ARSENAL 3D

A Project Report

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING IN AIT-CSE

Submitted by:

SAYAN KUMAR DAS- 20BCS3943

Under the Supervision of:

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING APEX INSTITUTE OF TECHNOLOGY

CHANDIGARH UNIVERSITY, GHARUAN, MOHALI - 140413, PUNJAB

APRIL 2025

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

BONA FIDE CERTIFICATE

I certify that this project report on **"ARSENAL 3D"** is the bona fide work of **SAYAN KUMAR DAS**, who carried out the project work under my supervision.

SIGNATURE

Mr. Aman Kaushik

HEAD OF DEPARTMENT

Apex Institute of Technology

Submitted for the project viva-voice examination held on

INTERNAL EXAMINER **EXTERNAL EXAMINER**

Table of Contents

ABSTRACT

Arsenal 3D is a first-person shooter (FPS) game developed using the Unity game engine, aimed at delivering an interactive and visually engaging experience through a stylized 3D environment. The project incorporates core gameplay mechanics such as player navigation, shooting, and basic enemy encounters, along with functional user interface elements. All in-game models and assets were either created or refined in Blender and seamlessly imported into Unity, allowing for a hands-on understanding of both modeling and game integration processes. This game was designed with simplicity in mind, focusing on solidifying the foundational concepts of 3D game development including scene setup, object interaction, and player control scripting.

Beyond the technical aspects, *Arsenal 3D* represents a collaborative learning experience between team members, combining creativity with structured development practices. The objective of the project was to gain practical exposure to the entire lifecycle of game creation—from concept design and asset development to logic scripting and playtesting. Throughout the development process, the team encountered various challenges related to design balance, scripting logic, and system performance, each of which contributed to a deeper understanding of Unity's workflow and the importance of iterative testing. Ultimately, the project reflects a successful application of theoretical knowledge into a real-world game environment, serving as a strong base for more advanced projects in the future

CHAPTER - 1
INTRODUCTION

The gaming industry has witnessed exponential growth in recent years, with 3D games gaining immense popularity due to their immersive visuals, interactive mechanics, and engaging environments. For aspiring developers, understanding the complete lifecycle of a game project—from asset creation to implementation—is essential for building foundational skills in game development. Arsenal 3D is a first-person shooter (FPS) game created as part of an academic project to explore and apply the principles of modern 3D game design. Developed using Unity and enhanced with 3D models crafted in Blender, the game provides a demonstration of teamwork, creativity, and technical execution in a real-time gaming environment.

1.1 Problem Definition:

The primary objective of this project was to develop a playable and engaging 3D FPS game that demonstrates essential gameplay elements such as enemy AI, player health systems, shooting mechanics, and level progression. One of the challenges we aimed to address was integrating custom-designed 3D models into Unity while maintaining smooth gameplay and optimized performance across different devices. Another key problem was designing varied environments that could reflect different difficulty levels and maintain player interest. Additionally, as first-time developers, we had to overcome learning curves related to Unity scripting, Blender modeling, and asset management. This project not only served as a learning experience but also helped us understand the collaborative dynamics of game development.

Project Overview:

Project Name: Arsenal 3D

Project Description: *Arsenal 3D* is a first-person shooter (FPS) game developed using the Unity game engine, featuring custom-designed 3D assets created in Blender. The game is set in a sci-fi themed universe where players must navigate hostile environments and combat alien-like monsters using a stylized energy weapon. The gameplay experience includes health tracking, enemy AI, dynamic backgrounds, and immersive sound design. The game is designed to offer engaging combat mechanics within two unique levels—one set in a rocky desert terrain and another in an icy, futuristic zone. Each level introduces distinct enemies and gameplay dynamics, providing variety and gradual difficulty progression.

Project Goals:

- To design and implement a fully functional 3D FPS game using Unity.
- To gain practical experience in integrating Blender-created models into a real-time game environment.
- To apply foundational game development concepts such as character movement, shooting mechanics, level design, and basic enemy AI.
- To understand collaborative project development through task division and coordinated effort.
- To explore UI design, background music integration, and player feedback mechanisms for enhanced gameplay experience.

- To build a portfolio-ready project that reflects creativity, technical skills, and design thinking.

Project Technologies:

- Unity game engine
- C# programming language
- Graphic design tools for asset creation

1.2 Hardware Specifications:

- CPU – Core i5 10Gen /Ryzen 5 or above
- RAM – 8Gb or above
- ROM - 4GB or above

1.3 Software Specifications:

- Unity game engine
- VS code
- Networking Library
- Windows Edition - 10/11
- OS build Version: 19043.1526
- Windows Feature Experience Pack 120.2212.4170.0


C#

CHAPTER -2 LITERATURE REVIEW

2.1 Existing System:

The development of 3D first-person shooter (FPS) games has significantly evolved with the advancement of modern game engines and design tools. Industry-standard platforms such as Unity, Unreal Engine, and CryEngine have enabled developers to create realistic and immersive environments with minimal hardware limitations. Among these, Unity has gained substantial popularity among indie and student developers due to its user-friendly interface, real-time rendering capabilities, and seamless integration with external tools like Blender. Unity's compatibility with the C# programming language allows for precise control over game mechanics, AI behaviors, and interaction systems.

Game development today follows structured methodologies such as agile development and iterative prototyping, which promote continuous feedback and gradual improvement. Tools like Unity Profiler and Unity Test Framework help in identifying bugs, optimizing performance, and maintaining gameplay consistency across different levels. For asset creation, software such as Blender is extensively used to design 3D models, textures, and animations, which can then be imported into Unity to build interactive game environments. The combination of these technologies enables the creation of high-quality FPS experiences that are both engaging and technically sound.

Core Mechanics and Gameplay

In FPS games like *Arsenal 3D*, the gameplay revolves around player movement, environmental navigation, and combat dynamics. Players typically control the game from a first-person perspective, allowing them to aim and shoot with precision while exploring different areas. The game includes features such as enemy detection, damage systems, health tracking, and a minimal but informative heads-up display (HUD). Each level presents unique visual settings, such as icy terrains and rocky landscapes, which influence gameplay strategy and player immersion.

The game also incorporates sound design and background music to intensify the atmosphere and provide audio cues during combat. Enemy AI is programmed to respond to player actions, adding challenge and unpredictability to the gameplay. These elements collectively create a dynamic and replayable experience, allowing players to engage with both the environment and the in-game enemies in a meaningful way.

Impact and Social Implications

3D FPS games like *Arsenal 3D* hold significant potential in both educational and entertainment domains. They offer an engaging platform to understand programming logic, 3D design, and real-time system interaction, making them valuable learning tools for students and early-stage developers. Game development projects also encourage collaborative learning and enhance soft skills such as teamwork, project management, and creative problem-solving.

From a social perspective, FPS games can foster communities of players who share gameplay tips, discuss strategies, and provide feedback, creating a sense of connection and collaboration. However, it is important to consider the potential downsides, such as excessive screen time or exposure to intense content, and to approach game development with a sense of responsibility and ethical awareness.

Games like *Arsenal 3D* can also promote inclusivity by featuring diverse environments, characters, and narrative elements that resonate with a broad audience. When designed thoughtfully, such games contribute positively to the gaming landscape, encouraging innovation and responsible content creation in the broader tech and creative industry.

2.2 Proposed System:

The proposed system for developing a 2D Unity Platformer Game includes three main components: the game engine, scripting and asset creation tools, and testing and optimization framework.

