iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** XI **Month of publication:** November 2025

DOI: https://doi.org/10.22214/ijraset.2025.75174

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue XI Nov 2025- Available at www.ijraset.com

First Person Shooter Game

Kevin Sinclair Crasta[1], Monesh S[2], Musharaff Sheriff M C[3], Mrs. K. Yazhini[4]
[1, 2, 3]*Bachelor of Engineering in Computer Science and Engineering, Adhiyamaan College Of Engineering (An Autonomous Institution), ANNA University, Chennai*
[4]*M.E., Supervisor Assistant Professor, Department of CSE, Adhiyamaan College of Engineering, ANNA University, Chennai*

Abstract: This project is a fast-paced First-Person Shooter game developed using Unity and C#. The game challenges players to survive as long as possible by fighting off endless waves of enemies that become stronger over time. Players use a variety of weapons to eliminate enemies while avoiding turret attacks placed across the platform. The enemies follow a loop-based spawning system, creating continuous and increasingly difficult gameplay. The game's 3D models and animations are created using Blender, and textures are enhanced using additional tools to improve visual quality and realism. Key features of the game include custom shooting mechanics, intelligent enemy AI, a dynamic spawn system, and survival-based progression. This project showcases skills in game design, programming, 3D asset creation, and real-time performance optimization, offering an intense and immersive gaming experience on both mobile and Personal Computer platforms.

I. INTRODUCTION

A. Overview

The First-Person Shooter genre has consistently dominated the gaming landscape, captivating millions with its adrenaline-pumping action and immersive virtual worlds. This industry growth, however, has inadvertently created a barrier to entry the escalating demand for high-end hardware and stable, high-speed internet connectivity. These requirements often exclude a significant portion of the gaming community utilizing low or mid-range computing systems or those with unreliable network access.

At its heart, Endless Assault implements a compelling wave-based survival system. The fundamental objective is simple: survive for as long as possible against continuous, escalating waves of enemy combatants. This core loop drives player engagement by relentlessly increasing the difficulty, demanding a potent blend of strategic planning, resource management, quick reflexes, and skilful use of available weaponry. The gameplay effectively balances the need for immediate tactical decisions with long-term survival strategy.

The game's development leveraged a robust and industry-standard technical stack to ensure both stability and high performance:

1) Game Engine: Unity 6 was chosen as the primary development environment, providing a powerful and flexible platform for building interactive 3D experiences.
2) 3D Assets: Blender was utilized for creating and refining the 3D models and environments, guaranteeing optimized and visually consistent game assets.
3) Scripting and Logic: All core functionalities, including player movement controls, complex Enemy Artificial Intelligence, comprehensive scoring logic, and the user interface systems, were meticulously scripted using C#.

A key differentiator for Endless Assault is the sophisticated AI-driven enemy behavior. This system is powered by a Finite State Machine, which allows enemies to exhibit dynamic and realistic behavior cycles: Detect, Chase, and Attack. This dynamic AI significantly enhances the game's challenge and replay ability. Furthermore, enemy navigation across the map is seamlessly managed by Unity's powerful NavMesh system, ensuring intelligent and efficient pathfinding.

To uphold the commitment to performance on lower-spec hardware, the project incorporated several critical optimization techniques:

- Object Pooling: Minimizing runtime instantiation and destruction of frequent game objects like bullets and enemies to reduce memory overhead and stuttering.
- Light Baking: Pre-calculating complex static lighting to drastically reduce the real-time processing load on the GPU.
- Static Batching: Grouping multiple static objects into a single draw call to minimize CPU cycles required for rendering.

These optimizations are crucial in delivering a smooth and fluid 60 frames per second experience, fulfilling the project's goal of true accessibility.

B. Objective

The overarching goal of the Endless Assault project is to successfully design, develop, and deploy a fully functional, high-performance, and immersive offline First-Person Shooter survival game. This project aims to validate a technical solution that effectively overcomes the hardware accessibility limitations prevalent in modern AAA gaming.

The specific, measurable objectives guiding the development of Endless Assault are detailed below:

1) Accessibility and Performance

- To Define Performance Requirements: Thoroughly analyze the target system specifications low-to-mid range PCs and define clear technical requirements for rendering, processing speed, and memory usage. This establishes the baseline for all subsequent optimization efforts.
- To Ensure Optimized Gameplay: Successfully integrate and validate optimization techniques such as object pooling, static batching, and light baking to maintain a minimum stable frame rate of 60 FPS even on the defined low-end hardware, thus achieving true accessibility.

2) Game Design and Functionality

- To Implement Core Survival Mechanics: Design and develop the central wave-based survival system where difficulty dynamically scales with each successful wave, effectively challenging the player's strategy and reflexes.
- To Develop Advanced Enemy AI: Implement a robust Artificial Intelligence system utilizing a Finite State Machine. This system must enable enemies to Accurately detect, intelligently navigate via NavMesh, and dynamically engage the player, ensuring realistic and engaging combat scenarios.
- To Integrate Immersive Systems: Implement an intuitive Heads-Up Display, a realistic damage/health system, and effective in-game audio to provide a seamless and immersive user experience without unnecessary technical complexity.

3) Technical Integrity and Documentation

- To Ensure Code Reliability: Write clean, modular, and well-commented C# scripts for all game logic player controls, scoring, AI, ensuring the system is reliable and scalable for future content additions or feature expansions.
- To Validate and Test System Stability: Conduct rigorous validation, verification, and performance testing across multiple hardware configurations to confirm the system's security against common exploits and its stability under continuous load during extended gameplay sessions.
- To Systematically Document Development: Create comprehensive documentation detailing the project's planning, technical design, implementation, and testing procedures. This documentation will serve as a critical reference for future maintenance, upgrades, and knowledge transfer.
- To Provide Practical Demonstration: Deliver a compelling final demonstration of the game that showcases its smooth performance on low-end hardware, the dynamic enemy AI, and the engaging core survival loop, highlighting its advantages over typical hardware-intensive FPS titles.

C. Scope

The scope of the Endless Assault mini-project rigorously defines the boundaries of the developed system, focusing on delivering a technically sound, accessible, and highly engaging single-player, wave-based First-Person Shooter survival game.

The project scope is explicitly limited to the following key functional and non- functional requirements:

1) Functional Scope

- Wave-Based Survival System: The central element is the implementation of a single-player survival mode. This includes the generation of continuous enemy waves, where both the quantity and difficulty of enemies escalate over time, defining the core challenge.
- Realistic Combat Mechanics: The scope covers the development of realistic player controls, hit detection systems, and a small, functional arsenal of firearms with distinct properties.
- Finite State Machine AI: The project includes developing dynamic Non- Player Character Artificial Intelligence driven by a Finite State Machine. The AI will be capable of state transitions, including idle, detection, chasing, and attacking the player, ensuring varied and challenging enemy behaviour.
- Scoring and Persistence: The system will accurately track the player’s score, wave progression, and survival time, with basic functionality to display the highest score achieved to encourage replay ability.

2) *Non-Functional Scope*

- Optimization for Low-End Hardware: A critical boundary of this project is its accessibility. The system must be developed and optimized using techniques like object pooling, static batching, and effective asset management to ensure stable performance targeting 60 FPS on specified low- to-mid-range PC hardware, thereby functioning fully offline.
- Immersive Graphics and Audio: The scope includes integrating high- quality, yet optimized, 3D assets created via Blender and Unity. This involves basic, high-performance lighting and a set of custom sound effects to create a visually and audibly immersive experience without excessive resource drain.
- Expandable System Architecture: The code and level design architecture will be developed with modularity in mind, allowing for easy future expansion. This ensures that new levels, weapons, enemy types, or features can be integrated later without needing a complete overhaul of the core game logic.
- User Interface and Feedback: The project scope includes a clear and functional Heads-Up Display providing essential real-time feedback health, ammunition, wave count, score that is intuitive and non-intrusive.

