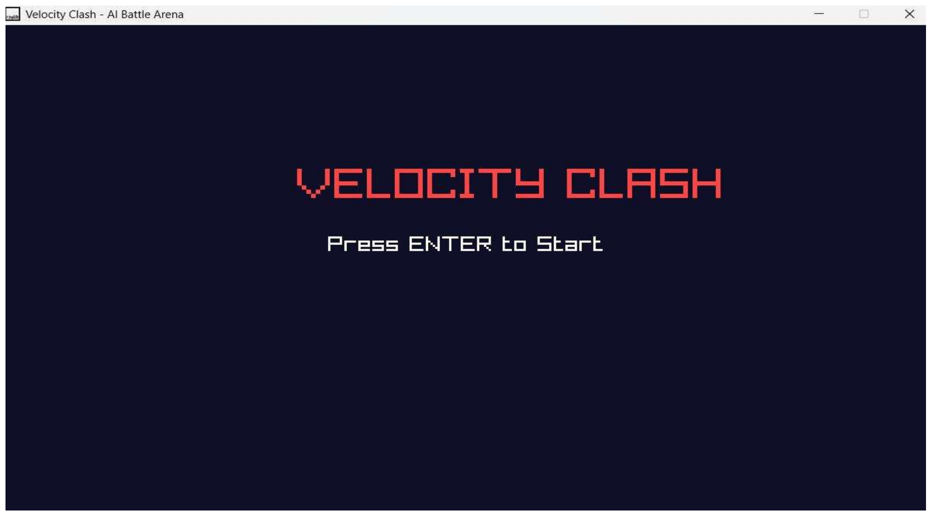

Figure 3.3: Menu Screen of Velocity Clash

3.5.2 Gameplay Screen

The playing screen renders both paddles, the ball, the centre dividing line, and the live score for each side. The player's paddle appears on the right and the AI paddle on the left.

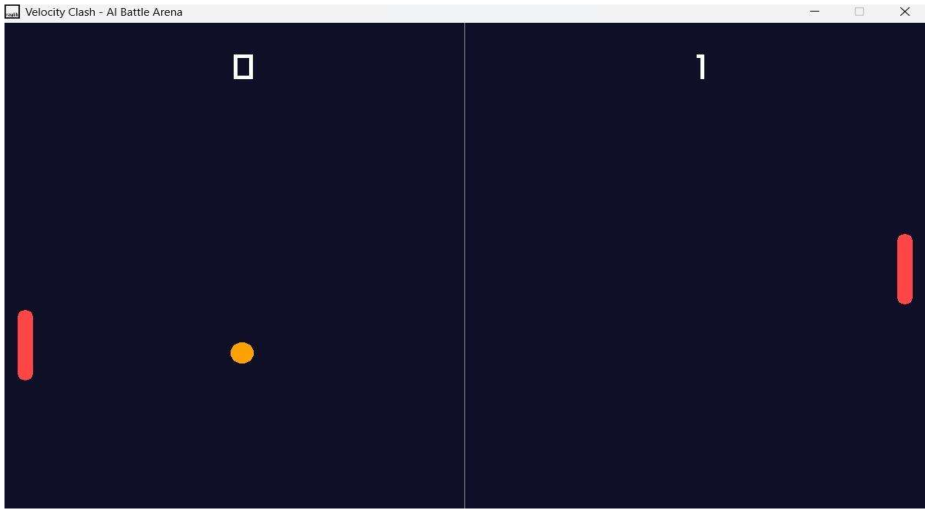

Figure 3.4: Gameplay Screen (PLAYING state)

3.5.3 Game Over Screen

When either score reaches five, the game transitions to the GAMEOVER state and displays the result — either 'YOU WIN!' or 'AI WINS!' — with a prompt to press R to restart.

Figure 3.5: Game Over / Win Screen

3.6 Implementation Details

3.6.1 GameBall Class

The GameBall class stores the ball's position (x, y), speed components (speedX, speedY), and radius. The Draw() method renders the ball as a filled orange circle using DrawCircle(). The Update() method handles vertical wall bouncing by inverting speedY when the ball touches the top or bottom boundary, score updates when the ball exits the screen edge, and a Reset() call that recentres the ball and randomises the launch direction using GetRandomValue().