1. **Game Engine:**
    - **Unity Engine:** Utilizes Unity's robust features like a visual editor, physics engine, and animation tools for 2D game development.

2. **Scripting and Asset Creation Tools:**
    - **C# Programming Language:** For scripting game logic and mechanics.
    - **Graphic Design Tools:** Adobe Photoshop, Illustrator, and Aseprite for creating 2D sprites and visual assets.
    - **Audio Tools:** Audacity for sound effects and music editing.

3. **Testing and Optimization Framework:**
    - **Playtesting:** To gather feedback and improve gameplay.
    - **Automated Testing:** Using Unity Test Framework to ensure functionality.
    - **Performance Optimization:** Employing Unity Profiler for smooth performance across devices.

Core Gameplay Mechanics

The 2D Unity Platformer Game features several core gameplay mechanics:

Character Movement and Controls: Players can control the character's movements, such as running, jumping, and crouching. The controls should be intuitive and responsive to ensure an engaging gameplay experience.

Platforming Challenges: The game includes various obstacles and platforms that players must navigate. These could involve moving platforms, spikes, and other hazards that require precise timing and skill.

Enemies and Combat: The game features different types of enemies that players must defeat or avoid. Players can use various abilities or weapons to overcome these adversaries.

Health and Lives: Players have a health system, where taking damage from enemies or hazards reduces their health. Health packs or similar items can restore health, and losing all health results in losing a life or restarting the level.

Additional Features

To enhance the gameplay experience, the 2D Unity Platformer Game can incorporate additional features:

Character Customization: Players can personalize their characters' appearance, giving them a sense of ownership and individuality.

Variety in Environments: The game features different environments, such as forests, caves, and urban settings, providing a diverse gameplay experience.

Regular Updates: The game should receive regular updates to introduce new levels, features, and content, as well as to balance gameplay and address any issues.

CHAPTER - 3

DESIGN FLOW/PROCESS

Design Flow and Process:

Phase 1: Concept and Requirements Gathering

- Identify the target audience and their preferences for platformer games.
- Define the core gameplay mechanics and objectives.
- Outline the game world design, levels, and environments.
- Specify characters, enemies, and collectible items.
- Determine key features and unique selling points of the game.

Phase 2: Game Design and Prototyping

- Create detailed game design documents and specifications.
- Develop a prototype to test core mechanics and gameplay.
- Gather feedback from playtesters and refine the game design.
- Create concept art and visual assets for the game world.

Phase 3: Development and Implementation

- Set up the development environment and select the appropriate tools and technologies.
- Develop the game world, levels, and environments in Unity.
- Implement player movement, interactions, and combat mechanics using C#.
- Integrate game assets, including graphics and audio.

Phase 4: Testing and Iteration

- Conduct rigorous testing to identify and fix bugs.
- Gather feedback from playtesters and refine gameplay balance.
- Optimize game performance across various devices.
- Address compatibility issues to ensure a smooth experience.

Phase 5: Deployment and Maintenance

- Deploy the game to target platforms (PC, mobile devices).
- Monitor player feedback and address any post-launch issues.
- Release regular updates to introduce new content, fix bugs, and improve performance.
- Maintain the game's online features and ensure a smooth multiplayer experience, if applicable.

By following this design flow and process, developers can ensure that the 2D Unity Platformer Game is well-designed, engaging, and meets the needs of its target audience.

Here are some additional details on each phase of the design flow and process:

Phase 1: Concept and Requirements Gathering This phase focuses on understanding player preferences and market needs. Developers should conduct market research to identify the target audience and gather feedback from potential players. This input will help define the game's mechanics, objectives, and overall design.

Phase 2: Game Design and Prototyping With a clear understanding of the requirements, developers create detailed game design documents outlining features, mechanics, and systems. A prototype is developed to test core mechanics and gameplay, allowing early identification of any issues. Feedback from playtesting is used to refine the game design.

Phase 3: Development and Implementation This phase involves setting up the development environment and selecting appropriate tools and technologies. Developers build the game world and levels in Unity, implement player movement and interactions using C#, and integrate graphical and audio assets. The core gameplay mechanics are brought to life during this phase.

Phase 4: Testing and Iteration Continuous testing is essential to identify and fix bugs throughout development. Developers gather feedback from playtesters to balance gameplay and optimize performance across various devices. Compatibility issues are addressed to ensure smooth functionality.

Phase 5: Deployment and Maintenance Upon completion, the game is deployed to target platforms. Developers monitor player feedback and resolve post-launch issues. Regular updates are released to introduce new content, fix bugs, and improve performance. Ongoing maintenance ensures the game remains engaging and functional.

Why Use Unity?

Unity is an excellent choice for developing a 2D platformer game due to several advantages:

Cross-Platform Development: Unity enables developers to create games that run on multiple platforms, including PC, consoles, and mobile devices, ensuring a wide reach and accessibility.

Powerful Game Engine: Unity offers a comprehensive set of tools and features for building games, including a robust physics engine, animation system, and a versatile editor, which simplifies the creation of complex and engaging gameplay experiences.

Large Community and Resources: Unity has a vibrant community of developers who share resources, provide support, and create plugins and extensions. This community support makes it easier for developers to overcome challenges and accelerate their development process.

Free to Use: Unity offers a free version that is accessible to most developers, especially those who are just starting out or have limited resources. This makes it a cost-effective option for game development.

Versatility: Unity is highly versatile and can be used to create a wide variety of games. Its flexibility makes it suitable for developing 2D platformer games as well as other game genres.

Here are some specific examples of successful 2D platformer games developed using Unity:

Hollow Knight: This acclaimed game was developed using Unity and features stunning visuals and engaging gameplay, garnering a large fanbase.

Cuphead: Known for its unique hand-drawn animation style and challenging gameplay, Cuphead was also developed using Unity.

Ori and the Blind Forest: This beautiful and emotive platformer game was created with Unity, showcasing the engine’s capability to deliver high-quality gaming experiences.

These examples illustrate Unity's effectiveness in developing successful 2D platformer games. Its popularity and features make it a strong choice for developers looking to create their own engaging and innovative 2D platformer games.battle royale game.

More Information on Developing a 2D Platformer Game Using Unity:

Familiarize Yourself with Unity's Basics: Before delving into the specific tools required for a 2D platformer game, it's essential to understand Unity's core functionality. This includes mastering the interface, asset management, scripting, and the various components that make up a Unity project.

Master the Game Physics: A fundamental aspect of platformer games is precise and responsive physics. Unity provides built-in physics engines that handle gravity, collisions, and other physical interactions. Familiarize yourself with these tools to ensure smooth and realistic player movement and interactions with the game world.

Design Levels and Environments: Level design is critical in platformer games. Use Unity's tilemaps and tools to create engaging and challenging levels. Focus on creating diverse environments with varying degrees of difficulty to keep players engaged.

Implement Player Movement and Controls: Player movement is a core mechanic in platformer games. Unity offers a range of components and scripts to create smooth and responsive movement mechanics. This includes running, jumping, crouching, and other actions that define the gameplay experience.

Create and Animate Characters: Character design and animation play a significant role in the game's appeal. Use tools like Unity’s Animator and Sprite Renderer to bring characters to life with fluid animations and detailed visuals.

Add Enemies and Obstacles: Populate your levels with various enemies and obstacles to challenge players. Implement different behaviors and patterns for enemies to keep the gameplay dynamic and engaging.