II. LITERATURE SURVEY

1) Hernandez, A., Lee, S., & Thompson, E. [1] introduced "AI-Powered Player Behaviour Prediction in FPS Games." This paper investigates the use of machine learning algorithms to predict player actions and strategies in real time. It emphasizes how predictive models can enhance NPC responsiveness and adapt game difficulty dynamically. The study details the use of reinforcement learning for training NPCs to respond intelligently to diverse player tactics. Additionally, it discusses analytics for tracking player engagement and optimizing game flow. Furthermore, it explores the potential for integrating these techniques with multiplayer matchmaking systems to provide balanced and immersive experiences.
2) Johnson, L., Chen, H., & Rodriguez, M. [2] introduced "Pathfinding Optimization for Non-Player Characters in FPS Games." This paper explores the use of A* and Dijkstra algorithms for NPC navigation, emphasizing how these algorithms improve movement efficiency and reduce computational overhead in complex game environments. It details the integration of dynamic obstacle avoidance and decision trees to enhance NPC combat behaviour. Additionally, it discusses multi-agent coordination to create realistic group tactics, increasing game immersion. The study also examines adaptive difficulty scaling to maintain challenge based on player performance. Furthermore, it explores the potential of integrating machine learning models to predict player strategies, improving NPC responsiveness.
3) Kumar, S., Williams, D., & Lee, T. [3] introduced "Procedural Level Generation in First-Person Shooter Games." This paper focuses on generating dynamic and varied game maps using procedural generation techniques. It emphasizes how algorithms like Perlin noise and cellular automata can create realistic terrains and layouts. The study details the benefits of procedural content for replay ability and Player engagement. Additionally, it examines AI-driven placement of obstacles and rewards to maintain balanced gameplay. Moreover, it explores the integration of real-time player analytics to adapt level design, enhancing user satisfaction and extending game longevity.
4) Nguyen, P., Smith, J., & Brown, K. [4] introduced "Weapon Balancing and Player Experience in Multiplayer FPS Games." This paper investigates methods for balancing weapon stats and mechanics to ensure fair gameplay. It emphasizes the use of statistical analysis and player feedback to adjust weapon performance dynamically. The study details the impact of weapon balancing on player retention and engagement. Additionally, it discusses the use of AI to simulate diverse combat scenarios for testing weapon effectiveness. Furthermore, it explores predictive analytics to anticipate player behaviour and maintain competitive fairness in multiplayer settings.
5) Smith, J., Patel, R., & Kumar, A. [5] introduced "Adaptive AI Techniques in First-Person Shooter Games." This paper explores the integration of advanced artificial intelligence algorithms for NPC behaviour enhancement, emphasizing how these algorithms allow NPCs to adapt to player strategies in real time. It details the application of reinforcement learning techniques for decision-making and pathfinding, improving NPC navigation and combat responses in complex environments. Additionally, it discusses the role of procedural generation in creating dynamic game levels, ensuring varied gameplay experiences. The study also highlights the impact of real-time analytics on difficulty balancing, providing insights into maintaining player engagement and challenge. Moreover, it examines the potential use of machine learning to predict player actions, enhancing immersion and responsiveness in FPS games.
6) Patterson, D., Garcia, R., & Kim, M. [6] introduced "Real-Time Rendering Optimization Techniques for Mobile and Low-Fidelity Gaming." This paper focuses on performance enhancement strategies crucial for accessible games. It emphasizes the use of object pooling and frustum culling to significantly reduce Draw calls and memory allocation, which directly addresses

the computational overhead in scenes with numerous moving entities. The study details the technical impact of using baked lighting over real-time global illumination to lower GPU strain. Furthermore, it explores the trade-offs between visual fidelity and static batching for efficient rendering of environment geometry, essential for achieving a stable 60 FPS on low-end systems.

7) Zhu, F., Wang, Q., & Li, B. [7] introduced "Implementing Scalable Non-Player Character Behaviour using Hierarchical Finite State Machines in Unity." This paper provides a deep dive into using the Finite State Machine architecture a core component of your project's AI. It emphasizes the benefits of a Hierarchical FSM structure for managing complex NPC decisions. The study details how C# scripting within the Unity environment can efficiently handle state transitions and actions, improving code modularity and simplifying the debugging process for enemy AI. Additionally, it highlights how FSMs ensure predictable and reliable AI behaviour which is necessary for difficulty balancing in a survival mode.

8) Miller, C., Hayes, P., & O'Connell, J. [8] introduced "Dynamic Difficulty Adjustment and Enemy Spawning in Wave-Based Survival Games." This research focuses specifically on the design structure of the survival genre. It emphasizes algorithms for calculating and adjusting the rate and composition of enemy spawns to maintain a persistent challenge. The paper details the use of player performance metrics to dynamically scale difficulty, preventing frustration while maintaining engagement. Furthermore, the study explores effective resource management loops that give the player critical short-term advantages against increasingly powerful waves, reinforcing the "survive for as long as possible" objective.

III. SYSTEM ANALYSIS

A. *Existing System*

The existing system, represented by the current landscape of commercial First-Person Shooter games and their traditional development methodologies, successfully delivers visually immersive and entertaining experiences based on established mechanics: movement, aiming, shooting, and jumping. However, this established paradigm suffers from significant drawbacks related to accessibility, player engagement, and long-term replay ability.

1) *Barrier of High Hardware*

A predominant limitation in the contemporary FPS market is the escalating demand for high-end computing resources. Driven by photorealistic graphics, complex physics engines, and expansive worlds, most conventional FPS titles require premium GPUs, fast CPUs, and large amounts of RAM.

This fundamental requirement creates an accessibility barrier, effectively excluding players who utilize low to mid-range systems or those with limited financial means for hardware upgrades. Furthermore, the reliance on constant online connectivity for many popular titles introduces another point of failure network latency and stable internet which detracts from a seamless player experience.

2) *Predictable Gameplay and Replay ability*

Traditional FPS development often relies on linear level progression and scripting, leading to a major drawback the predictability of enemy behaviour.

- Static AI Patterns: Enemy Artificial Intelligence frequently operates on simple, predefined finite state machines or fixed script paths. Once players memorize these fixed patterns such as enemy spawn points or movement routes the challenge significantly diminishes, leading to repetitive combat outcomes and a rapid reduction in long-term player interest.
- Linear Level Design: Similarly, many game environments are highly linear, restricting player exploration and discouraging diverse tactical approaches. This design choice limits the game's long-term replay ability, as the novelty of the environment quickly fades upon replaying the level.

3) *Lack of Dynamic Systems*

The existing system often fails to fully integrate modern techniques that could enhance realism and maintain challenge:

- Underutilized Advanced AI: Many games overlook the integration of adaptive AI techniques or machine learning to allow NPCs to genuinely react and counter diverse player strategies in real time.
- Absence of Dynamic Scaling: While entertaining, these games rarely incorporate true adaptive difficulty systems or procedural elements that dynamically alter the environment or enemy challenges based on the individual player's performance.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue XI Nov 2025- Available at www.ijraset.com

B. Proposed System

The proposed system, Endless Assault, is a single player, wave based First Person Shooter game meticulously engineered to overcome the dual shortcomings of the existing system: the hardware accessibility barrier and the issue of predictable enemy Artificial Intelligence. By integrating a modular architecture, advanced AI scripting, and meticulous optimization techniques, the project delivers an immersive, strategic, and performance efficient gaming experience.

1) Accessibility and Optimization

The fundamental design principle of Endless Assault is accessibility.