3.6.2 Paddle and Derived Classes

The Paddle base class defines shared attributes — position, dimensions, and speed — along with a Draw() method using DrawRectangleRounded() for visually smooth paddles, and a LimitMovement() method that constrains the paddle within screen boundaries. PlayerBar inherits Paddle and adds keyboard-based movement via IsKeyDown(KEY_W) and IsKeyDown(KEY_S). ComputerBar inherits Paddle and implements AI tracking: it computes the centre of the paddle and moves toward the ball's Y position, with speed scaled by 0.6 on Easy difficulty and full speed on Hard.

3.6.3 Collision Detection

Collision between the ball and each paddle is detected using raylib's CheckCollisionCircleRec() function, which tests intersection between a circle (ball) and an axis-aligned rectangle (paddle). Upon collision, the ball's horizontal speed is multiplied by −1.1 (direction reversal with speed increase) and its vertical speed by 1.05. This progressive escalation ensures that each rally incrementally raises the game's pace.

3.6.4 Build System

The project uses a custom Makefile supporting four build targets: build (default), debug (with -g -O0 flags), release (with -O2 -s flags), and clean. The Makefile links against the raylib, opengl32, gdi32, and winmm system libraries required for Windows desktop deployment. The project compiles as a single translation unit from main.cpp.

3.7 Tools and Technologies Used

- C++ (C++14 Standard): Core programming language
- raylib 4.2.0: 2D graphics and window management library
- g++ (GNU C++ Compiler): Compilation and linking
- Custom Makefile: Build automation with debug and release modes
- Visual Studio Code: Development environment with C/C++ extension
- Windows Desktop: Target deployment platform

3.8 Summary

This chapter described the complete methodology adopted in developing Velocity Clash – AI Battle Arena. The system architecture, game state machine, class design, collision and AI logic, speed escalation mechanics, and build system were all discussed in detail. The modular OOP design and structured game loop provide a clean, maintainable foundation suitable for future enhancements.

Chapter 4
Discussion and Achievements

4.1 Project Overview

Velocity Clash – AI Battle Arena was developed with the goal of creating an engaging, well-structured 2D arcade game that applies core computer science concepts in a practical and interactive context. The project successfully delivers a complete gaming experience — from the initial menu screen through gameplay and into post-match states — built entirely in C++ using the raylib library. The game is compiled to a standalone Windows executable requiring no installation or external runtime dependencies beyond the bundled DLL files.

4.2 Object-Oriented Architecture

One of the central achievements of this project is the implementation of a clean, extensible OOP architecture. The Paddle base class provides shared geometry and boundary logic, while PlayerBar and ComputerBar extend it with domain-specific behaviour — keyboard input and AI tracking respectively. GameBall encapsulates all ball-related state and behaviour. This separation of concerns ensures that each class has a single, well-defined responsibility, making the codebase easy to read, modify, and extend.

4.3 AI Paddle Behaviour

The AI-controlled paddle successfully demonstrates a fundamental approach to game AI: reactive tracking. The ComputerBar class compares the vertical centre of the paddle to the ball's current Y position and adjusts movement direction accordingly. The difficulty parameter allows the AI's effective speed to be scaled — operating at 60% speed on Easy mode and full speed on Hard mode. This creates a tangible and tunable challenge level, making the game accessible to beginners while remaining competitive for more experienced players. The implementation deliberately avoids perfect tracking to maintain fairness and game longevity.

4.4 Dynamic Collision and Speed Escalation

The collision detection system using CheckCollisionCircleRec() proved accurate and responsive throughout testing. The progressive speed escalation — a 10% horizontal increase and a 5% vertical increase per paddle hit — creates a natural game arc where early rallies are manageable and extended rallies become increasingly intense. This mechanic prevents matches from lasting indefinitely and rewards players who keep the ball in play. During testing, it was observed that a well-played rally could bring the ball to approximately three times its initial speed within ten to twelve collisions.