Integrate Collectibles and Power-Ups: Enhance the gameplay by adding collectibles and power-ups. These items can provide temporary boosts, health regeneration, or other enhancements that add depth to the game.

Test and Iterate: Continuous testing is crucial for identifying and fixing bugs. Gather feedback from playtesters to refine the gameplay balance and ensure a smooth and enjoyable player experience.

Design and Integrate Enemies and Obstacles: 2D platformer games feature a variety of enemies and obstacles that players must navigate and overcome. Unity provides tools to create and animate 2D sprites, apply particle effects, and manage in-game interactions. This ensures an engaging and challenging gameplay experience.

Implement Level Progression and Challenges: A core element of platformer games is the progression through levels with increasing difficulty. Unity's tools allow for dynamic level design and the creation of interactive elements that make each level unique. These can include moving platforms, puzzles, and environmental hazards that test the player's skills.

Character Customization: Many platformer games offer customization options for characters. Unity provides tools for creating and managing 2D sprites, textures, and user interface elements to enable players to personalize their characters' appearance, enhancing player engagement and satisfaction.

Add Collectibles and Power-Ups: Collectibles and power-ups add depth to the gameplay. Unity's scripting capabilities allow developers to implement these items seamlessly, providing players with temporary boosts, health regeneration, and special abilities that enrich the gameplay experience.

Design and Create Diverse Environments: Diverse environments enhance the visual appeal and variety of gameplay. Unity's tilemaps and asset store offer a plethora of resources for creating different settings, from lush forests to dark caves, each with unique challenges and aesthetics.

Implement Regular Updates and Content: To keep the game fresh and players engaged, regular updates are essential. Unity's asset management and scripting tools facilitate the introduction of new levels, characters, and features, ensuring the game remains exciting and up-to-date.

Unity's Layered Architecture

The Unity game engine utilizes a layered architecture that provides a modular and flexible

framework for developing interactive applications, divided into three main layers:

Engine Layer The engine layer forms the core of Unity, providing essential functions such as rendering graphics, handling input, and managing physics. Written in native code (C/C++), it interacts directly with the hardware to ensure optimal performance.

C# Scripting Layer This layer allows developers to create custom logic and behaviors for game objects. Unity offers a comprehensive set of APIs and libraries accessible through C# scripts, enabling the implementation of game mechanics, user interactions, and other dynamic aspects.

Assets Layer Encompassing all visual and audio elements, the assets layer includes 2D sprites, textures, sounds, and animations. Unity provides tools and workflows for creating, managing, and importing these assets into the game.

Interplay of Layers These layers work together seamlessly to create a cohesive game development environment. The engine layer provides the foundation, the C# scripting layer offers control over game logic, and the assets layer brings the game world to life.

Key Components of Unity Architecture

Game Objects: Fundamental building blocks representing individual entities within the game, with components like transforms, scripts, and renderers attached.

Components: Modular units that define the behavior and properties of game objects, handling tasks like rendering, physics, networking, and user input.

Scenes: Individual levels or environments within the game containing game objects, components, and lighting.

MonoBehaviour: A base class for C# scripts, allowing developers to attach scripts to game objects and implement event-driven callbacks.

Physics Engine: Handles realistic collisions and interactions, simulating gravity, motion, and other physical forces.

Rendering Engine: Converts game objects and assets into visual output, managing lighting, shadows, and post-processing effects.

Audio Engine: Manages the playback and mixing of sound effects and music, supporting various audio formats.

Benefits of Unity Architecture

Modular and Flexible: Allows developers to customize and extend the engine's functionality with their own scripts and components.

Cross-Platform Development: Supports a wide range of platforms, including PC, consoles, mobile devices, and web browsers.

Large Community and Resources: Offers extensive support and resources through a vast developer community.

Free to Use: Provides a free version with comprehensive features, making it accessible to hobbyists and professional developers alike.

What is C# ?

C# is a general-purpose, object-oriented programming language developed by Microsoft in the early 2000s. It is a versatile language that can be used to develop a wide variety of applications, including games, web applications, desktop applications, and scientific computing software.

Key Features of C#

C# is a powerful and expressive language that offers a number of features that make it well-suited for game development:

Garbage Collection: C# has built-in garbage collection, which automatically manages the memory allocation and deallocation of objects. This frees developers from having to manually manage memory, which can lead to memory leaks and other errors.

Strong Typing: C# is a strongly typed language, which means that variables have a specific data type. This helps to prevent errors and makes code more readable and maintainable.

Object-Oriented Programming: C# is based on object-oriented programming (OOP) principles, which allow developers to create modular and reusable code. OOP is well-suited for game development, as it can be used to model complex game objects and systems.

Exception Handling: C# has a powerful exception handling mechanism that helps to catch and handle errors in the program. This is essential for game development, as errors can cause unexpected behavior and crashes.

Applications of C#

C# is a versatile language that can be used to develop a wide variety of applications, including:

Games: C# is a popular language for developing AAA and indie games. It is used to develop games for a variety of platforms, including PC, consoles, and mobile devices.

Web Applications: C# is a popular choice for developing web applications, as it can be used to build server-side applications and client-side applications.

Desktop Applications: C# is a good choice for developing desktop applications, as it can be used to create cross-platform applications that run on Windows, macOS, and Linux.

Scientific Computing Software: C# can be used to develop scientific computing software, as it has a number of built-in libraries for scientific and mathematical calculations.

Learning C#

There are a number of resources available for learning C#, including online tutorials, books, and courses. C# is a relatively easy language to learn, especially for developers with experience in other programming languages.

Game Structure:

The structure of a 2D Unity Platformer Game typically involves three main components: the game engine, the game client, and the asset management system.

Game Engine

The game engine acts as the core framework for creating and managing the game world and mechanics. It handles rendering graphics, processing physics, and managing game logic. Unity serves as the engine, providing a robust set of tools for these tasks.

Key responsibilities of the game engine include:

Rendering graphics: The engine processes and displays 2D sprites, animations, and visual effects.

Managing physics: Handles collisions, gravity, and other physical interactions within the game world.

Running game logic: Executes scripts and manages game mechanics, such as player movements and interactions.

Game Client

The game client is the software running on each player's device. It renders the game world, handles player input, and updates the game state based on interactions within the game.

Key responsibilities of the game client include:

Rendering the game world: Utilizes the GPU to display 2D sprites, textures, and visual effects.

Handling player input: Captures input from devices like keyboards, controllers, or touchscreens and processes it within the game.

Updating the game state: Manages in-game interactions and updates the local representation of the game world based on player actions and events.

Asset Management System

The asset management system encompasses all visual and audio elements that make up the game world, including sprites, textures, sounds, and animations. Unity provides tools and workflows for creating, managing, and importing these assets.

Key aspects of the asset management system include:

Asset creation: Developing and importing 2D assets using tools like Photoshop, Illustrator, and Audacity.

Asset management: Organizing and optimizing assets within Unity to ensure efficient use and easy access.

Resource loading: Managing how and when assets are loaded into the game to ensure smooth performance and gameplay experience.

Additional Components

In addition to the core components, a 2D Unity Platformer Game may also include other elements:

Character Customization: Players may have the option to customize their characters' appearance, giving them a sense of individuality and ownership.

Varied Environments: The game world should feature a variety of environments, such as forests, caves, and urban settings, to provide diverse gameplay experiences.

Power-Ups and Collectibles: The game can include various power-ups and collectibles that enhance gameplay by providing temporary boosts or special abilities.

Regular Updates and Content: The game should receive regular updates to introduce new levels, characters, and features, balance gameplay, and address any issues.