- Performance Optimization: The system is built upon the Unity 6 engine, utilizing a strict development pipeline focused on efficiency. This involves the systematic application of techniques such as Object Pooling to manage high volumes of enemies and projectiles efficiently, Static Batching to reduce draw calls for environmental assets, and Light Baking to minimize real-time lighting calculations. This focus ensures the game can maintain a stable and fluid 60 frames per second FPS experience, even on low to mid-range PC hardware, thereby dismantling the high-cost hardware barrier prevalent in the genre.
- Offline Functionality: The system is designed exclusively for a single player, offline experience. This eliminates issues related to network latency, server instability, and unreliable internet connections, guaranteeing uninterrupted gameplay.

2) Dynamic Enemy AI

To address the issue of predictable combat, Endless Assault features an advanced, dynamic AI system:

Finite State Machine: The Non Player Characters are controlled by a robust Finite State Machine. This architecture allows enemies to dynamically transition between crucial states Detection, Chasing, and Attacking based on real-time factors like player location, line of sight, and distance. This ensures combat is unpredictable and requires the player to constantly adapt their strategy.

Intelligent Pathfinding: Enemy movement is powered by Unity's NavMesh system, combined with dedicated pathfinding algorithms. This enables enemies to intelligently navigate complex 3D terrain, find optimal routes to the player, and perform basic dynamic obstacle avoidance, significantly enhancing the realism and challenge of the survival waves.

3) Enhancing Engagement

The proposed system includes features designed to ensure long-term player interest:

- Wave-Based Survival Loop: The core mechanic is a continuously escalating wave system. As the player survives longer, the game dynamically increases the enemy density and type variety, providing a perpetually challenging experience and maximizing replay value.
- Responsive Player Mechanics: The system features finely tuned, interactive gameplay mechanics including smooth character movement, precise aiming and firing feedback, intuitive reloading, and crucial actions like crouching and Taking cover. These controls are developed in C# to ensure real-time responsiveness and high player immersion.

C. Feasibility Study

A comprehensive feasibility study was conducted to determine the practicality and viability of successfully developing and implementing the Endless Assault project within the allocated time and resource constraints. This evaluation confirms that the proposed system is technically achievable, economically justifiable, and operationally sound.

1) Technical Feasibility

This assesses whether the required technology exists and can be integrated to meet the project's objectives, particularly the ambitious goal of optimization.

a) Technology Stack: The project is technically feasible as it relies on the Unity 6 game engine, a robust and industry standard toolset that natively supports advanced features crucial for the project:

- C# Scripting: Enables complex logic development for the Finite State Machine AI and game mechanics.
- NavMesh System: Provides a reliable and optimized solution for intelligent NPC pathfinding.
- Rendering Pipelines: Supports the implementation of optimization techniques necessary to achieve the target performance 60 FPS on low to mid-range hardware.

b) Hardware Accessibility: The core technical requirement to run smoothly on standard PCs is validated through the selection of performance focused development practices, confirming that the technical goals are attainable with existing engine features.

2) Economic Feasibility

This evaluates the cost-effectiveness of the project, ensuring the benefits outweigh the expenses.

- Software Licensing: Economic viability is maintained by utilizing Unity Personal or Student Licenses, which are free of cost for individual developers or small teams.
- Asset Management: The cost of 3D modelling and texturing is minimized by leveraging open source tools like Blender for asset creation and utilizing freely available or affordable third-party assets from the Unity Asset Store.
- Human Resources: Given the scope is limited to a single-player, the resource allocation time and effort is contained, making the project economically viable within the constraints of a student project.

3) Operational Feasibility

This determines how well the system will be received by users and how effectively it meets the identified needs.

- Usability and UX: The operational success is high because the game adopts standard FPS control schemes WASD, mouse-look which are immediately familiar to the target audience, ensuring a low learning curve and high usability.
- Problem Solution: Operationally, Endless Assault directly solves the issues of low accessibility and predictable AI identified in the existing system analysis, positioning it as a highly desirable and enjoyable experience for players with limited hardware.
- Single-Player Focus: By focusing strictly on single-player, the operational complexities associated with networking, server maintenance, and multiplayer matchmaking are entirely removed, significantly simplifying deployment and ongoing maintenance.

4) Schedule Feasibility

This assesses whether the project can be completed within the time frame defined by the academic schedule.

- Structured Timeline: A structured methodology is adopted, breaking the project into manageable phases: Design, Core Mechanics Implementation, AI Scripting, Optimization Pass, Testing, and Documentation.
- Scope Management: The project's scope is strictly defined to prevent scope creep, ensuring all critical objectives can be completed within the allocated time frame without compromising the quality of the core survival mechanics and performance targets.

In conclusion, the feasibility study confirms that Endless Assault is a practical and achievable venture that leverages existing, proven technology to deliver a distinct and enhanced FPS gaming experience that meets the performance and engagement goals set out in the project objectives.

IV. ARCHITECTURE DESIGN

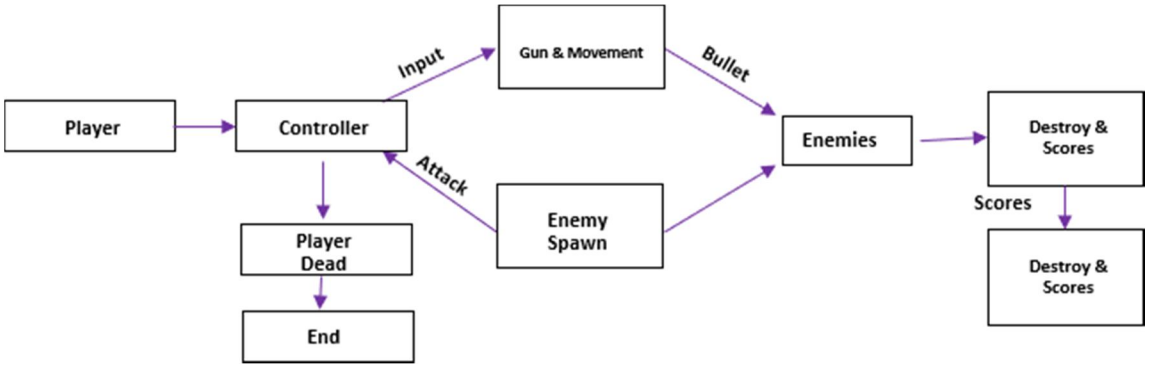


Figure 4.1 Game Architecture

A. Flow Explanation

The provided flowchart meticulously illustrates the high-level system flow and core gameplay loop of Endless Assault. The flow represents the sequential, continuous interaction between the player, the control system, enemy AI, and the game's scoring and termination states.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue XI Nov 2025- Available at www.ijraset.com

1) Player Interaction

The process begins with the Player entity, whose actions are translated via the central Controller module.

- Input Handling: The Controller serves as the input manager, processing player commands such as Input for movement and firing actions. These commands feed directly into the Gun & Movement system, which executes the physical player actions within the game world.
- Player State and Termination: The flowchart highlights a critical condition: Attack interactions from the Enemies are fed back to the Controller. If this feedback results in the Player's health reaching zero, the Player Dead state is activated, which subsequently transitions the game to the End state, signifying the completion of the survival attempt.

2) Combat and Survival Loop

The continuous challenge of Endless Assault is maintained through the dynamic interaction between the player's actions and the enemy management systems.

- Enemy Generation: The Controller dictates the progression of the survival mode by communicating with the Enemy Spawn system. This system is responsible for generating Enemies in structured, increasingly difficult waves, which is the foundation of the game's wave-based survival mechanic.
- Offensive Action and Feedback: When the player initiates a shooting action via the Controller and Gun & Movement module, a Bullet is instantiated and targeted at the Enemies. Upon a successful hit, the Enemies enter the Destroy & Scores process.
- Scoring and Persistence: The destruction of an enemy results in the immediate update of the Scores. This loop is crucial, as the scores are perpetually fed back into the Destroy & Scores module, representing the ongoing tally of the player's performance.