4.5 Game State Management

The finite state machine governing MENU, PLAYING, PAUSED, and GAMEOVER states worked reliably in all testing scenarios. Transitions were immediate and visually clear, with appropriate screen messages displayed in each state. The PAUSED state correctly froze all game logic while preserving score and ball position, allowing players to resume seamlessly. The GAMEOVER state correctly identified the winner and provided a restart mechanism that fully reset the game state without requiring application restart.

4.6 Rendering Performance

The game maintained stable 60 FPS performance throughout all testing sessions on the development machine. raylib's hardware-accelerated rendering ensured that all draw calls — background, centre line, ball, paddles, and score text — were completed well within the frame budget. No frame rate drops or rendering artefacts were observed during gameplay.

4.7 Build System

The custom Makefile successfully automated both debug and release compilation workflows. The debug build with -g -O0 flags facilitated development-time debugging, while the release build with -O2 -s produced a smaller, optimised executable suitable for distribution. The make run, make clean, and make rebuild targets further streamlined the development workflow.

4.8 Achievements

Through the development of Velocity Clash – AI Battle Arena, several technical and functional milestones were achieved:

- Successfully implemented a fully functional, graphical 2D arcade game from scratch using C++ and raylib.
- Designed and implemented a clean OOP class hierarchy (GameBall, Paddle, PlayerBar, ComputerBar) with clear separation of responsibilities.
- Developed a reactive AI opponent with tunable difficulty scaling, providing competitive and engaging gameplay.
- Implemented dynamic collision detection with progressive speed escalation, ensuring varied and challenging matches.
- Built a robust four-state game state machine (MENU, PLAYING, PAUSED, GAMEOVER) with clean transitions.
- Delivered a standalone Windows executable with a professional Makefile build system supporting debug and release modes.
- Gained practical experience in real-time graphics programming, event-driven input handling, game loop architecture, and applied OOP design.

Chapter 5
System Requirements

5.1 Software Requirements

This section presents the required software components for developing and running Velocity Clash – AI Battle Arena.

5.1.1 Minimum Software Requirements

Table 5.1: Software Requirements

Component	Specification
Operating System	Windows 7 / 8 / 10 / 11 (64-bit recommended)
Programming Language	C++14 or above
Graphics Library	raylib 4.2.0 (static library)
Compiler	g++ (MinGW-w64 or equivalent)
Build System	GNU Make (via Makefile)
IDE / Editor	Visual Studio Code with C/C++ extension
Additional DLLs	libgcc_s_dw2-1.dll, libstdc++-6.dll (bundled)

5.2 Hardware Requirements

5.2.1 Minimum Hardware Requirements

Table 5.2: Hardware Requirements

Component	Specification
Processor	Dual-core CPU, 1.5 GHz or higher
RAM	4 GB minimum (8 GB recommended)
Storage	50 MB free disk space
Display	1200×700 resolution or higher
GPU	Any GPU supporting OpenGL 3.3 or later
Input Devices	Keyboard (W, S, P, R, ENTER keys required)

Table 5.2: Hardware Requirements

5.3 Functional Requirements

The following functional requirements were defined and achieved in the final implementation:

- The game shall render a 1200×700 pixel window at a target of 60 frames per second.
- The player shall control the right paddle using the W (up) and S (down) keyboard keys.
- An AI opponent shall control the left paddle, tracking the ball's vertical position with configurable difficulty.
- The ball shall bounce off the top and bottom walls by reversing its vertical velocity.
- Each time the ball collides with a paddle, its speed shall increase by a fixed multiplier.
- A score counter shall be maintained for both the player and AI, displayed on screen during gameplay.
- The game shall transition to GAMEOVER when either score reaches MAX_SCORE (5 points).
- The game shall support PAUSE and RESUME functionality via the P key.
- Upon GAMEOVER, the winner shall be announced and the player offered the option to restart via the R key.

5.4 Non-Functional Requirements

5.4.1 Performance

The game must maintain stable 60 FPS rendering on all target hardware configurations. The game loop must complete each iteration — including updates, collision detection, and rendering — within the 16.67 ms frame budget to avoid visual stuttering.