Popular Tools and Resources for 2D Game Development:

Adobe Photoshop and Illustrator:

- **Collaboration:** These tools are widely used for creating and sharing visual assets. Users can easily collaborate by sharing project files.
- **Mobile App Support:** Adobe offers mobile versions of these tools, allowing users to work on their projects on the go.
- **Strengths:** Known for their powerful features and versatility in creating high-quality graphics and textures.
- **Weaknesses:** They can be expensive and have a steep learning curve for beginners.

Aseprite:

- **Collaboration:** Aseprite allows users to create pixel art and animations, which can be easily shared with team members.

- **Mobile App Support:** While primarily desktop-based, its files can be viewed and shared on mobile devices.
- **Strengths:** Specializes in pixel art and animation, making it a favorite among 2D game developers.
- **Weaknesses:** Limited to pixel art, so it may not be suitable for all types of graphics.

Audacity:

- **Collaboration:** Audacity is an open-source audio editing tool that allows users to share and collaborate on audio files.
- **Mobile App Support:** While Audacity itself is desktop-based, audio files can be shared and accessed on mobile devices.
- **Strengths:** Free to use, with a wide range of features for editing and mixing audio.
- **Weaknesses:** Interface can be a bit dated and may require additional plugins for certain features.

Unity Asset Store:

- **Collaboration:** Developers can purchase and share assets from the Unity Asset Store, providing a vast library of ready-made assets.
- **Mobile App Support:** Assets can be accessed and managed through Unity's mobile interface.
- **Strengths:** Offers a wide variety of assets, plugins, and tools to enhance game development.
- **Weaknesses:** Quality can vary, and some assets may require additional customization.

POPULAR 3D FIRST PERSON SHOOTER GAMES

History of 3D First-Person Shooter (FPS) Games

The origins of 3D first-person shooter (FPS) games can be traced back to the early 1990s when titles like *Wolfenstein 3D* (1992) introduced players to a groundbreaking perspective that placed them directly in the eyes of the protagonist. Developed by id Software, *Wolfenstein 3D* established the basic framework for FPS gameplay, including shooting mechanics, maze-like levels, and enemy encounters—all viewed from a first-person point of view.

Shortly after, the release of *DOOM* in 1993 marked a pivotal moment in the history of gaming. With improved graphics, faster gameplay, and multiplayer features, *DOOM* laid the foundation for the modern FPS genre. Its success spurred the development of other popular titles such as *Quake* (1996), which further advanced 3D graphics and introduced online multiplayer, a feature that would become essential to the FPS experience.

The late 1990s and early 2000s witnessed significant technological advancements that reshaped the FPS landscape. Games like *Half-Life*, *Halo: Combat Evolved*, and *Counter-Strike* pushed the genre forward by incorporating narrative depth, improved AI, and team-based mechanics. These games not only refined shooting mechanics but also began to emphasize immersive storytelling and strategic gameplay.

As hardware capabilities grew, so did the complexity and realism of FPS games. The introduction of physics engines, dynamic lighting, and open-world design allowed developers to create richer and more interactive environments. Titles like *Call of Duty*, *Battlefield*, and *Far Cry* brought cinematic elements to the genre, blurring the lines between storytelling and action.

Key Factors Driving the Popularity of 3D FPS Games

The lasting success of 3D first-person shooter (FPS) games, particularly those developed in engines like Unity, can be attributed to several fundamental factors:

Dynamic and Skill-Based Gameplay:
FPS games offer a thrilling blend of precision aiming, quick reflexes, and strategic movement. The intensity of combat and the need for rapid decision-making create deeply engaging and satisfying gameplay experiences.

Immersion and Perspective:
By placing players directly into the eyes of the character, 3D FPS games deliver unparalleled immersion. This perspective strengthens player connection to the game world, enhancing emotional investment and replayability.

Diverse Maps and Replay Value:
Varied level designs—from open battlefields to tight urban environments—combined with dynamic enemy behavior, keep gameplay fresh. Procedural generation, custom mods, and multiplayer maps ensure that no two matches feel identical.

Accessibility and Wide Audience Appeal:
Modern FPS titles offer scalable difficulty, control customization, and cross-platform support, making them accessible to both new players and seasoned veterans. Unity's flexible development environment has further expanded accessibility by enabling smaller studios to create polished FPS experiences.

Innovation through Indie Development:
Independent developers using engines like Unity have introduced bold new ideas into the FPS genre, experimenting with aesthetics, movement mechanics, and narrative techniques that challenge traditional designs and keep the genre vibrant.

Evolution and Future Directions

The 3D FPS genre continues to evolve rapidly, fueled by technological innovation, genre fusion, and changing player expectations.

Genre Expansion and Hybrids:
Modern FPS games increasingly blend mechanics from role-playing games (RPGs), survival games, and adventure genres. Skill trees, crafting systems, and open-world exploration are now common, expanding the scope and depth of shooter experiences.

Rise of Mobile and Cloud FPS Gaming:
Mobile platforms and cloud gaming services have dramatically broadened the FPS audience. Unity's cross-platform capabilities allow developers to adapt full-scale FPS titles for mobile devices without sacrificing gameplay quality, reaching players wherever they are.

Emerging Technologies: VR, AR, and AI Integration:
Virtual Reality (VR) is revolutionizing how players experience FPS games, offering true 360-degree immersion and intuitive motion controls. Augmented Reality (AR) is also opening new possibilities for blending physical and digital gameplay. Meanwhile, advanced artificial intelligence (AI) is being used to create smarter, adaptive enemies, dynamic environments, and personalized experiences that evolve based on player behavior.

Future Directions and Potential Trends in 3D FPS Games

The future of 3D first-person shooter (FPS) games is poised to be shaped by several transformative trends, driven by technological innovation and evolving player expectations:

- **Cross-Platform Compatibility:**
  Future FPS titles will focus heavily on seamless integration across PC, consoles, and mobile devices, allowing players to maintain progress and enjoy multiplayer experiences regardless of the platform they choose.
- **Cloud Gaming Expansion:**
  Cloud-based gaming will enable developers to deliver richer, more visually complex FPS experiences without requiring high-end hardware. Players will be able to access graphically intensive shooters instantly on a variety of devices.
- **Enhanced Social Features and Community Building:**
  Upcoming FPS games are expected to deepen social integration, offering new ways for

players to team up, form communities, share achievements, and participate in live events or tournaments.

- **Mobile-First Development Strategies:**
  As mobile gaming continues to grow, developers will prioritize optimizing FPS mechanics for mobile platforms, ensuring intuitive controls, scalable graphics, and cross-play capabilities to reach a wider, global audience.
- **Narrative Depth and Emotional Storytelling:**
  Integrating deeper narrative elements into FPS campaigns will create more emotionally resonant experiences. Story-driven missions, character arcs, and decision-based outcomes will add new dimensions to the traditionally action-focused genre.

As the 3D FPS landscape continues to evolve, its place at the forefront of gaming culture remains secure. The creativity and adaptability of developers—especially those leveraging engines like Unity—will be key to pushing the boundaries of what FPS games can deliver, ensuring that the genre remains vibrant, engaging, and innovative for years to come.