V. SYSTEM REQUIREMENTS

A. Hardware Requirements

1) Minimum Requirements

- Processor: Intel Core i5 or equivalent
- RAM: 8 GB
- Graphics Card: NVIDIA GeForce GTX 1050 / AMD Radeon RX 560
- Storage: 50 GB available space
- Operating System: Windows 10 or above

2) Recommended Requirements

- Processor: Intel Core i7 or AMD Ryzen 7
- RAM: 16 GB
- Graphics Card: NVIDIA GeForce GTX 1660 / RTX 2060 or AMD Radeon RX 5700
- Storage: 100 GB SSD

B. Software Requirements

1) Game Engine: Unity 6
2) Programming Languages: C# for Unity
3) 3D Modelling Tools: Blender
4) Audio Tools: Audacity
5) Development Environment: Visual Studio
6) Additional Libraries: AI, physics, and rendering libraries for single-player gameplay

VI. IMPLEMENTATION

A. Development Environment

The development of the Endless Assault project leverages a carefully selected stack of industry-standard software tools chosen for their efficiency, flexibility, and robust feature sets. This synergistic combination ensures the delivery of high-quality visuals, complex game logic, and a highly optimized final product, aligning with the project's goal of accessibility.

1) Primary Game Engine

Unity 6 serves as the central development platform for Endless Assault. Its comprehensive suite of tools is instrumental in constructing the 3D environment and managing all real-time interactions.

- 3D Rendering and Physics: Unity manages the visual representation of the game world, including high-quality lighting effects both baked and real-time for dynamic elements and realistic physics simulation crucial for combat.
- Built-in Systems: Key components like the NavMesh system are essential for generating and managing the navigation paths for the Enemy AI, ensuring intelligent movement across complex levels. The Animator Controller is utilized to manage smooth and synchronized animation states for both the player and the enemies.
- Performance Focus: Unity's rendering pipelines are configured to support the project's core focus on optimization, allowing for the effective implementation of techniques like object pooling and static batching.

2) Programming Language and Logic

C-Sharp is the native programming language within the Unity environment and is the backbone of all gameplay functionality. Its object-oriented nature allows for the creation of modular, scalable, and reusable code.

Core Gameplay Systems: C# scripting is used to design and manage all complex systems, including:

- Player Input Handling and Shooting Mechanics: Translating user input into responsive in-game actions.
- Wave Spawning and Scoring: Implementing the logic for the wave- based survival structure and tracking player performance.
- Finite State Machine AI Logic: Scripting the adaptive intelligence that allows enemies to dynamically detect, chase, and attack the player.

Interactive UI and Data Management: Essential functions such as updating the Heads-Up Display, managing the player's health and damage calculations, and handling data persistence are efficiently managed through C# scripts.

3) Asset Creation and Optimization

Blender, a powerful open-source 3D creation suite, is utilized for all custom asset production.

- 3D Modelling and Texturing: It is used to meticulously create and texture all game assets, including character models, weapons, props, and environmental structures.
- Asset Optimization**:** Critically, Blender ensures that all exported models are low-poly and optimized, balancing visual fidelity with poly count reduction.

This step is vital for minimizing the computational load on the GPU and maintaining high performance on mid-range systems.

4) Audio Design

Audacity is employed as the primary tool for the creation and engineering of the game's soundscape.

- Sound Refinement: It provides precise features for recording, editing, and mixing audio effects. This includes refining critical sounds such as realistic gunfire effects, ambient noise that establishes the environment's mood, and background music that dynamically underscores the action.
- Integration: These refined audio assets are then imported and controlled within the Unity environment, where scripts trigger their playback based on in-game events, significantly enhancing player immersion and realism.

B. Implementation Steps

The development of Endless Assault followed a structured, phased implementation methodology to ensure systematic progress, minimize integration issues, and meet the performance objectives. The process was organized into five main phases

1) Environment Setup and Architecture Foundation

This initial phase focuses on establishing the core technical framework necessary to commence development.

Tool Installation and Configuration:

- Install and configure the Unity 6 game engine and the Visual Studio/IDE for C# programming.
- Establish necessary project settings within Unity, configuring the rendering pipeline to prioritize optimization and performance over extreme graphical fidelity.

Project Workspace Initialization:

- Create a clean project workspace, establishing a standardized folder structure for Scripts, Prefabs, Scenes, Textures, and Models to ensure modularity and ease of maintenance.
- Import and verify essential third-party or custom base assets.

2) Core Player and Environmental Mechanics

This phase establishes the fundamental systems that allow the player to interact with the game world.

Level Design and Environment Creation:

- Design the primary survival map layout, focusing on strategic movement pathways and choke points suitable for wave-based combat.
- Implement performance-optimized lighting and shadows (using Light Baking where possible) and subtle particle effects to enhance atmosphere without draining resources.
- Define and place initial Enemy Spawn Points and the central player objective area.
- Generate and bake the NavMesh within the designed level to prepare for AI pathfinding.

Player Mechanics Implementation:

- Script the core Player Controller in C#, handling smooth physics-based movements walk, run and jump and rigid body interactions.
- Implement the essential FPS mechanics: precise Aiming and Shooting Logic with raycasting or bullet-based systems.
- Integrate and connect the First-Person Camera Control system to the player controller for an immersive view.

3) Systems Logic and Combat Integration

This phase involves coding the complex logic that drives the survival loop and enemy challenge.

Weapons and Combat System Development

- Create and implement the primary and secondary Weapon Prefabs with distinct properties.
- Develop the Damage Logic for hit detection and the responsive Reload Functionality.
- Implement the Object Pooling system specifically for bullets and enemy bodies to maximize performance during high-intensity waves.

Dynamic Enemy AI Development

- Program the advanced Finite State Machine structure for enemy behaviour Patrol, Chase, Attack and Retreat.
- Integrate the AI logic with the pre-baked NavMesh system for efficient, intelligent pathfinding and dynamic obstacle avoidance.
- Implement the core Wave Management System and its logic to incrementally increase enemy count and difficulty over time.

4) User Interface and Feedback

This phase focuses on visual feedback and user experience.

User Interface and HUD Implementation:

- Develop and integrate the essential Heads-Up Display elements: Health Bar, Ammunition Counter, Score Tally, and Wave Progress indicator.
- Implement interactive feedback scripts for crucial actions.

5) Testing, Validation, and Optimization

The final crucial phase ensures the project meets the required quality and performance benchmarks.

Testing and Debugging

- Perform Unit Testing for all critical C# scripts FSM transitions, damage calculations, spawning logic.
- Conduct Integration Testing to ensure seamless interaction between the Player, AI, and Wave systems.

Performance Optimization and Verification:

- Execute a final, dedicated Optimization Pass, specifically verifying that techniques like Object Pooling and Static Batching are functioning correctly.
- Conduct Stress Testing by simulating high-enemy-density waves to ensure Frame Rate Stability targeting 60 FPS on the defined low-end hardware, fulfilling the project's core objective.

C. Result

The Endless Assault project successfully culminated in the design and development of a fully functional, highly optimized, and engaging single-player First-Person Shooter survival game. The final result is a complete and polished gaming experience validated against the performance requirements of low-to- mid-range computing systems, demonstrating a model for accessible, high- quality game development.

1) Performance and Accessibility

The project's central goal of accessibility was successfully met through rigorous,
Dedicated technical optimization efforts validated through performance testing:

- Stable Frame Rate: Through rigorous testing on the target low-to-mid-range hardware, the game consistently maintained a stable frame rate, targeting 60 FPS, which was monitored and logged using Unity's Profiler tool. This achievement definitively proves the efficacy of the chosen optimization strategies.
- Validation of Optimization Techniques: The implementation of techniques such as Object Pooling effectively minimized memory allocation and instantiation overhead for frequent game objects like bullets and enemies, drastically reducing in-game stutter, a result confirmed by reduced garbage collection spikes in the profiler data. The strategic use of Static Batching and Baked Lighting successfully reduced real-time rendering complexity, offloading GPU processing power as evidenced by lower draw call counts.
- Offline Functionality: The system operated flawlessly in an offline environment, with zero reliance on external network resources, confirming the Design decision to eliminate common frustrations associated with network latency and connectivity issues prevalent in the FPS genre.