5.4.2 Usability and Accessibility

The game interface must be immediately understandable to a new player without requiring external documentation. On-screen text must clearly communicate available actions (ENTER to start, P to pause, R to restart) at all relevant state transitions. Paddle and ball visuals must be clearly distinguishable against the dark background.

5.4.3 Portability and Maintainability

The source code must be compilable on any Windows system with a compatible g++ compiler and raylib installation, using only the provided Makefile. The single-file source structure (main.cpp) simplifies distribution and compilation. The modular class design ensures that individual components can be modified or extended without affecting unrelated parts of the system.

Chapter 6
Conclusion and Recommendation

This project successfully developed a desktop-based 2D arcade game, Velocity Clash – AI Battle Arena, using C++ and the raylib graphics library. The core objectives outlined in Section 1.2 have been fully achieved: the game provides a complete interactive experience with AI-controlled opposition, dynamic collision physics, progressive speed escalation, structured state management, and a professional build system.

The object-oriented architecture — comprising GameBall, Paddle, PlayerBar, and ComputerBar classes — proved effective in organising game logic into manageable, reusable components. The AI paddle's reactive tracking strategy, combined with difficulty-scaled speed, delivers engaging and competitive gameplay without requiring complex AI frameworks. The finite state machine cleanly manages transitions between MENU, PLAYING, PAUSED, and GAMEOVER states, ensuring a smooth and intuitive player experience.

From an educational perspective, this project provided valuable practical experience in real-time game loop architecture, hardware-accelerated 2D rendering with raylib, applied OOP design in C++, collision detection algorithms, AI movement logic, and build system automation. These skills are broadly applicable across game development, systems programming, and software engineering disciplines.

6.1 Limitations

While the system meets all primary requirements, the following limitations have been identified:

- The game currently supports a single player versus AI mode only; a player-versus-player mode with two keyboard controllers is not implemented.
- Audio feedback — such as sound effects for paddle collisions, scoring events, and game over — has not been integrated in this version.
- The difficulty level is hardcoded at compile time; there is no runtime difficulty selection menu allowing the player to choose Easy or Hard before starting.
- The AI tracking logic is purely reactive and does not employ any predictive or anticipatory behaviour, which could make it exploitable by experienced players using extreme angles.
- The game window size (1200×700) is fixed and does not support dynamic resizing or full-screen mode.
- No persistent score tracking or high-score leaderboard is implemented; all progress is lost when the application closes.

6.2 Future Enhancements

The project provides a strong and extensible foundation for future development. Possible enhancements include:

- Implementing audio support using raylib's built-in audio module to add sound effects for collisions, scoring, and game events, significantly improving game feel.
- Adding a runtime difficulty selection menu allowing the player to choose between Easy, Medium, and Hard AI difficulty levels before each match.
- Developing a two-player local multiplayer mode, assigning the left paddle to one player and the right to another using independent key bindings.
- Implementing a predictive AI algorithm that estimates the ball's future position based on its current trajectory, providing a more challenging opponent.
- Adding particle effects and screen-shake feedback on collisions using raylib's rendering features to enhance the visual experience.
- Implementing a persistent high-score system using local file storage to track the longest rallies and match results.
- Adding full-screen mode support and dynamic window scaling to accommodate different display resolutions.
- Expanding the game with power-up items, variable paddle sizes, and multiple ball mechanics to create a more feature-rich experience.

References

Alkaison. (2025). Movie ticket booking management system (Java/SQLite). Retrieved from https://github.com/Alkaison/Movie-Ticket-Booking-Management-System

Ghosh, S. S. (2020). Kashipara cinema ticket booking system in C++. Retrieved from https://www.kashipara.com/project/c-c-/3862/cinema-ticket-booking-system

Nystrom, R. (2014). Game programming patterns. Retrieved from https://gameprogrammingpatterns.com

Raylib Documentation. (2024). raylib – A simple and easy-to-use library to enjoy videogames programming. Retrieved from https://www.raylib.com

Shabrawy, E. E. (2025). Airline reservation system in C++ using Qt and MySQL. Retrieved from https://github.com/EyadShabrawy/airline-reservation-system

Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley.