BASIC FLOWCHART:

Start Game

- Load the game
- Display main menu
- Select level or continue from saved progress

Pre-Game Setup

- Select character (if applicable)
- Customize character appearance
- Equip any starting items or abilities (if applicable)

Enter Level

- Load the selected level
- Display level introduction or storyline (if applicable)

Gameplay Begins

- Control character movements (run, jump, crouch)
- Navigate through platforms and obstacles
- Collect items and power-ups

Encounter Enemies and Obstacles

- Defeat enemies using attacks or abilities
- Avoid or overcome environmental hazards (spikes, moving platforms)

Progress Through Level

- Continue navigating the level, solving puzzles, and overcoming challenges
- Use collected items and power-ups to aid progression

Boss Battle (if applicable)

- Engage in boss battles at the end of certain levels
- Defeat the boss to progress to the next level

Complete Level

- Reach the end of the level
- Display level completion screen with stats (time, items collected, score)

Post-Level

- Save progress
- Return to main menu or select next level

Repeat Until Game Completion

- Continue through successive levels until the final level is completed
- Complete the game storyline or objectives

Core Elements of 3D FPS Games

3D first-person shooter (FPS) games stand out as one of the most engaging and immersive genres in modern gaming. *Arsenal 3D* embraces the core principles of this genre while incorporating unique elements to enhance the gameplay experience. Below are the defining features that structure the game's design and interaction:

First-Person Perspective and Movement

The game is played entirely through the eyes of the player character, allowing for a direct and immersive view of the environment. Smooth movement controls—including forward navigation, strafing, and camera-based aiming—are essential to navigating combat zones and reacting to threats.

Combat and Shooting Mechanics

At the heart of *Arsenal 3D* is real-time combat. Players use a stylized weapon with unique projectile visuals to engage enemies. Precision aiming, ammo management, and quick reflexes are vital as enemies respond with their own attacks and unique behaviors.

Level Design and Environmental Variation

The game features two distinct levels with contrasting themes: an icy battlefield and a mountainous terrain. These areas are designed to test the player's mobility and adaptability. Each level introduces different visual cues, enemy spawn points, and pathways, making exploration a core element of progression.

Enemy AI and Challenge Scaling

Enemies in *Arsenal 3D* are not static targets—they engage the player with unique weapons and movement patterns. AI behavior adapts to the player’s approach, offering a balance between predictability and surprise. Boss enemies like the “Hive Queen” introduce spike challenges to test player endurance.

Health System and HUD

The game includes a dynamic health bar visible on the screen, helping players monitor damage and strategize survival. The heads-up display (HUD) also includes weapon indicators and minimal UI elements to keep the screen clear and focused on gameplay.

Audio and Visual Feedback

Sound design plays an important role in immersion—background music intensifies the atmosphere, and sound cues help indicate enemy presence or low health. Visual effects, such as screen flashes or color shifts, reinforce gameplay events like taking damage or switching weapons.

Popular 3D FPS Titles

The first-person shooter genre has produced some of the most influential and beloved titles in gaming history, each contributing unique elements that have helped shape the evolution of FPS mechanics:

DOOM (1993):

A pioneering FPS title known for its fast-paced action, maze-like levels, and groundbreaking 3D graphics. *DOOM* established many of the core mechanics that define the genre to this day.

Half-Life (1998):

Blending first-person shooting with narrative depth, *Half-Life* introduced seamless storytelling through in-game events, setting a new standard for immersive gameplay and intelligent enemy AI.

Halo: Combat Evolved (2001):

A revolutionary console FPS that combined expansive science-fiction settings, fluid shooting mechanics, and vehicular combat. *Halo* helped popularize multiplayer FPS gaming and influenced countless later titles.

Call of Duty 4: Modern Warfare (2007):

Renowned for its cinematic campaign and refined multiplayer experience, *Modern Warfare* set a benchmark for modern FPS design with its progression systems, loadout customization, and tight shooting mechanics.

Overwatch (2016):

A team-based FPS that introduced colorful heroes with distinct abilities, promoting strategic teamplay and accessibility. *Overwatch* blended traditional shooting mechanics with class-based combat, bringing a fresh twist to the genre.

Impact on Gaming Culture

3D first-person shooter (FPS) games have played a transformative role in shaping gaming culture, not only through their mechanics and storytelling but also through their influence on community engagement, media trends, and game development practices. Here are some of the most significant impacts of the genre:

Legacy and Genre Influence:
Iconic FPS titles such as *DOOM*, *Half-Life*, and *Call of Duty* have left a lasting imprint on the gaming industry. Their design principles have influenced the development of countless other genres, from survival horror to RPGs, establishing FPS games as a foundational pillar of modern game design.

Streaming and Content Creation:
3D FPS games are a dominant force in the world of online streaming and video content. Their fast-paced gameplay, competitive modes, and replayability make them ideal for live broadcasts, walkthroughs, and esports commentary. Games like *Valorant* and *Overwatch* have become central to Twitch and YouTube gaming culture.

Online Communities and Competitive Play:
FPS games often foster strong online communities built around teamwork, competition, and shared goals. Whether through ranked matches, clan systems, or custom lobbies, players form social bonds and rivalries that drive long-term engagement. Fan-made mods, skins, and maps also contribute to the genre's cultural footprint.

Indie FPS Development:
With the rise of accessible engines like Unity and Unreal, indie developers have begun crafting unique and experimental FPS experiences. This has led to a wave of innovation within the genre, allowing smaller teams to explore storytelling, aesthetic styles, and mechanics outside the mainstream formula—much like *Arsenal 3D*.

Educational and Technical Learning Tools:
Due to their complex yet structured design, FPS games are increasingly used in educational contexts to teach programming logic, 3D design, level structuring, and AI behavior. Game development courses often use FPS mechanics as a foundation for student projects, providing hands-on learning.

Potential Future and Innovations in 3D FPS Games

The future of 3D first-person shooter (FPS) games is filled with opportunities for technological advancements and creative innovation:

Cross-Platform Compatibility:
Developers are increasingly focusing on enabling seamless gameplay across multiple platforms, including PC, consoles, and mobile devices. Cross-platform functionality would allow players to enjoy FPS titles together, regardless of their device, strengthening global gaming communities.

Cloud Gaming and Remote Access:
The rise of cloud gaming technology holds the potential to transform FPS gaming experiences. Players will be able to enjoy graphically rich, complex shooter games without needing high-end hardware, making the genre more accessible to a broader audience.

Deeper Storytelling and Narrative Complexity:
Future FPS games are expected to integrate richer narratives, emotional depth, and character development alongside traditional combat mechanics. Story-driven missions, branching paths, and moral choices could add new layers of immersion.

AI-Enhanced Gameplay and Dynamic Environments:
Artificial intelligence will increasingly play a role in creating smarter enemies, procedurally generated maps, and adaptive gameplay that responds to player behavior. This would create unpredictable challenges and enhance replayability.

Metaverse and Virtual Worlds Integration:
FPS games could find a place within emerging metaverse ecosystems, offering persistent virtual environments where players can not only engage in battles but also socialize, trade, and participate in large-scale virtual events.

Psychological Factors Contributing to FPS Games' Success

Several psychological factors explain why 3D FPS games continue to captivate and retain large audiences:

Thrill of Action and Skill Mastery:
The high-stakes, fast-paced nature of FPS combat provides a constant adrenaline rush. Mastering aiming, movement, and tactical strategies offers a strong sense of achievement and keeps players

engaged.

Sense of Agency and Control:
FPS games offer players direct control over their actions and decisions in a 3D space. Successfully overcoming challenges and defeating enemies builds a powerful feeling of empowerment.