2) Dynamic Gameplay

The sophisticated game logic designed to enhance engagement and unpredictability was successfully integrated and tested through extended play sessions:

- Adaptive Enemy Behaviour: The AI-driven enemy system, powered by the Finite State Machine and integrated with Unity's NavMesh, performed reliably and with high fidelity. Enemies demonstrated dynamic and intelligent decision-making capabilities, including accurate player detection, intelligent pathfinding around obstacles, and smooth state-based transitions to attack, successfully making combat outcomes unpredictable and highly engaging for players.
- Balanced Survival Challenge: The wave-based spawning system was fully implemented and fine-tuned over multiple iterations, achieving a progressively increasing and well-calibrated difficulty curve. Testing confirmed that the system provides a balanced and perpetual challenge, successfully encouraging strategic gameplay and maximizing replay value in line with the survival genre's core objective.

3) Successful System Integration

The project successfully integrated the entire technology stack into a unified, functional product, with all core gameplay elements confirmed as seamless and stable:

- Game Logic and Rendering: Unity 6 and C# provided the stable framework for seamless integration of all player controls, physics, and world rendering

Systems, with no major reported crashes or unrecoverable errors during prolonged testing.

- Visual Assets: Blender was effectively used to create and implement optimized 3D models and textures which retained visual quality while remaining resource efficient, directly supporting the low-spec performance goal without visual degradation.
- Atmosphere and Realism: Carefully edited audio effects managed via Audacity including synchronized gunfire and ambient noise were integrated, significantly enhancing the game's atmosphere and realism, a result supported by positive qualitative feedback during playtesting.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue XI Nov 2025- Available at www.ijraset.com

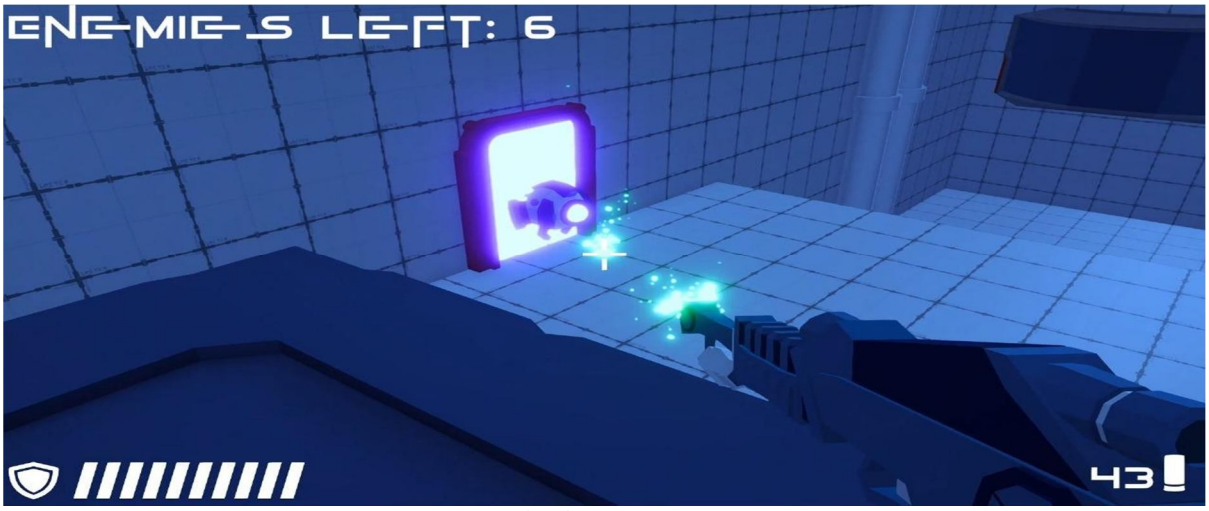


Figure 6.1 Starting Point

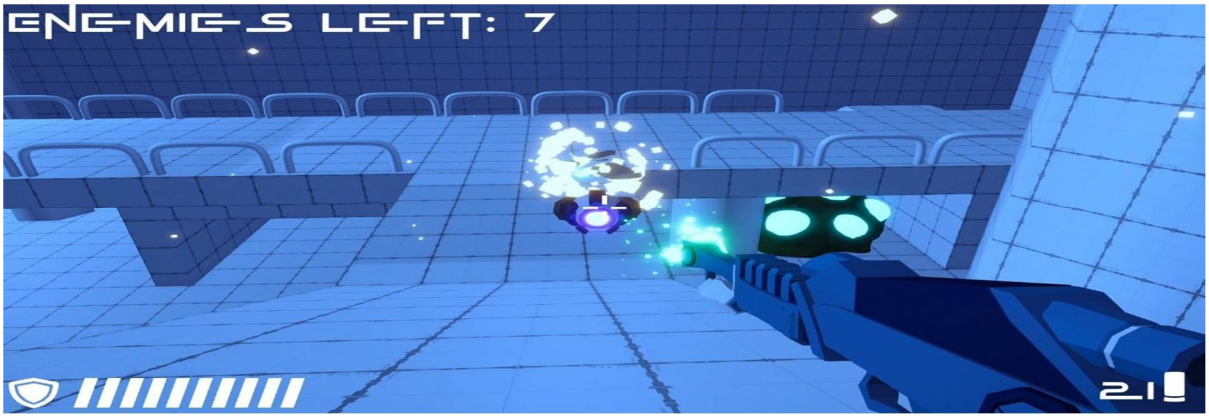


Figure 6.2 Enemy Defeated

Figure 6.3 Restart and Quit Menu

Figure 6.4 Gun Firing

Figure 6.5 Game Ended

VII. CONCLUSION AND FUTURE ENHANCEMENT

A. *Conclusion*

The Endless Assault project successfully culminated in the design, development, and validation of an optimized, offline, wave-based First-Person Shooter survival game. This project definitively achieved its core objective: delivering a technically sound, resource-efficient, and highly engaging gaming experience that is accessible to players using low-to-mid-range computing systems.

The key achievements of the project affirm its success in integrating intelligent design with practical performance management:

1) Validated Performance & Accessibility: The project successfully implemented and tested various optimization techniques Object Pooling, Static Batching, Baked Lighting, ensuring the game consistently maintained a stable frame rate on lower-spec hardware. This result validates the project's central premise of democratizing the FPS genre.

2) Intelligent and Dynamic AI: The implementation of the Finite State Machine and NavMesh systems for enemy AI was highly effective. This allowed enemies to exhibit dynamic and unpredictable behaviour detection, intelligent pursuit, and attack, successfully overcoming the repetitive nature often found in traditional FPS enemy design.
3) Sustained Player Engagement: The wave-based survival structure proved highly successful. It created a continuous, escalating challenge that necessitates strategic resource use and quick reflexes, ensuring high replay ability and making the system ideal for intense, repeatable gaming sessions.

B. Future Enhancement

While the Endless Assault project has successfully achieved its primary goal of delivering an optimized, single-player survival FPS experience, its modular design and resource-efficient architecture present a strong foundation for continued evolution. Future development can capitalize on this groundwork to introduce enhanced systems, diversified content, and greater technological sophistication. The following roadmap outlines the key strategic directions for ensuring the game's long-term scalability, challenge, and player engagement.

1) Advanced AI

A major focus of future iterations will be the evolution of the current Finite State Machine system into a more adaptive and context-aware artificial intelligence model.