Appendix

.1 Additional Figures

The diagram below illustrates the sequential execution flow of the Velocity Clash game loop.

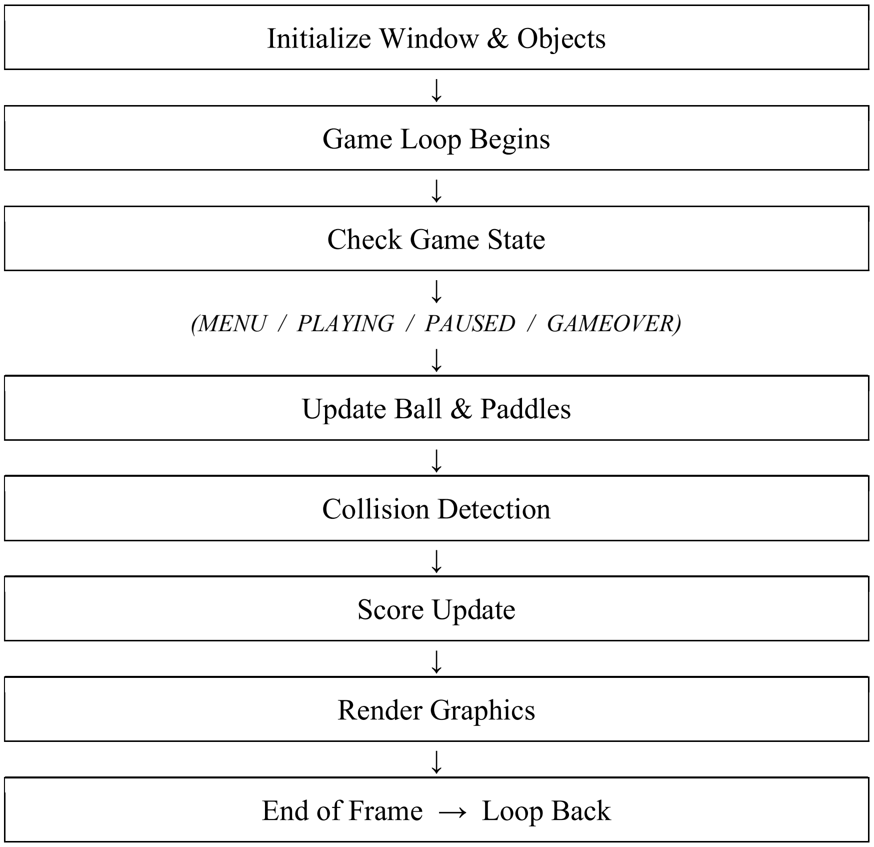

Figure A1: Game Flow Diagram

Hierarchical relationship between the main game loop, core classes, and the raylib rendering engine.

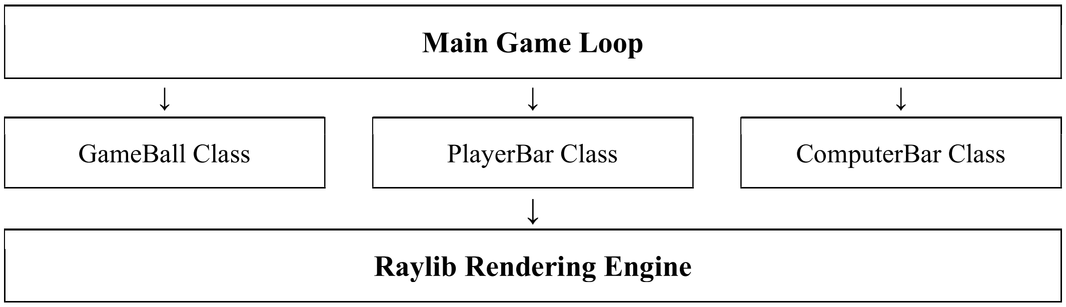

Figure A2: System Architecture Diagram