Social Connectivity and Competitive Spirit:
Multiplayer FPS games foster vibrant communities where teamwork, competition, and rivalry drive deep emotional investment. Esports events, ranked modes, and online leaderboards offer additional layers of social interaction.

Replayability through Variety:
Diverse maps, weapons, enemy types, and multiplayer modes ensure that no two gameplay sessions feel exactly the same, encouraging players to return for repeated experiences.

Accessibility and Broad Appeal:
Modern FPS games offer scalable difficulty settings, customizable controls, and various playstyles, making the genre approachable for both casual players and hardcore enthusiasts.

Additional Factors Enhancing 3D FPS Games' Appeal

Beyond psychological appeal, several additional factors have contributed to the widespread popularity and continued growth of 3D first-person shooter (FPS) games, particularly those developed using accessible engines like Unity:

Esports Growth and Competitive Leagues:
The inherently competitive nature of FPS gameplay has led to the creation of global esports tournaments and professional leagues. Titles like *Valorant* and *Counter-Strike* have set benchmarks, demonstrating how skill-based shooter mechanics can captivate massive audiences and foster vibrant competitive scenes.

Streaming and Content Creation:
FPS games consistently dominate platforms like Twitch and YouTube due to their dynamic gameplay, unpredictability, and highlight-worthy moments. Streamers often showcase Unity-developed FPS games, bringing exposure to indie projects and helping them build loyal fanbases.

Frequent Updates and Seasonal Content:
Successful FPS titles maintain player interest through regular updates, including new maps,

weapons, characters, and game modes. These updates, often seen in Unity-powered games, ensure that the community remains engaged and the game environment stays dynamic.

Collaborations and Crossover Events:

Partnerships with popular brands, franchises, and even other games introduce exclusive content such as skins, missions, and limited-time events. These collaborations not only generate excitement among existing players but also attract new audiences curious about the crossover experiences.

Mobile Adaptation and Broader Accessibility:

The increasing power of mobile devices and Unity's versatile cross-platform development capabilities have enabled many FPS games to be adapted for smartphones and tablets. Mobile-friendly shooters offer flexible access to gameplay, allowing users to experience competitive 3D FPS action on the go and greatly expanding the genre’s global reach

Impact on Society

3D first-person shooter (FPS) games, particularly those developed using engines like Unity, have made a significant impact on society, influencing cultural trends, education, and social behavior while also raising important discussions around responsible gaming.

Cultural Influence:
FPS games have become a major part of global entertainment culture. Franchises like *Call of Duty*, *Halo*, and various Unity-based indie shooters have shaped how video games are perceived—as serious competitive sports, storytelling mediums, and forms of creative expression.

Social Dynamics and Community Building:
Multiplayer FPS games have fostered a strong sense of community among players. Cooperative missions, team-based matches, and online tournaments have encouraged teamwork, communication, and long-term friendships. Unity-developed FPS games often empower smaller communities through custom servers, mods, and community events.

Educational Applications:
Beyond entertainment, FPS games have found use in educational and training environments. Concepts such as spatial awareness, hand-eye coordination, and even basic programming (through modding or game creation in Unity) are taught using FPS frameworks. Some serious games use shooter mechanics for military, medical, or tactical training simulations.

Mental Health and Well-being:
While FPS games can offer stress relief, improve cognitive reflexes, and serve as a healthy outlet for competition, concerns remain regarding potential downsides. Excessive gaming, exposure to violent content, and the pressure of competitive environments can contribute to mental health challenges if not balanced with real-world responsibilities.

Addiction Concerns and Responsible Gaming

The immersive and competitive nature of 3D first-person shooter (FPS) games, particularly those developed using platforms like Unity, has raised concerns regarding player behavior, well-being, and the importance of promoting healthy gaming habits. Here are some key concerns associated with excessive FPS gameplay:

Immersive Combat and Over-Engagement:

The fast-paced, high-adrenaline gameplay of FPS games can easily lead players to lose track of time. The constant drive to achieve victories, complete missions, or improve rankings can foster dependency and extended gaming sessions.

Escapism and Social Disconnection:
FPS games often provide a strong sense of escape, allowing players to immerse themselves in alternate worlds. While this can be a form of stress relief, excessive reliance on virtual experiences can sometimes lead to neglect of real-world relationships and responsibilities.

Virtual Achievements vs. Real-Life Growth:
While achieving milestones within games can boost self-esteem, an overemphasis on digital success may detract from pursuing real-world personal, academic, or professional achievements.

Emotional Impact and Competitive Frustration:
The competitive environment of FPS games can result in emotional highs and lows. Frequent losses, difficulty progressing, or toxic interactions in online matches can cause frustration, and stress, and even impact mental well-being if not managed carefully.

Responsible Gaming Practices

To ensure that gaming remains a healthy and positive experience, it is important for players, developers, and guardians to encourage responsible habits:

Setting Clear Time Limits:
Players should define specific gaming durations and take regular breaks to maintain a healthy balance between gaming and daily life activities.

Diversifying Gaming Experiences:
Engaging with different types of games and activities can help prevent overdependence on a single FPS title and promote a broader range of skills and interests.

Maintaining Real-World Connections:
Prioritizing offline relationships, social activities, and community involvement is crucial for maintaining emotional well-being alongside gaming hobbies.

Mindful Spending and In-Game Purchases:
Players should approach in-game purchases, particularly for cosmetic or competitive advantages, with caution to avoid overspending and reliance on microtransactions.

CODES (SCRIPTS) -

SCRIPT 1 (playerController.cs)

```
using System;
using UnityEngine;
using UnityEngine.InputSystem;

public class playerController : MonoBehaviour
{
    public float walkspeed = 5f;
    public float runspeed = 10f;
    public float Airwalkspeed = 3f;
    public float JumpImpulse = 10f;
    Vector2 moveInput;
    TouchingDirection direction;
    Damageable Damageable;
    Vector2 knockBackVelocity;

    public float CurrentMoveSpeed
    {
        get
        {
            if (CanMove)
            {
                if (IsMoving && !direction.IsOnWall)
                {
                    if (direction.IsGrounded)
                    {
                        if (IsRunning)
                        {
                            return runspeed;
                        }
                        else
                        {
                            return walkspeed;
                        }
                    }
                    else
                    {
                        return Airwalkspeed;
                    }
                }
                else
                {
                    return 0;
                }
            }
            else
            {
                return 0;
            }

        }
```

```
}


[SerializeField]
private bool _isMoving = false;

public bool IsMoving { get
    {
        return _isMoving;
    }
    private set
    {
        _isMoving = value;
        animator.SetBool(AnimationStrings.IsMoving, value);
    }
}

[SerializeField]
private bool _isRunning = false;

public bool IsRunning
{
    get
    {
        return _isRunning;
    }
    set
    {
        _isRunning = value;
        animator.SetBool(AnimationStrings.IsRunningPressed, value);
    }
}

public bool _isFacingRight = true;
public bool IsFacingRight { get
    {
        return _isFacingRight;
    }

    private set
    {
        if (_isFacingRight != value)
        {
            transform.localScale *= new Vector2(-1,1);
        }


        _isFacingRight = value;
    }
}

public bool CanMove
{
    get
    {
```

```
            return animator.GetBool(AnimationStrings.canMove);
        }
    }

    public bool IsAlive
    {
        get
        {
            return animator.GetBool(AnimationStrings.IsAlive);
        }
    }



    Rigidbody2D rb;
    Animator animator;
    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        direction = GetComponent<TouchingDirection>();
        Damageable = GetComponent<Damageable>();
    }


    private void FixedUpdate()
    {
        if (!Damageable.LockVelocity) //if we are hit, dont let any input affect movement
        {
            rb.linearVelocity = new Vector2(moveInput.x * CurrentMoveSpeed, rb.linearVelocity.y); //dont need to multiply
by Time.fixedDeltaTime because velocity handles that
            animator.SetFloat(AnimationStrings.Yvelocity, rb.linearVelocity.y);

        }
        else
        {
            rb.AddForce(knockBackVelocity * Time.deltaTime); //needs to be here because it's physics
        }
    }

    public void OnMove(InputAction.CallbackContext context)
    {
        moveInput = context.ReadValue<Vector2>();

        if (IsAlive)
        {
            IsMoving = moveInput != Vector2.zero;

            SetFacingDirection(moveInput);
        }
        else
        {
            IsMoving = false;
        }
    }
```

```
    private void SetFacingDirection(Vector2 moveInput)
    {
        if (moveInput.x > 0 && !IsFacingRight)
        {
            IsFacingRight = true;
        }
        else if (moveInput.x < 0 && IsFacingRight)
        {
            IsFacingRight = false;
        }
    }

    public void onRun(InputAction.CallbackContext context)
    {
        if (context.started)
        {
            IsRunning = true;
        }
        else if (context.canceled)
        {
            IsRunning = false;
        }

    }

    public void OnJump(InputAction.CallbackContext context)
    {
        if (context.started && direction.IsGrounded && CanMove)
        {
            animator.SetTrigger(AnimationStrings.jump);
            rb.linearVelocity = new Vector2(rb.linearVelocity.x, JumpImpulse);
        }
    }

    public void OnAttack(InputAction.CallbackContext context)
    {
        if(context.started)
        {
            animator.SetTrigger(AnimationStrings.attack);
        }
    }

    public void OnHit(int damage, Vector2 knockback)
    {
        rb.linearVelocity = new Vector2(knockback.x, rb.linearVelocity.y + knockback.y); //this line basically does
nothing, it does not affect the x direction for some reason
        knockBackVelocity = knockback; //set the new variable, put it in AddForce in FixedUpdate
    }
}
```