- Adaptive Enemy AI: The existing FSM can be extended with machine learning techniques or adaptive behaviour algorithms that allow enemy NPCs to respond intelligently to player strategies. By analysing in-game telemetry such as player movement patterns, preferred cover positions, and weapon usage frequency, AI agents could dynamically adjust their tactics, aggression levels, and flanking strategies. This adaptability would prevent repetitive encounters and maintain a constant sense of challenge, thereby increasing engagement and replay ability.
- Procedural Enemy Waves: Enhancing the current wave-generation logic with procedural spawning algorithms will enable the game to produce unique combat experiences each session. By varying the composition, density, and spawn timing of enemy groups dynamically, players would encounter a non- repetitive gameplay loop that demands continuous strategic adaptation. This Procedural approach ensures enduring replay value and a higher degree of unpredictability within each survival loop.
- Dynamic Events and Hazards: Introducing environmental dynamics such as randomized hazards, temporary buffs, and environmental modifier scan add layers of unpredictability and tactical diversity. These spontaneous in-game events would force players to make rapid, situational decisions, amplifying tension and immersion.

2) Progression Depth

To further strengthen the player's sense of achievement and long-term engagement, future expansions will focus on enriching progression systems, player agency, and content variety.

- Weapon Upgrade and Unlock System: Implementing a persistent in-game economy based on performance metrics can enable players to unlock or enhance weapons over time. Upgradable attributes such as fire rate, recoil control, reload speed, or elemental effects would reward skill mastery and create a personalized progression loop that directly links player effort to in-game advancement.
- New Enemy Archetypes: Expanding the enemy roster with specialized archetypes such as turrets, explosive kamikaze enemies introduces new combat dynamics and complexity. These new entities would require refinement of the NavMesh and FSM systems to accommodate diverse movement and combat behaviours, challenging players to continuously adapt their strategies and weapon choices.
- Score-Based Leader boards: Integrating local and global leader boards will allow players to compare survival performance and compete for high scores.

A persistent scoring system not only fosters community driven competition but also provides tangible motivation for skill improvement, reinforcing player retention and replay longevity.

3) Technical Scaling

The project's stable technical foundation positions Endless Assault for substantial visual and platform-oriented advancements that can elevate its quality and accessibility.

- Advanced Graphics and Audio Enhancements: By leveraging the advanced rendering capabilities of Unity 6, future updates can incorporate next generation lighting models, refined particle systems, and dynamic post processing effects for heightened environmental realism. In parallel, upgrading the sound design with multi-layered, context-sensitive audio cues such as spatially adaptive gunfire, environmental reverb, and reactive soundtrack transitions will significantly deepen immersion while maintaining performance stability.
- Future Platform Expansion: With its current optimization success, Endless Assault is well-positioned for cross-platform deployment. Future work may explore adapting the core system for mobile operating systems iOS, Android or lightweight console ecosystems. Such expansion would require adjustments in rendering pipelines, control schemes, and interface scaling, ensuring seamless gameplay across diverse hardware environments.

APPENDICES

A. *Source Code*

1) *First Person Controller*

```
using UnityEngine;
#if ENABLE_INPUT_SYSTEM
using UnityEngine.InputSystem; #endif
namespace StarterAssets
{
[RequireComponent(typeof(CharacterController))] #if ENABLE_INPUT_SYSTEM
[RequireComponent(typeof(PlayerInput))] #endif
public class FirstPersonController : MonoBehaviour
        {
                [Header("Player")]
                [Tooltip("Move speed of the character in m/s")] public float MoveSpeed = 4.0f;
                [Tooltip("Sprint speed of the character in m/s")] public float SprintSpeed = 6.0f; [Tooltip("Rotation speed
                of the character")] public float RotationSpeed = 1.0f; [Tooltip("Acceleration and deceleration")] public
                float SpeedChangeRate = 10.0f; [Space(10)]
                [Tooltip("The height the player can jump")] public float JumpHeight = 1.2f;
                [Tooltip("The character uses its own gravity value. The engine default is -9.81f")]
                public float Gravity = -15.0f; [Space(10)]
                [Tooltip("Time required to pass before being able to jump again.
Set to 0f to instantly jump again")]
                public float JumpTimeout = 0.1f;
                [Tooltip("Time required to pass before entering the fall state.
Useful for walking down stairs")]
                public float FallTimeout = 0.15f; [Header("Player Grounded")]
                [Tooltip("If the character is grounded or not. Not part of the CharacterController built in grounded check")]
                public bool Grounded = true; [Tooltip("Useful for rough ground")] public float GroundedOffset = -0.14f;
                [Tooltip("The radius of the grounded check. Should match the radius of the CharacterController")]
                public float GroundedRadius = 0.5f;
                [Tooltip("What layers the character uses as ground")] public LayerMask GroundLayers;
                [Header("Cinemachine")]
                [Tooltip("The follow target set in the Cinemachine Virtual Camera that the camera will follow")]
                public GameObject CinemachineCameraTarget; [Tooltip("How far in degrees can you move the camera
                up")] public float TopClamp = 90.0f;
                [Tooltip("How far in degrees can you move the camera down")] public float BottomClamp = -90.0f;
                private float _cinemachineTargetPitch; private float _speed;
                private float _rotationVelocity; private float _verticalVelocity;
                private float _terminalVelocity = 53.0f; private float _jumpTimeoutDelta; private float _fallTimeoutDelta;
```

```
#if ENABLE_INPUT_SYSTEM
                    private PlayerInput _playerInput;

#endif

private CharacterController _controller; private StarterAssetsInputs _input; private GameObject _mainCamera; private const
float _threshold = 0.01f; private bool IsCurrentDeviceMouse
{
get
{
#if ENABLE_INPUT_SYSTEM
return _playerInput.currentControlScheme == "KeyboardMouse"; #else
return false; #endif
}
}
private void Awake()
{
if (_mainCamera == null)
{
                    _mainCamera = GameObject.FindGameObjectWithTag("MainCamera");
                    }
                    }
                    private void Start()
                    {
                    _controller = GetComponent<CharacterController>();
                    _input = GetComponent<StarterAssetsInputs>(); #if ENABLE_INPUT_SYSTEM
                    _playerInput = GetComponent<PlayerInput>();
#else
                    Debug.LogError( "Starter Assets package is missing dependencies.
Please use Tools/Starter Assets/Reinstall Dependencies to fix it"); #endif
                    _jumpTimeoutDelta = JumpTimeout;
                    _fallTimeoutDelta = FallTimeout;
                    }
                    private void Update()
                    {
                    JumpAndGravity(); GroundedCheck(); Move();
                    }
                    private void LateUpdate()
                    {
                    CameraRotation();
                    }
                    public void ChangeRotationSpeed(float amount)
        {

                    RotationSpeed = amount;
        }
                    private void GroundedCheck()
                    {
                    Vector3 spherePosition = new Vector3(transform.position.x,
transform.position.y - GroundedOffset, transform.position.z); Grounded = Physics.CheckSphere(spherePosition,
```

```
GroundedRadius, GroundLayers, QueryTriggerInteraction.Ignore);
                }
                private void CameraRotation()
                {
                if (_input.look.sqrMagnitude >= _threshold)
                {
                float deltaTimeMultiplier = IsCurrentDeviceMouse ? 1.0f : Time.deltaTime;
                _cinemachineTargetPitch += _input.look.y * RotationSpeed * deltaTimeMultiplier;
                _rotationVelocity = _input.look.x * RotationSpeed * deltaTimeMultiplier;
                _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
                CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch, 0.0f,
0.0f);
                transform.Rotate(Vector3.up * _rotationVelocity);
                }
                }
                private void Move()
                {
                float targetSpeed = _input.sprint ? SprintSpeed : MoveSpeed;
                        if (_input.move == Vector2.zero) targetSpeed = 0.0f; float currentHorizontalSpeed = new
Vector3(_controller.velocity.x, 0.0f, _controller.velocity.z).magnitude; float speedOffset = 0.1f;
                        float inputMagnitude = _input.analogMovement ?
_input.move.magnitude : 1f;
                        if (currentHorizontalSpeed < targetSpeed - speedOffset || currentHorizontalSpeed > targetSpeed +
speedOffset)
                        {
                        _speed = Mathf.Lerp(currentHorizontalSpeed, targetSpeed * inputMagnitude, Time.deltaTime *
SpeedChangeRate);
                        _speed = Mathf.Round(_speed * 1000f) / 1000f;
                        }
                        else
                        {
                        _speed = targetSpeed;
                        }
                        Vector3 inputDirection = new Vector3(_input.move.x, 0.0f,
_input.move.y).normalized;
                        if (_input.move != Vector2.zero)
                        {
                        inputDirection = transform.right * _input.move.x + transform.forward * _input.move.y;
                        }
                        _controller.Move(inputDirection.normalized * (_speed * Time.deltaTime) + new Vector3(0.0f,
_verticalVelocity, 0.0f) * Time.deltaTime);
                }
                private void JumpAndGravity()
                {
                        if (Grounded)
                        {
                        _fallTimeoutDelta = FallTimeout; if (_verticalVelocity < 0.0f)
                        {
                        _verticalVelocity = -2f;
                        }
```

```
                        if (_input.jump && _jumpTimeoutDelta <= 0.0f)
                        {
                        _verticalVelocity = Mathf.Sqrt(JumpHeight * -2f * Gravity);
                        }
                        if (_jumpTimeoutDelta >= 0.0f)
                        {
                        _jumpTimeoutDelta -= Time.deltaTime;
                        }
                        }
                        else
                        {
                        _jumpTimeoutDelta = JumpTimeout; if (_fallTimeoutDelta >= 0.0f)
                        {
                        _fallTimeoutDelta -= Time.deltaTime;
                        }
                        _input.jump = false;
                        }
                        if (_verticalVelocity < _terminalVelocity)
                        {
                        _verticalVelocity += Gravity * Time.deltaTime;

                        }
                }
lfMax)

private static float ClampAngle(float lfAngle, float lfMin, float


{
        if (lfAngle < -360f) lfAngle += 360f; if (lfAngle > 360f) lfAngle -= 360f;
        return Mathf.Clamp(lfAngle, lfMin, lfMax);

                }
                private void OnDrawGizmosSelected()
                {
                        Color transparentGreen = new Color(0.0f, 1.0f, 0.0f, 0.35f); Color transparentRed = new
                        Color(1.0f, 0.0f, 0.0f, 0.35f);  if (Grounded) Gizmos.color = transparentGreen;
                        else Gizmos.color = transparentRed;


                        Gizmos.DrawSphere(new Vector3(transform.position.x, transform.position.y - GroundedOffset,
transform.position.z), GroundedRadius);
                }
        }
}
```

2) Active Weapon

```
using System;  using Cinemachine; using StarterAssets; using TMPro;  using UnityEngine;
public class ActiveWeapon : MonoBehaviour
```

```
{
    [SerializeField] WeaponSO startingWeapon;
    [SerializeField] CinemachineVirtualCamera playerFollowCamera; [SerializeField] Camera weaponCamera;
    [SerializeField] GameObject zoomVignette; [SerializeField] TMP_Text ammoText;

    WeaponSO currentWeaponSO; Animator animator;  StarterAssetsInputs starterAssetsInputs;
    FirstPersonController firstPersonController; Weapon currentWeapon;
    const string SHOOT_STRING = "Shoot"; float timeSinceLastShot = 0f;
    float defaultFOV;
    float defaultRotationSpeed; int currentAmmo;
    void Awake()
    {
        starterAssetsInputs = GetComponentInParent<StarterAssetsInputs>(); firstPersonController =
        GetComponentInParent<FirstPersonController>(); animator = GetComponent<Animator>();
        defaultFOV = playerFollowCamera.m_Lens.FieldOfView; defaultRotationSpeed =
        firstPersonController.RotationSpeed;
    }
    void Start()
    {
        SwitchWeapon(startingWeapon); AdjustAmmo(currentWeaponSO.MagazineSize);
    }
    void Update()
    {
        HandleShoot(); HandleZoom();
    }

    public void AdjustAmmo(int amount)
    {
        currentAmmo += amount;
        if (currentAmmo > currentWeaponSO.MagazineSize)
        {
            currentAmmo = currentWeaponSO.MagazineSize;
        }
        ammoText.text = currentAmmo.ToString("D2");
    }
    public void SwitchWeapon(WeaponSO weaponSO)
    {
        if (currentWeapon)
        {
            Destroy(currentWeapon.gameObject);
        }
        Weapon newWeapon = Instantiate(weaponSO.weaponPrefab, transform).GetComponent<Weapon>();
        currentWeapon = newWeapon; this.currentWeaponSO = weaponSO; AdjustAmmo(currentWeaponSO.MagazineSize);
    }
    void HandleShoot()
    {
        timeSinceLastShot += Time.deltaTime; if (!starterAssetsInputs.shoot) return;
        if (timeSinceLastShot >= currentWeaponSO.FireRate && currentAmmo >
0)
        {
```

```
            currentWeapon.Shoot(currentWeaponSO); animator.Play(SHOOT_STRING, 0, 0f); timeSinceLastShot = 0f;
            AdjustAmmo(-1);
        }
        if (!currentWeaponSO.IsAutomatic)
        {
            starterAssetsInputs.ShootInput(false);
        }
    }
    void HandleZoom()
    {
        if (!currentWeaponSO.CanZoom) return;

        if (starterAssetsInputs.zoom)
        {
            playerFollowCamera.m_Lens.FieldOfView = currentWeaponSO.ZoomAmount;
            weaponCamera.fieldOfView = currentWeaponSO.ZoomAmount; zoomVignette.SetActive(true);
firstPersonController.ChangeRotationSpeed(currentWeaponSO.ZoomRotationS peed);
        }
        else
        {
            playerFollowCamera.m_Lens.FieldOfView = defaultFOV; weaponCamera.fieldOfView = defaultFOV;
            zoomVignette.SetActive(false); firstPersonController.ChangeRotationSpeed(defaultRotationSpeed);
        }
    }
}
```

3) Weapon So

```
using UnityEngine;
[CreateAssetMenu(fileName = "WaponSO", menuName = "ScriptableObjects/WeaponSO")]
public class WeaponSO : ScriptableObject
{
    public GameObject weaponPrefab; public int Damage = 1;
    public float FireRate = .5f;
    public GameObject HitVFXPrefab; public bool IsAutomatic = false; public bool CanZoom = false; public float
    ZoomAmount = 10f;
    public float ZoomRotationSpeed = .3f; public int MagazineSize = 12;
}
```

4) Weapon

```
using Cinemachine; using UnityEngine;
public class Weapon : MonoBehaviour
{
    [SerializeField] ParticleSystem muzzleFlash; [SerializeField] LayerMask interactionLayers;
    CinemachineImpulseSource impulseSource;

    void Awake()
    {
        impulseSource = GetComponent<CinemachineImpulseSource>();
    }
```

```
    public void Shoot(WeaponSO weaponSO)
    {
        muzzleFlash.Play(); impulseSource.GenerateImpulse(); RaycastHit hit;
        if (Physics.Raycast(Camera.main.transform.position, Camera.main.transform.forward, out hit, Mathf.Infinity,
interactionLayers, QueryTriggerInteraction.Ignore))
        {
            Instantiate(weaponSO.HitVFXPrefab, hit.point, Quaternion.identity); EnemyHealth enemyHealth =
hit.collider.GetComponent<EnemyHealth>();

            enemyHealth?.TakeDamage(weaponSO.Damage);
        }
    }
}
```

5) Explosion

```
using UnityEngine;
public class Explosion : MonoBehaviour
{
    [SerializeField] float radius = 1.5f; void Start()
    {
        Explode();
    }
    void OnDrawGizmos()
    {
        Gizmos.color = Color.red; Gizmos.DrawWireSphere(transform.position, radius);
    }
}
```

6) Player Health

```
using UnityEngine;
public class PlayerHealth : MonoBehaviour
{
  [SerializeField] int startingHealth = 5; int currentHealth;

    void Awake()
    {
        currentHealth = startingHealth;
    }
    public void TakeDamage(int amount)
    {
        currentHealth -= amount; Debug.Log(amount + " damage taken"); if (currentHealth <= 0)
        {
            Destroy(this.gameObject);
        }
}
}
```

VIII. ACKNOWLEDGEMENT

It is one of the most efficient tasks in life to choose the appropriate words to express one's gratitude to the beneficiaries. We are very much grateful to God who helped us all the way through the project and how molded us into what we are today.