SCRIPT 2 (Knight.cs)

```
using System;
```

```
using UnityEngine;

public class Knight : MonoBehaviour
{
    public float walkspeed = 3f;
    public DetectionZone zone;

    Rigidbody2D rb;
    TouchingDirection Direction;
    Animator animator;

    public enum WalkDirection {Right, Left};

    private WalkDirection _walkDirection;
    private Vector2 WalkDirectionVector;

    public WalkDirection WalkDir
    {
        get { return _walkDirection; }
        set {
            if (_walkDirection != value)
            {
                gameObject.transform.localScale = new Vector2(gameObject.transform.localScale.x * -1,
gameObject.transform.localScale.y);

                if (value == WalkDirection.Right)
                {
                    WalkDirectionVector = Vector2.right;
                }else if (value == WalkDirection.Left)
                {
                    WalkDirectionVector = Vector2.left;
                }
            }
            _walkDirection = value; }
    }

    public bool _hasTarget = false;

    public bool HasTarget { get
        {
            return _hasTarget;
        }
        private set
        {
            _hasTarget = value;
            animator.SetBool(AnimationStrings.HasTarget, value);
        }
    }

    public bool CanMove
    {
        get
        {
            return animator.GetBool(AnimationStrings.canMove);
        }
```

```
    }

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        Direction = GetComponent<TouchingDirection>();
        animator = GetComponent<Animator>();
    }

    private void Update()
    {
        HasTarget = zone.detectedColliders.Count > 0;
    }
    void FixedUpdate()
    {
        if(Direction.IsGrounded && Direction.IsOnWall)
        {
            FlipDirection();
        }
        if (CanMove)
        {
            rb.linearVelocity = new Vector2(walkspeed * WalkDirectionVector.x, rb.linearVelocity.y);
        }
        else
        {
            rb.linearVelocity = new Vector2(0, rb.linearVelocity.y);
        }
    }

    private void FlipDirection()
    {
        if (WalkDir == WalkDirection.Right)
        {
            WalkDir = WalkDirection.Left;
        }
        else if (WalkDir == WalkDirection.Left)
        {
            WalkDir = WalkDirection.Right;
        }
        else
        {
            Debug.LogError("Current Walk Direction Not Valid");
        }

    }
}
```

SCRIPT 3 (Damageable.CS)

```
using UnityEngine;
using UnityEngine.Events;
```

```
public class Damageable : MonoBehaviour
{

    public UnityEvent<int, Vector2> damageableHit;
    Animator animator;
    // Start is called once before the first execution of Update after the MonoBehaviour is created
    [SerializeField]
    private int _maxHealth=100;

    public int MaxHealth
    {
        get
        {
            return _maxHealth;
        }
        set
        {
            _maxHealth = value;
        }
    }

    [SerializeField]
    private int _Health = 100;

    public int Health
    {
        get
        {
            return _Health;
        }
        set
        {
            _Health = value;
            if(_Health <= 0)
            {
                IsAlive = false;
            }
        }
    }

    private bool _IsAlive = true;
    private bool IsInvincible = false;


    private float timeSinceHit = 0;
    public float invincibleTime = 0.25f;

    public bool IsAlive { get
        {
            return _IsAlive;
        }
        set
        {
            _IsAlive = value;
```

```
            animator.SetBool(AnimationStrings.IsAlive, value);
        }
    }

    public bool LockVelocity
    {
        get
        {
            return animator.GetBool(AnimationStrings.LockVelocity);
        }
        set
        {
            animator.SetBool(AnimationStrings.LockVelocity, value);
        }
    }

    private void Awake()
    {
        animator = GetComponent<Animator>();

    }

    private void Update()
    {
        if(IsInvincible)
        {
            if(timeSinceHit > invincibleTime)
            {
                IsInvincible=false;
                timeSinceHit = 0;
            }

            timeSinceHit += Time.deltaTime;
        }

    }

    public bool Hit(int damage,Vector2 knockback)
    {
        if (IsAlive && !IsInvincible)
        {
            Health -= damage;
            IsInvincible=true;

            animator.SetTrigger(AnimationStrings.hit);
            LockVelocity = true;

            damageableHit?.Invoke(damage, knockback);

            return true;
        }else
            { return false; }
    }
}
```

SCRIPT 4 (Attack.CS)

```
using UnityEngine;

public class Attack : MonoBehaviour
{
    public int attackdamage = 10;
    public Vector2 knockback = Vector2.zero;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        Damageable damage = collision.GetComponent<Damageable>();

        if (damage != null)
        {
            bool GotHit  = damage.Hit(attackdamage, knockback);

            if (GotHit)
            {
                Debug.Log(collision.name + " hit for " +  attackdamage);
            }
        }
    }
    // Update is called once per frame
    void Update()
    {

    }
}
```

SCRIPT 5 (DetectionZone.CS)

```
using UnityEngine;
using System.Collections.Generic;

public class DetectionZone : MonoBehaviour
{
    public List<Collider2D> detectedColliders = new List<Collider2D>();
    Collider2D col;


    private void Awake()
    {
        col = GetComponent<Collider2D>();
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        detectedColliders.Add(collision);
    }
```

```
    private void OnTriggerExit2D(Collider2D collision)
    {
        detectedColliders.Remove(collision);
    }
}
```

SCRIPT 6 (ParallaxEffect.CS)

```
using UnityEngine;

public class ParallaxEffect : MonoBehaviour
{
    public Camera cam;
    public Transform followTarget;

    Vector2 startingPosition;

    float startingZ;

    Vector2 camMoveSinceStart => (Vector2) cam.transform.position - startingPosition;
    float ZdistanceFromTarget => transform.position.z - followTarget.transform.position.z;
    float clippingPlane => cam.transform.position.z + (ZdistanceFromTarget > 0 ? cam.farClipPlane :
cam.nearClipPlane);
    float parallaxFactor => Mathf.Abs(ZdistanceFromTarget) / clippingPlane;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {
        startingPosition = transform.position;
        startingZ = transform.position.z;
    }

    // Update is called once per frame
    void Update()
    {
        Vector2 newPosition = startingPosition + camMoveSinceStart * parallaxFactor;

        transform.position = new Vector3(newPosition.x,newPosition.y,startingZ);
    }
}
```

CHAPTER – 4

FUTURE SCOPE

Cross-Platform FPS Accessibility

Expanding *Arsenal 3D* across multiple platforms such as PC, console, and mobile would significantly boost its reach and versatility. Cross-platform compatibility would allow users on different systems to engage in the same multiplayer environments, reducing barriers to entry and fostering a broader, interconnected player community. It would also ensure that the game maintains consistency across devices, enhancing the overall user experience and accessibility.

Cloud-Based FPS Deployment

Integrating *Arsenal 3D* into cloud gaming platforms could redefine how players experience high-quality 3D FPS gameplay. With streaming technology, players would be able to enjoy smooth performance and detailed graphics without needing high-end hardware. This shift toward cloud deployment would open the game to a larger audience, including casual gamers and students using low-spec devices.

Immersive Storyline and Lore Development

While *Arsenal 3D* currently focuses on action and environment-based immersion, future versions could incorporate deeper narrative elements, including backstory, cutscenes, and mission-based objectives. Introducing a compelling in-game universe with named characters, factions, and lore-driven quests would strengthen player engagement and promote emotional investment in the gameplay.

Advanced Enemy AI and Adaptive Gameplay

Artificial intelligence can play a transformative role in enhancing *Arsenal 3D*. Future updates may feature smarter enemies that adapt to player behavior, team-based enemy formations, and boss characters that evolve in real time. This would not only increase difficulty but also encourage players to adopt strategic thinking and varied combat styles.

Virtual FPS Arenas and Social Interaction

Future development may explore FPS integration into the metaverse or social gaming hubs. Players could interact in virtual lobbies, customize avatars, or even participate in community missions and tournaments. Adding voice chat, friend systems, and virtual co-op missions would amplify social engagement within the game.

AR/VR Integration for Immersive Combat

Augmented Reality (AR) and Virtual Reality (VR) could introduce an entirely new dimension to *Arsenal 3D*. In AR, players could place FPS arenas into their physical space using mobile cameras, while VR could offer fully immersive battlefields with motion-controlled aiming, movement, and combat simulation. These technologies would enhance realism and player immersion on a level not possible through traditional screens.

Genre Blending: FPS Meets RPG and Survival

The future of *Arsenal 3D* could include genre-crossing elements such as survival mechanics (limited ammo, crafting health packs), RPG-style skill upgrades, or open-world exploration. Introducing these systems would offer richer progression, replayability, and broader player appeal beyond traditional shooter fans.

Mobile Optimization and Lite Versions

As mobile gaming dominates the market, a scaled-down version of *Arsenal 3D* optimized for lower-end smartphones could be developed. With touch controls, compact level designs, and cloud sync options, a mobile-first edition would cater to on-the-go players while maintaining the original experience's intensity.

Personalized Gameplay via Data Analytics

Player behavior tracking can unlock deeper personalization in *Arsenal 3D*. Analytics could recommend difficulty levels, loadout presets, or even generate AI enemies based on the player's past strategies. This data-driven customization would make gameplay more dynamic, rewarding, and user-centered.

Competitive Play and FPS Esports

The rising interest in competitive shooters opens opportunities for *Arsenal 3D* to expand into

esports. Structured tournaments, ranked modes, and spectator features could be introduced to attract serious players and content creators. With organized play and regular updates, the game could build a loyal community and establish a presence in the FPS esports space.

CHAPTER – 5

CONCLUSION

Arsenal 3D represents an ambitious step into the dynamic world of 3D game development, blending the excitement of first-person shooter gameplay with custom-designed environments and immersive mechanics. The project provided hands-on experience in working with professional tools such as Unity and Blender, allowing for the creation of a responsive game world filled with stylized enemies, level transitions, and combat-based challenges. Through the development process, we explored fundamental concepts like health management, enemy AI behavior, user interface design, and player feedback systems. The result is a compact yet engaging FPS game that reflects both technical learning and creative experimentation.

This project has not only enhanced our understanding of game logic and 3D environment design but has also highlighted the value of collaboration, planning, and iterative testing. As first-time developers, building *Arsenal 3D* introduced us to the complete lifecycle of game creation—from asset modeling and scripting to debugging and performance optimization. The experience underscored the importance of balancing creative vision with practical implementation, especially when managing resources, timelines, and learning curves.

- **Future Directions and Potential Trends**

  - **Multiplayer and Co-op Gameplay**
    Future updates could include real-time multiplayer support, allowing players to team up or compete in various game modes, enhancing engagement and replayability.
  - **Story-Based Campaign Mode**
    Adding a structured storyline with missions, characters, and cutscenes could give the game narrative depth and emotional impact, transforming it from a prototype into a full-fledged title.
  - **AI Enhancement and Dynamic Difficulty**
    Integrating machine learning or adaptive AI could allow enemies to adjust their strategies based on player behavior, making each playthrough uniquely challenging.
  - **Open-World Expansion**
    Transitioning from level-based design to open-world exploration with dynamic events, side missions, and resource management would significantly increase content and player

freedom.

- **Virtual Reality (VR) Integration**
  Introducing VR support could create a deeply immersive combat experience, with realistic aiming, movement, and interaction—perfect for FPS mechanics.
- **AR-Based FPS Training Mode**
  Augmented Reality features could allow players to place virtual enemies or levels in their physical space via mobile devices, bridging the digital-physical gameplay gap.
- **Mobile Optimization and Cross-Platform Play**
  Releasing a mobile-friendly version and enabling cross-platform progression would open up the game to a wider audience, especially casual gamers and on-the-go players.
- **Procedural Level Generation**
  Using procedural generation to create levels on the fly would offer unlimited gameplay variation and encourage longer play sessions without repetitive content.
- **User-Created Content & Level Editor**
  Empowering players to create and share custom levels, characters, or mods would foster a creative community and extend the game's life cycle.
- **Competitive FPS Leagues and Tournaments**
  Structured leaderboards, timed events, and esports-style tournaments could transform *Arsenal 3D* into a competitive platform for skill-based FPS players

CHAPTER – 6

REFERENCES

Animated Pixel Adventurer by rvros

Fantasy Knight - Free Pixelart Animated Character by aamatniekss

http://www.unity3dstudent.com/, accessed on 23rd January, 2013

Free Pixelart Tileset - Cute Forest by aamatniekss

Project "Perihelion" Document by Crtl Alt Elite, accessed throughout Documentation Period

Software Evaluation - A Product Perspective by Infosys, accessed on 28th May, 2013

Software Engineering in Games by Balazs Lichtl and Gabriel Wurzer from Institute of Computer Graphics, Technical University of Vienna, accessed on 26th May, 2013