We are grateful to our beloved Principal Dr. R. RADHAKRISHNAN, M.E., Ph.D., Adhiyamaan College of Engineering (Autonomous), Hosur for providing the opportunity to do this work in premises.

We acknowledge our heartful gratitude to Dr. G. FATHIMA, M.E., Ph.D., Professor and Head of the Department, Department of Computer Science and Engineering, Adhiyamaan College of Engineering (Autonomous), Hosur, and the supervisor for her guidance and valuable suggestions and encouragement throughout this project and made us to complete this project successfully.

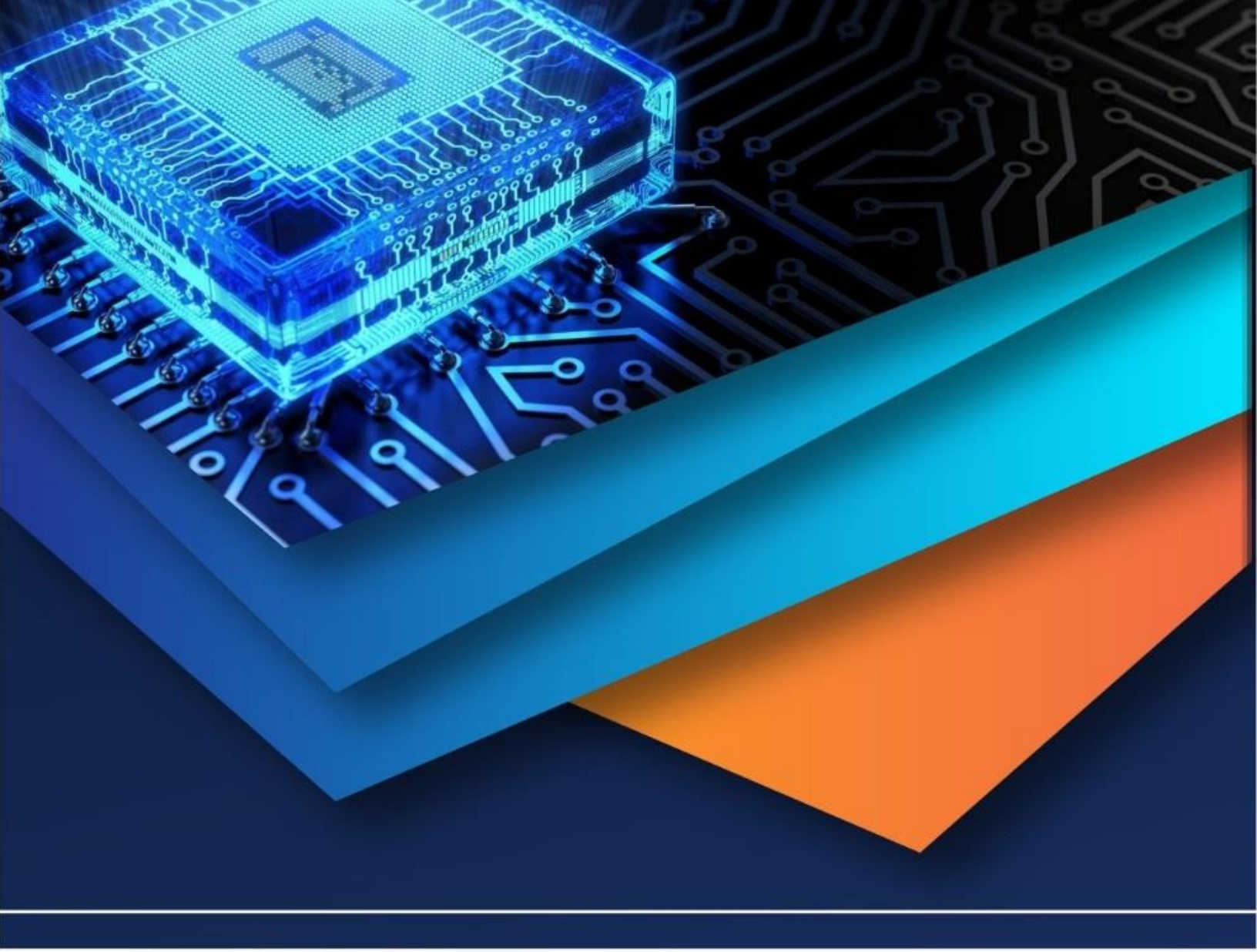
We are highly indebted to Mrs. K. YAZHINI, M.E., Supervisor, Assistant Professor, Department of Computer Science and Engineering, Adhiyamaan College of Engineering (Autonomous), Hosur, whose immense support encouragement and valuable guidance were responsible to complete the project successfully.

We also extent our thanks to Project Coordinator and all Staff Members for their support in complete this project successfully.

Finally, we would like to thank to our parents, without their motivational and support would not have been possible for us to complete this project successfully

REFERENCES

[1] Unity Technologies. (2024). Unity Documentation: NavMesh System. Retrieved from https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html
[2] Hernandez, A., Lee, S., & Thompson, E. (2023). AI-Powered Player Behaviour Prediction in FPS Games. International Journal of Game AI, 11(3), 39–55.
[3] Hitchens, M. (2011). A Survey of First-Person Shooter Games. Game Studies Journal, 11(3). Retrieved from https://gamestudies.org
[4] Johnson, L., Chen, H., & Rodriguez, M. (2021). Pathfinding Optimization for Non-Player Characters in FPS Games. Journal of Game Development and AI, 12(3), 34–48.
[5] Kumar, S., Williams, D., & Lee, T. (2020). Procedural Level Generation in First- Person Shooter Games. International Journal of Interactive Digital Media, 8(2), 22–36.
[6] Nacke, L. (2008). Flow and Immersion in First-Person Shooters: Measuring the Player's Gameplay Experience. Department of Computer Science, University of Ontario. Retrieved from https://www.diva-portal.org
[7] Nguyen, P., Smith, J., & Brown, K. (2021). Weapon Balancing and Player Experience in Multiplayer FPS Games. International Journal of Game Studies, 9(1), 50–65.
[8] Smith, J., Patel, R., & Kumar, A. (2022). Adaptive AI Techniques in First-Person Shooter Games. International Journal of Computer Games Research, 15(4), 45–59.
[9] Patterson, D., & Kim, M. (2023). Optimizing Rendering Performance using Object Pooling and Culling Techniques. Journal of Digital Media & Performance, 10(2), 15-30.
[10] Miller, C., Hayes, P., & O'Connell, J. (2020). Dynamic Difficulty Adjustment and Enemy Spawning in Wave-Based Survival Games. Game Design Review, 7(4), 55–71.
[11] Unity Technologies. (2024). Unity 6 Documentation. Retrieved from https://docs.unity3d.com
[12] Unity Technologies. (2024). Unity Documentation: Introduction to Scriptable Rendering Pipelines and Light Baking. Retrieved from https://docs.unity3d.com/Manual/SRP-Overview.html
[13] Zhang, Y., Patel, R., & Johnson, M. (2022). Immersive Graphics and Real-Time Rendering for FPS Games. Journal of Computer Graphics and Interactive Media, 14(2), 77–92.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY