Game Development Using Unity

Submitted by
Muhammad Atif

Supervised by
Dr. Sara Shahzad

BS Computer Science
Session 2020 - 2021

Department of Computer Science
University of Peshawar

Contents

Chapter 1

Introduction

Proximity Assault is an action game in which the player takes out foes that stand in the way of advancement. The player must strategically battle the enemies, who are actual human. The difficulty level rises as the player advances; adversaries attack when the player gets too close to them. The game has an AI engine that increases the difficulty and realism of enemy engagements, as well as a shop system where players may buy weaponry.

1.1 Story Line

The area was attacked by terrorists and they took control of most of, the nuclear plans, the role of the player is to terminate them and protect the world from another destructive war. The player plays as a skilled terminator who is charged with the mission of terminating all the enemies and securing the nuclear plans

1.2 Features

- **Uniqueness in environment:** The game provides a unique environment for every level and player will experience gameplay in different environment every times.
- **Diverse Bestiary:** As the player progresses in the game he will different levels of enemies.
- **Different Types of Arsenal:** The game provides a store where you can purchase different types of weapons.
- **Shop System:** To make the game more challenging a shop system can be used to buy weapons.
- **Learn Game Development** The design and implementation of this game will help me learn the basic interaction with unity environment and learn how game development works.
- **Target Audience**: The target audience for this game is players who enjoy action games with focus on exploring different conspiracies.

1.3 Project Description

Aspiring game developers lack practical experience in game development, this project provides a platform for learning the basic interaction with unity environment as well as hand on experience on animation, scripting and level design. The main focus of this project is to gain industrial level experience in gaming and learn all the basic technologies used during game development journey.

1.4 Objectives

The objectives of this project are to:

- Develop a 3D mobile game called Proximity Assault using Unity Game Engine.
- Implement a variety of game play systems;
- including an inventory system,
- a health system,
- a damage system,
- an AI perception system,
- a projectile system,
- a shop system,
- a UI management system and
- A level management system.
- Test the game on a variety of devices, including Android and iOS devices.

Unity Game Engine use C# (C-Sharp) language for programming. This Engine was developed by Unity Technologies, released in June of 2005 in Apple Developer Conference as a Mac OS X Game Engine. Unity Game Engine is proprietary software but free for students, this project was developed using an educational version of the Unity Game Engine. Unity Engine targets the following API

- Direct3D on Windows and X-box 360: Used to render in three-dimensional graphics in applications where performance is very important. It uses hardware acceleration of the 3D rendering pipeline.
- OpenGL on Mac: Abstract API for drawing 2D and 3D graphics.
- OpenGL ES on Android and iOS: A subset of the OpenGL API designed for embedded systems.

The Unity Game Engine supports the use of texture compression and resolution settings for all the platforms that the game engine supports. The Engine provides support to bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadows maps, render-to-texture and full screen-post-processing effects.

1.5 History of Unity Game Unity Engine

The engine made its first appearance in June of 2005 in Apple Worldwide Developer Conference, it was extended to 21 platforms and the latest version of Unity is currently called Tech Stream with version number 2023.1.5 released in July of 2023. Games developed by Unity were downloaded more than 5 billion time and about 2.4 billion different mobile devices were used to develop them.

1.6 Main Concept of Unity

The workflow of unity is built around the structure of component. A component is a smaller part of larger machine and in simple words its something that is complete on its own. For example in a PlayStation controller, it has many buttons but each button has no idea that there are other buttons with different behavior. Every button function independently and the function controller is a one way street, and its task will never change due to what it is plugged into. This component can work as a standalone device and with multiple devices.

1.7 Project Motivation

This project aims to revolutionize the genre of action based games with modern design and that provide a fresh feel to the new generation. There were a lot of other motivations for the game including the creative environment of Unity. It's a chance to provide the developer with a learning curve and improve their knowledge of game development. One of the reasons that motivate me to this project was the programming language used as a developer I like to program in C++ so I decided to give a try to game development with C# without switching to another language.

1.8 Project Significance:

The significance for this project are as followed

- **Refreshing the Action Game Genre:** Many of the modern entries/players still adhere to this category of gaming and Proximity Assault on the other hand filled with strategic planning and humorous tone could attract wider range of audience.
- **Show Casing the Power of Unity:** This game serve as a proof about the informing developers about the power of unity by making this game more visually appealing and engaging in term of graphics.
- **Encourage Developers:** The development of this game will be documented and shared with public.

1.9 Project Uniqueness:

This game development project puts many aspects from real world in to this project.The game is an action based game. Player navigating through different environment experiences different types of enemies.

1.10 Project Scope:

Project Scope is a map that guides the project keeping it on the track to the development journey, ensuring the team is on the track and in the end delivers an excellent product.

1.10.1 Imagine:

- You, the fearless hero navigating through the bustling areas of enemies.
- Fighting with a diverse set of enemies in which each carries a different set of abilities and power that can cause a lots of damage to you.
- Equipping you with huge arsenal of bug-bashing tools, along with some special abilities from store and some new experimental gadgets from shop/inventory.

The project scope will clarify:

- **What's In:**We will have 10-15 hour of level design, 10-15 different enemies to fight with along with at least 8 upgradable weapons/tools, Leaderboards, achievements, and an optional humorous touches add spice to the mix.
- **What's Out:**VR support, complex multiplayer features and extensive character customization won't be part of the initial launch. The main focus will be on delivering the core gameplay first.
- **Why these Choices:** Budget, timeline and team size play a key role in these deliverables. We aim to allocate all the resources with great care and completely utilize them without compromising the project.

The project scope keeps the project in a fence. We can always adjust and adopt the boundaries, ensuring a fun and achievable development process. The table below summarize these features in a more understandable way.

Elements	**What's Included**	**What's Excluded**
Platforms	PC & Mobile	VR
Engine	Unity	None
Game Play Hour	10-15	None
Environment	Diverse City Locations	None
Enemies	Increased with level	None
Extermination Tools	8+ Up gradable Options	None

Chapter 2

Feasibility and Analysis

2.1 Existing Systems

Call of Duty, often abbreviated as CoD, is a first-person shooter video game series centered on humanity's intense battles against various threats, including terrorists, military factions, and supernatural forces. Developed by multiple studios and published by Activision, the franchise is known for its cinematic action, diverse gameplay modes, and competitive multiplayer experience.

2.1.1 Variety of Enemies

- **Modern Warfare:** Engage in intense firefights against ruthless terrorist organizations, heavily armed insurgents, and tactical military units. Expect high-stakes missions that test your skills in real-world conflict scenarios.
- **Zombies Mode:** Prepare for waves of undead hordes, featuring a variety of monstrous foes. Survive increasingly difficult rounds while utilizing an arsenal of weapons and traps to fend off the relentless onslaught.
- **Advanced Warfare:** Battle futuristic enemies equipped with exosuits and advanced technology. Utilize enhanced mobility and advanced weaponry to outmaneuver and defeat these formidable opponents.

2.1.2 Weapons

- **Diverse Arsenal:** Call of Duty offers an extensive range of weapons, from classic rifles and shotguns to modern assault weapons and high-tech gadgets, allowing players to customize their loadouts to fit their playstyle.
- **Vehicles and Equipment:** Take control of armored vehicles, drones, and helicopters to dominate the battlefield, providing tactical advantages and unleashing devastating firepower.

2.1.3 Intense Action

- **Explosive Moments:** Call of Duty thrives on fast-paced, action-packed gameplay. Expect dramatic set pieces, high-octane firefights, and cinematic explosions that elevate the adrenaline levels.
- **Co-op and Multiplayer Chaos:** Team up with friends in cooperative modes or compete against players worldwide. The shared excitement and strategic coordination in multiplayer battles are what make Call of Duty an unforgettable experience.

2.1.4 A Series Full of Innovation

- **Multiple Titles, Multiple Eras:** With numerous main entries and spin-offs, the Call of Duty series spans various historical periods and future settings. Each title introduces new gameplay mechanics, enemy types, and immersive narratives, keeping the experience fresh.
- **Different Styles:** From the gritty realism of "Call of Duty: WWII" to the futuristic warfare of "Call of Duty: Infinite Warfare," there's a Call of Duty game for every preference.

2.1.5 Advantages

- **Diverse Enemy Types:** Call of Duty offers a wide variety of adversaries, from ruthless terrorist factions to advanced robotic enemies. Each encounter feels unique and engaging, keeping players on their toes.
- **Extensive Arsenal:** The game features a vast array of weapons, from classic firearms to cutting-edge technology like drones and tactical gear. Players can experiment with different loadouts to find the most effective combinations for their playstyle.
- **Co-op Mayhem:** Team up with friends in cooperative modes for chaotic and thrilling gameplay. Coordinating tactics, reviving teammates, and sharing the thrill of victory create an unforgettable multiplayer experience.
- **Variety of Game Modes:** The Call of Duty series caters to diverse preferences. From the realistic combat of "Call of Duty: Modern Warfare" to the arcade-style fun of "Call of Duty: Warzone," there's something for everyone.

2.1.6 Drawbacks

- **Repetitive Gameplay:** Despite the diverse enemies, the core gameplay loop can become monotonous. Engaging in similar combat scenarios repeatedly, even with different weapons, may lose its excitement for some players.
- **Cinematic Clichés:** Call of Duty often embraces its blockbuster action movie roots. If you're not a fan of over-the-top scenarios and scripted sequences, the experience might feel predictable.

- **Technical Issues:** The series occasionally suffers from bugs, server issues, and imbalances, which, while not game-breaking, can detract from the overall experience at times.

2.1.7 Problems

- **Steep Learning Curve:** The fast-paced nature of the game can be overwhelming for newcomers. Without adequate tutorials or guidance, new players might find themselves struggling to keep up.
- **Narrative Focus:** The story often takes a backseat to the action. While some titles attempt a deeper narrative, most prioritize explosive gameplay over storytelling, which might leave players wanting more depth.
- **Limited Replayability:** Depending on the specific game mode, the lack of substantial endgame content or varied experiences may result in diminished replay value once the campaign is completed.

2.1.8 Overall

Call of Duty offers a thrilling and dynamic experience for players who enjoy its high-octane action, cinematic gameplay, and competitive multiplayer. However, its repetitive mission structure, occasional technical issues, and sometimes shallow narrative may not resonate with everyone. If you're seeking a deeply immersive and polished shooter, Call of Duty might not be your ideal choice. But if you're in for intense co-op action and fast-paced combat, Call of Duty is definitely worth diving into.

2.2 Proximity Assault

Proximity Assault is an action based game, the player is aimed with the objectives of eliminating all the terrorist in the specific area. The area was occupied by some terrorist organization and they have occupied some nuclear plans from the government. The player will recover all these objective and move forward in the game.

2.2.1 Twist

Enemies Apocalypse: In a world overrun by a terrorist organization and they took controll of some nuclear plans from the government, The player is aimed with the objectives to recover those plan and uncover the aims of occupying these nuclear plans.

2.2.2 Dream world Assault

Enter the surreal realm of dreams where nightmares manifest as monstrous skilled terrorist. As a oneiric hero, you must navigate the ever-shifting landscape of the subconscious, solving puzzles and using dream-warping tools to vanquish these fantastical skilled enemies.

2.2.3 Target Audience

Proximity Assault targets a more casual market with simpler controls and less intense action. Or maybe it focuses on educating players about real-world situations of controlling the an organization which rebelled and went on this way.

2.3 Positive Impact

2.3.1 Educational Value

Exterminator could raise awareness about responsible pest control practices and the importance of protecting ecosystems. Educational elements could be subtly woven into the gameplay or story, making it both entertaining and informative.

2.3.2 Co-op Fun

Teaming up with friends to tackle infestations could foster teamwork and communication, making Exterminator a fun and social experience.

2.4 Detail Feasibility Report

Below is a detail feasibility report for Proximity Assault.

2.4.1 Technical Feasibility

The technical requirements for this project are a system that can support the game engine which in this case will be Unity, Unity requires a subscription but this project will be using the student plan, in which the team can use all the features for free. Unity works with C# which is a similar to C++. As a C++ Programmer we have a very good understanding of the concepts in C++ which we can easily apply to this scenario. The IDE the project will be coded with is Visual Studio Community which have a built in compiler with it. The design phase of the project will take a little longer because the development team will have to consult someone to deal with the design of the interfaces and design architecture of the system. The system can be easily completed with the above mention tools and techniques. However the system will also be tested and development will be continued in small phases as to have the room for maxi-mum improvements to interface, internal mechanisms and other features to make it very engaging. The development will also requires advance features like complex AI or detail environment, which will requires additional learning resources.
The problems faced by implementing the system will requires an expertise in testing for which the system must be test be a person from out of the team, as the a solo developer we will have to manage the time and available resources with great efficiency in order to complete all the phases of the project. The testing and designing will require some extra time which can be solve either by hiring a person from out of the team or we will develop them ourself but the at least one will requires time.
The system will probably take months to be completed because it requires the process

of learning along with development of the system. The team will try to adjust everything accordingly in order to achieve the development of the system in the limited time available.

2.4.2 Economic Feasibility

The resources required to learn the basic interaction with Unity environment will requires the team to have some skills in becoming familiar with the environment. The team will purchase some courses from Udemy in order to achieve their objective in faster and efficient way. The team will use assets which are probably free or low cost but once the testing is done the team will invest in the purchasing of assets but if there was sufficient time most of the assets will be developed for the system by the team which at some stage of development will include a designers and testers. As development is completed the team will be testing and finding bugs in the system before publishing it online, once everything is good it will be uploaded at some platforms like steam and mobile app stores with a revenue model like paid download and in-app purchases. The game market is very competitive; it would require some time to adjust itself to stakeholders out there. The team will also keep a backup plan to have budget for the marketing of the system. If the system requires some developers to be hired for some parts of the project, the team will make sure there is enough budget to get the best developers out there.

2.4.3 Social Feasibility

The most important of the social feasibility is the ethical considerations in the game, as discussed be-fore the game aims to highlight a point to the nation which is responsibility; Responsibility is one of the important aspect of the humans. This will provide a message to the nation to be aware and be awake to take action before its too late. The system will be designed in such a way that no culture, race, age or gender is targeted. The player will be able to experience a very friendly environment in which they will have no idea of the they being discriminated in anyway instead the team will design strategies in order to remove anything that will contribute to discrimination of that particular group. Instead the game target audience globally. The development of the game will be an educational asset for other students, developers and anyone who wants to start with game development but requires a clear road map for them. All the resources, documentation and any assets that are required for redesigning of the game will be available to the everyone from around the globe. The last point is basically highlighted to give the project and educational value. Accessibility features will be given more diverse touch in order to make the game playable for people with different abilities like adjusting the difficulties level and color blind options along with subtitle options to be incorporated in to the game.

Chapter 3

Requirement Specification

3.1 Introduction

This part of the document outlines the requirement specification of the game Proximity Assault which will serve as the blueprint or the base for the features, functionality and quality of the game. The aim of clearly defining these requirements is to keep everyone including developers, designers, testers and even the lovely audience to be on the same page, leading to a successful development journey.

3.2 Functional Requirements

3.2.1 Main Gameplay

- Player controls a character.
- Levels filled with different enemies.
- Player uses different weapons and tools to eliminate enemies

3.2.2 Progression

- Unlock new game level with experience or in-game currency.
- Difficulty increases with every new level.
- Optional side objectives to make the game more engaging (Optional).

3.2.3 Inventory and Upgrades

- Player can use different weapons and tools with different attributes.
- Option to upgrade weapons for improving power and efficiency.
- Able to use some special abilities in the game.

3.2.4 Menus and UI

- Feature an extremely smooth and easy User interface for the players.
- Pause menus with options to save, quite or adjust setting of the game.
- Health bar, objective status indicator and other elements for feedback.

3.3 Non-Functional

The non-functional requirements of the system are about how a system should behave instead of what specific function to perform.

3.3.1 Performance

- Maintain smooth frame rate and responsiveness on target platforms (iOS, Android).
- ‘ Optimize every feature in order to avoid performance issues.

3.3.2 Graphics And Audio

- Make sure the frame rate does not fall and is stable [4]
- Graphics should be visually appealing and according to the theme and tone.
- Good sounds and background music.
- Sounds that inform the user about the certain actions like low health and enemy response fire.

3.3.3 Controls

- Responsive and faster controls for aiming, and interacting with the environment..
- Sufficient options for customization.
- Sensitivity features.

3.3.4 User Interface

- New game, continue game, exit.
- Links to social media platforms.
- Setting button to adjust controls.

3.4 UI Requirements

This includes the guidelines for how a game user interface should look and function when the player is interacting with the system. It includes menus, buttons, maps, inventory screen and health bar.

3.4.1 Main menus

- Start game, continue game, and option buttons.
- Information about the game and social links.

3.4.2 Gameplay Screen

- Health bar, Objectives status indicators, ammo/resources indicators.
- Mini map or level overview (Optional).
- Button to access inventory.

3.4.3 Inventory Screen

- Display of the available resources.
- Upgrade options with cost and description.
- Special abilities management with usage indicators.

3.4.4 Pause menu

- Options to save, resume, quit and adjust game settings.
- Control layout and settings overview.

The main and most important requirement of the game will be learning how the process of designing and developing a video game works. To extract useful results from all the phases involved in game development like pre-production, production and post-production. More requirements include learning the art of how to draw graphics and music for game, scripting, artificial intelligence, level creation and user interface.

3.5 Persona

The Operation Bio-Purge has a very simple persona in which a user named Ali, the Casual Gamer.

3.5.1 Demographics

- Name: Ali
- Age: 23
- Occupation: Marketing Specialist
- Location: Peshawar City
- Device: IPhone 11

3.5.2 Gaming Habits

- Play mobile game in short breaks or during free time.
- Prefer casual, puzzle like games with simple mechanics and clear goals.
- Play games with light heart themes and great visuals.

3.5.3 Motivation

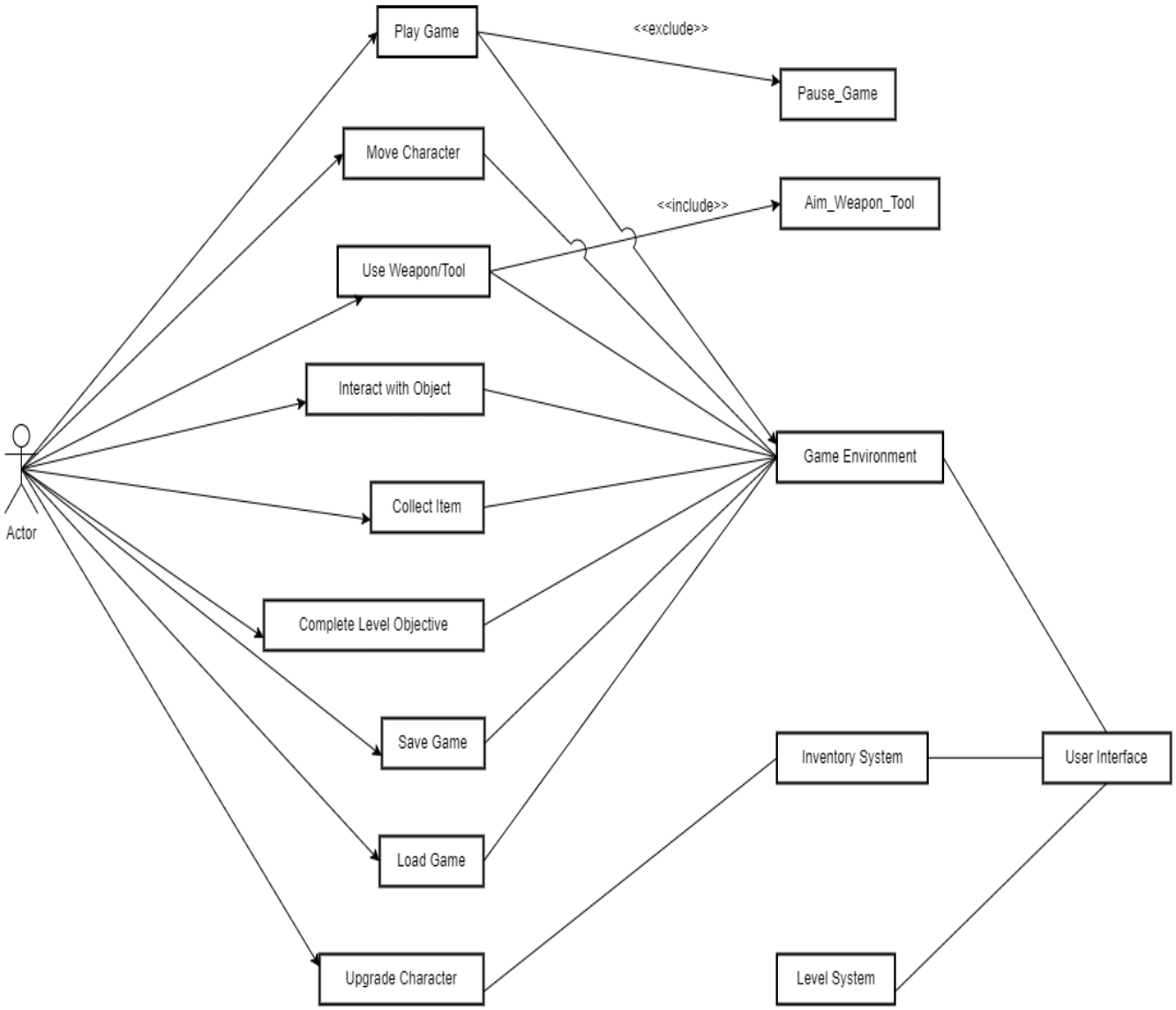
- For relaxing himself and enjoying in free time.
- Enjoys the satisfaction of completing short objectives.
- Appreciates game with quick to do actions.
- Like good game designs.

3.5.4 Concerns

- Limited time to play the game
- Dislike ads and in-game pop-ups for purchases or any sort of disturbance.
- Do not like repetitive game play.

3.6 User Interactions with the System through Use Case Diagram

This section dives into the user interaction between the user and the game using a use case diagram. A use case diagram describes all the functionalities of the system along with external users aiming to describe how an actor (user) interacts with the system. A use case diagram also specifies the main functionalities of the system that a user can perform while interacting with the system. This diagram also communicates the system behavior to the stakeholder in an easy and understandable way. The testing phase also uses the use case diagram as a base.



The UML diagram describes the user interaction and functionalities of the "Proximity Assault" game system.

3.7 System Boundary

- **Actor(Player):**The player represents the user interacting with the system.
- **Game Environment:**This represents the virtual environment where the game takes place including elements like terrains, enemies and objects.
- **Inventory System:** Manages the player inventory, the items that a player collects during the game play or purchases them from a store.
- **Level System:** This functionality manages the progression and objectives with in each level.

- **User Interface:** Provides the visual and interactive way through which a user interacts with the game.

3.7.1 Use Case

The use case describes various actions or functionalities that a player can perform while interacting with the system.

- **Play Game:** Begins the game play with the game environment.
- **Move Character:** Allows the player to move their character with in the game environment.
- **Interact with Object:** Enables the player to interact with objects present in the game environment.
- **Use Weapons/Tools:** Allows the player to utilize weapons or tools with in the game.
- **Collect Item:** Enables the player to collect game items found with in the game environment.
- **Upgrade Character:** Allow the user to upgrade their character abilities.
- **Complete Level Objectives:** Indicates the accomplishment of objectives specific to a level.
- **Save Game:** Allow the players to save their current progress within the game.
- **Load Game:** Allow to load the game from previously saved state to provide more engaging feeling the users.

3.7.2 Include and exclude

Indicate the relationships between use cases

- **Play game exclude pause game:** This means that when the player is playing the game, the pause functionality is not available.
- **Use Weapons/Tools includes Aim Weapons/Tools:** This signifies that aiming is a part of the process of using weapons within the game.

3.7.3 Actor Use Case Relationships

Arrows connecting actors (Player) to use cases indicate that players interact with those functionalities.

3.8 Development Environment Requirements

Development of this game involves various tools and software with their own system requirements, below is a very detailed information of the development environment required for this game.

3.8.1 Development Environment

Operating System

- **Minimum:** Windows 10 64-bit (recommended latest stable version).
- **Alternative:** MacOs Mojave 10.14+ (recommended: latest stable version).
- **Justification:** Both Unity and MacOs are officially supported by Unity.

3.8.2 Hardware

Processor

- **Minimum:** x86 or x64 architecture with SSE2 instruction set support(e.g. Intel core i5-4460 or AMD equivalent).
- **Recommended:** Newer processor with multiple cores (e.g., Intel core i7 or AMD Ryzen) for smoother performance during development and testing.

RAM

- **Minimum:** 8GB
- **Recommended:** 16GB for multitasking and handling larger projects

Graphics Card

- **Minimum:** DX10, DX11, DX12 capable (e.g., NVidia GTX 970 or AMD Radeon R9 290)

Storages

- **Minimum:** 20GB available space for Unity and Project Files
- **Recommended:** More space depending on project size, additional tools and asset libraries.

3.8.3 Software

Unity Game Engine

- Download the latest stable version that is compatible with your system.
- Consider Unity Personal for small projects or Unity Plus for additional features.

Version Control

- Git (Recommended) with code hosted on GitHub

Graphics Designing Software

- Photoshop, GIMP for creating UI elements textures or logos.

3D Modeling Software

- Blender

Audio Editing

- Audacity

Asset Creations

- **Asset Creation:** Asset will be used from unity market place.

3.9 Minimum Hardware Requirements for Running the Game

These are the requirements for running the Operation Bio-Purge game on Android or iOS. These are the general requirements needed for running the system.

3.9.1 Android

- **Processor:** Dual-core 1.2 GHz processor (recommended: Quad-core 1.5 GHz or better)
- **RAM:** 1GB RAM (recommended 2GB or more)
- **Storage:** 500 MB free space
- **Operating System:** Android 5.0 or later (recommended Android 8.0 or later).
- **Graphics:** Adreno 305 GPU or equivalent (recommended: Adreno 405 or later).

3.9.2 iOS

- **Device:** iPhone 6 or later (recommended iPhone 7 or later).
- **Operating System:** iOS 10 or later (recommended: iOS 12 or later).
- **RAM:** 1GB RAM (recommended: 2GB or more)
- **Storage:** 500 MB free space.

Chapter 4

System Design

System design is the process of defining the [2] architecture, components, modules, interfaces, and data for a software system to meet specified requirements.

System design involves defining the overall architecture, breaking down the system into components and modules. Architectural design establishes the high-level structure, relationships between components, and communication mechanisms within the system.

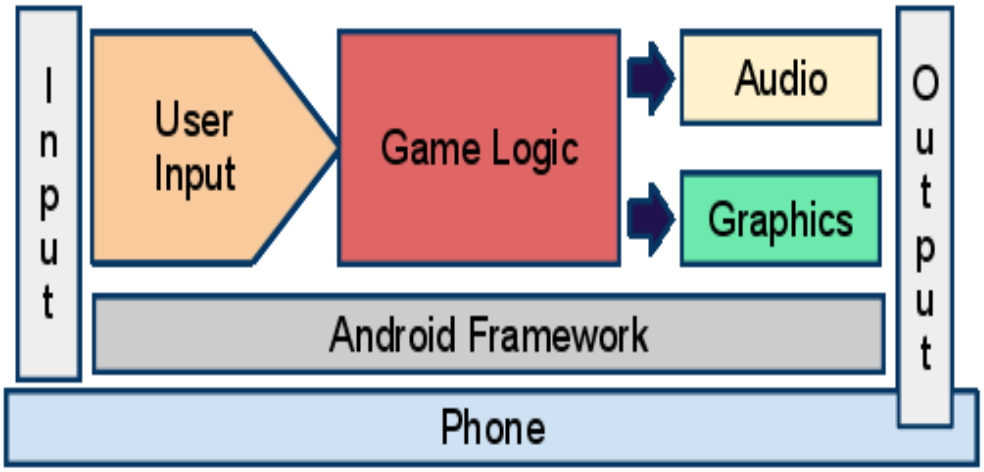
4.1 Architecture Design

Architecture design is an important phase in system development that defines the structure and organization of a system. It defines the overall framework that indicates how various components interact, ensuring that the system meets functional requirements and performs efficiently. The architecture serves as a blueprint, guiding developers throughout the implementation process and enabling scaling and maintenance.

- **Component Identification:** Defining the individual components that make up the system, such as modules, libraries, and services.
- **Interaction Patterns:** Specifying how these components communicate and collaborate, including protocols, data flow, and interfaces.
- **Design Patterns:** Utilizing established design patterns to solve common problems in software architecture, promoting re-usability and clarity.
- **Scalability and Performance:** Considering how the architecture will support future growth and maintain performance standards under varying loads.

4.1.1 Architecture Design of Gaming System

The architecture of a game plays a very important role in engaging in responsive experiences [3]. Following are the architectural components typically found in the gaming system.



- **Game Engine:** A game's central component that handles input, handles physics, renders images, and coordinates game logic. With built-in tools and libraries, game engines such as Unity or Unreal Engine facilitate creation.
- **Game Client:** Players communicate with the game using this interface. It consists of the graphics systems, gaming mechanics, and user interface (UI). Input from the player is processed by the client, which also connects to the server and shows game statuses.
- **AI System:** This part controls how adversaries and NPCs (Non-Player Characters) behave, including how they respond to player input. Path finding, decision-making algorithms, and state machines are examples of AI.
- **Networking Module:**The networking module, which manages data exchange between clients and the game server, is crucial for multiplayer games. It guarantees the synchronization of player movements, game states, and other in-the-moment interactions.
- **Database/Storage System:**This component is responsible for storing the player data, player progress and other information that needs to be persisted in the database. A cloud-based database is a good option to be used depending on the game architecture.
- **Scripts:** Scripting capabilities offered by many games can be used to modify the game play depending on the game's architecture.

4.1.2 Types of Architectures

The following text discuss some of the Architecture that could be used in the game systems [**?**].

- **Modular Architecture:** A game with a modular architecture is easier to maintain and expand since it is divided into discrete modules for things like physics, AI, and rendered. Code reuse is encouraged by the separate development and testing of each module. Careful design is necessary to provide seamless integration and manage dependencies.
- **Entity-Component-System (ECS) Architecture:** Game logic is divided into entities (game objects), components (data), and systems (logic) in ECS design. This method encourages code reuse and the division of responsibilities while improving flexibility and efficiency. ECS scales well and works well for applications that require high performance, despite its initial complexity of implementation.
- **Client-Server Architecture:** With client-server architecture, user input and rendering are handled by the client, and game logic is executed on a server. Updates and management of multiplayer games are made easier by this centralization. But it needs stable network connections, and a strong infrastructure is needed because the server can become a bottleneck.
- **Peer-to-Peer (P2P) Architecture:** Through direct game state sharing, each player's device serves as both a client and a server in a peer-to-peer (P2P) architecture. This can lower latency and lessen server load. But maintaining consistency and security is difficult, and it's not appropriate for every kind of game, particularly ones that require central authority.
- **Hybrid Architecture:** To capitalize on each architecture's advantages, hybrid architecture blends components from other architectures. This adaptability enables customized solutions to meet particular gaming needs. It can, however, be difficult to plan and execute, requiring careful component integration.

4.1.3 Architecture of Proximity Assault

To maximize its design and performance, the Proximity Assault Game makes use of both Entity-Component-System (ECS) architecture and modular architecture. The game's modular architecture separates the game's components (physics, AI, and rendering) to enable separate development and upkeep. By breaking down game objects into entities, components, and systems, ECS Architecture improves flexibility and efficiency while making game behaviors easier to handle and encouraging code reuse. This combination guarantees a game structure that is performant, scalable, and maintainable.

Detailed Description of the Architecture

1. **Game Engine: Unity**
    - Unity is the game engine used, providing tools for rendering, physics, and scripting.
2. **Core Systems**

- **Game Loop**: Manages the main cycle of the game, including updating game states, handling user inputs, and rendering frames.
- **Input Handling**: Captures and processes player inputs from devices like keyboards, mice, or game controllers.
- **Physics Engine**: Simulates realistic physics, including collision detection and response.

3. **Graphics and Rendering**

   - **3D Models and Textures**: High-quality models and textures for characters, weapons, and environments.
   - **Shaders**: Used to create realistic lighting and effects.
   - **Animation System**: Manages character animations, including movement, shooting, and reloading.

4. **AI and NPCs**

   - **AI System**: Controls the behavior of enemy NPCs, making them react intelligently to player actions.
   - **Pathfinding**: Algorithms like A* for navigating the game environment.

5. **Audio**

   - **Sound Effects**: Realistic gunfire, explosions, and ambient sounds.
   - **Music**: Background music that enhances the gameplay experience.
   - **Audio Engine**: Manages sound playback and spatial audio effects.

6. **User Interface (UI)**

   - **HUD**: Displays health, ammo, mission objectives, and other critical information.
   - **Menus**: Main menu, settings, and pause menu interfaces.

7. **Game Logic**

   - **Mission System**: Defines and manages the objectives and progression of missions.
   - **Weapon System**: Manages different weapons, their stats, and behaviors.
   - **Health and Damage System**: Tracks player and enemy health, and calculates damage from attacks.

8. **Optimization**

   - **Level of Detail (LOD)**: Reduces the complexity of distant objects to improve performance.
   - **Occlusion Culling**: Prevents rendering of objects not visible to the player.
   - **Performance Profiling**: Tools to monitor and optimize game performance.

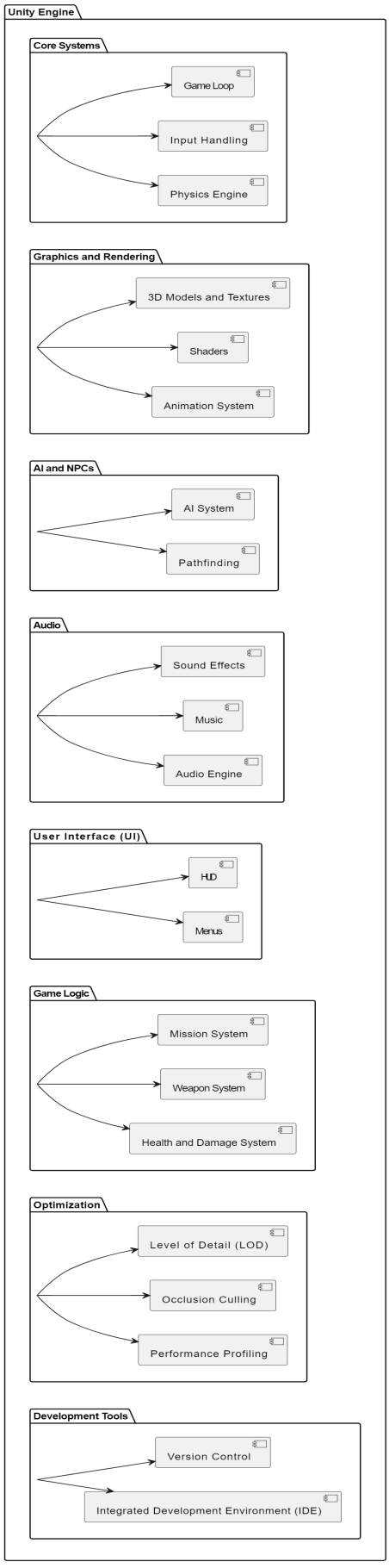
9. **Development Tools**

    - **Version Control**: Systems like Git for managing code and assets.
    - **Integrated Development Environment (IDE)**: Tools like Visual Studio for coding and debugging.

4.1.4 Architecture Design Diagram

Below is the attached architecture diagram for the Proximity Assault. The top-level package, **Unity Engine**, contains several key components organized into packages:

- **Core Systems** includes foundational elements like Game Loop, Input Handling and Physics Engine
- **Graphics and Rendering** focuses on visual aspects with 3D Models Shaders and Animation Systems
- **AI and NPCs** covers AI System and Path-finding algorithms
- **Audio** involves Sound Effects, Music and Audio Engine
- **User Interface (UI)** manages HUD and Menus
- **Game Logic** handles Mission System, Weapon System and Health and Damage System
- **Optimization** improves performance through Level of Detail (LOD), Occlusion Culling and Performance Profiling
- **Development Tools** supports Version Control Integrated Development Environment (IDE)


Unity Engine
Core Systems
Game Loop
Input Handling
Physics Engine
Graphics and Rendering
3D Models and Textures
Shaders
Animation System
AI and NPCs
AI System
Pathfinding
Audio
Sound Effects
Music
Audio Engine
User Interface (UI)
HUD
Menus
Game Logic
Mission System
Weapon System
Health and Damage System
Optimization
Level of Detail (LOD)
Occlusion Culling
Performance Profiling
Development Tools
Version Control
Integrated Development Environment (IDE)

4.2 Component Level Design

1. **Core Systems**
    - **Game Loop**
        - **Update Cycle**: Handles periodic updates to game state and logic.
        - **Render Cycle**: Manages rendering frames to the screen.
        - **Event Handling**: Processes user input and system events.
    - **Input Handling**
        - **Input Manager**: Captures and processes input from various devices (keyboard, mouse, game controllers).
        - **Action Mapping**: Maps input actions to game functions (e.g., move, jump, shoot).
    - **Physics Engine**
        - **Collision Detection**: Detects interactions between game objects.
        - **Physics Simulation**: Simulates physical behaviors (e.g., gravity, momentum).
        - **Rigid Body Dynamics**: Manages object movements and interactions based on physical properties.
2. **Graphics and Rendering**
    - **3D Models and Textures**
        - **Model Loader**: Loads and processes 3D models from files.
        - **Texture Mapping**: Applies textures to models.
    - **Shaders**
        - **Vertex Shaders**: Handle vertex transformations and lighting.
        - **Fragment Shaders**: Manage pixel-level coloring and effects.
    - **Animation System**
        - **Animation Controller**: Manages animation states and transitions.
        - **Bone System**: Handles skeletal animations for characters.
3. **AI and NPCs**
    - **AI System**
        - **Behavior Trees**: Define complex NPC behaviors and decision-making processes.
        - **Finite State Machines (FSM)**: Manage simple state-based behaviors.
    - **Pathfinding**
        - **Navigation Mesh (NavMesh)**: Defines walkable areas in the game environment.
        - **A* Algorithm**: Calculates optimal paths for NPC movement.
4. **Audio**

- **Sound Effects**
  - **Audio Clips**: Contains audio files for various game actions (e.g., gunfire, explosions).
  - **Sound Manager**: Plays and manages sound effects.
- **Music**
  - **Background Tracks**: Provides ambient music to enhance gameplay experience.
  - **Music Manager**: Controls playback and transitions between tracks.
- **Audio Engine**
  - **Spatial Audio**: Provides 3D audio effects based on player position.
  - **Volume Control**: Adjusts the volume of different audio sources.

5. **User Interface (UI)**
   - **HUD (Heads-Up Display)**
     - **Health Bar**: Displays the player's current health.
     - **Ammo Count**: Shows the number of bullets remaining.
     - **Mission Objectives**: Lists current goals and progress.
   - **Menus**
     - **Main Menu**: Provides options for starting the game, loading saved games, and accessing settings.
     - **Settings Menu**: Allows adjustments to game settings (e.g., audio, controls).
     - **Pause Menu**: Provides options to resume, restart, or exit the game.
6. **Game Logic**
   - **Mission System**
     - **Mission Manager**: Controls mission objectives, tracking, and progression.
     - **Quest Tracker**: Updates and displays the status of active quests.
   - **Weapon System**
     - **Weapon Manager**: Handles weapon switching, equipping, and firing mechanics.
     - **Weapon Stats**: Manages weapon attributes such as damage, range, and fire rate.
   - **Health and Damage System**
     - **Health Manager**: Tracks health points for the player and NPCs.
     - **Damage Calculation**: Calculates damage dealt to characters and applies effects.
7. **Optimization**
   - **Level of Detail (LOD)**

- **LOD Manager**: Adjusts the detail of objects based on their distance from the player.
- **LOD Models**: Provides different versions of models with varying detail levels.

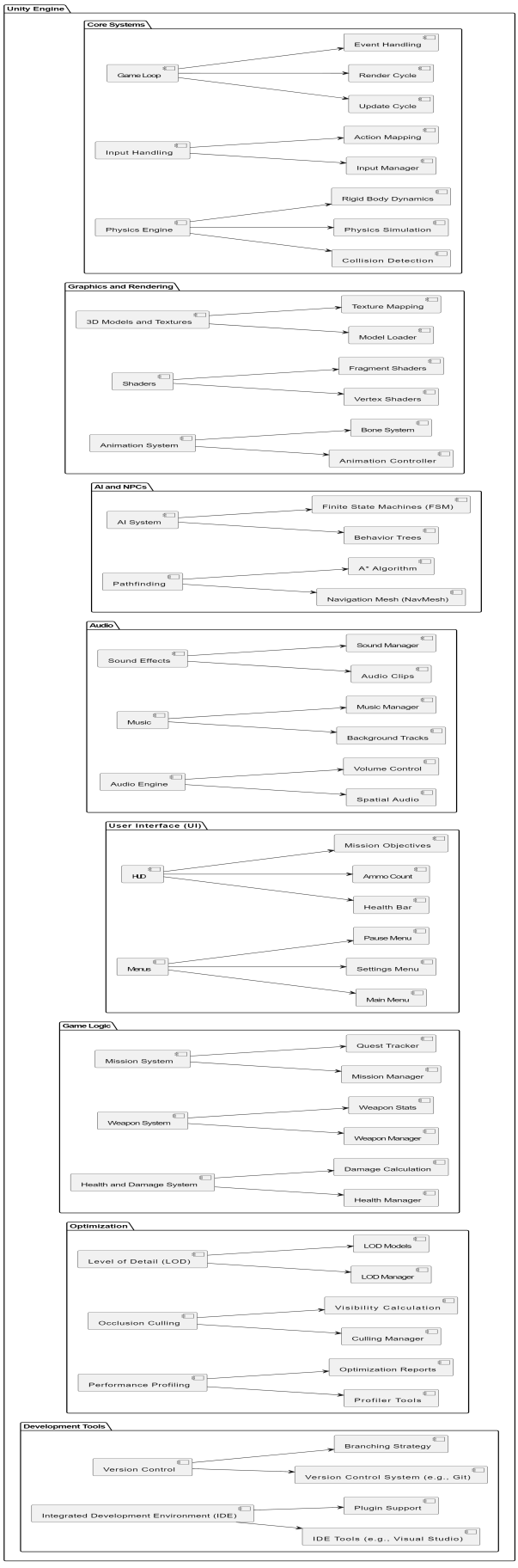
- **Occlusion Culling**
  - **Culling Manager**: Determines which objects are visible and should be rendered.
  - **Visibility Calculation**: Computes object visibility based on camera view.

- **Performance Profiling**
  - **Profiler Tools**: Monitors game performance metrics such as frame rate and memory usage.
  - **Optimization Reports**: Provides feedback on performance issues and potential improvements.

8. **Development Tools**

- **Version Control**
  - **Version Control System (e.g., Git)**: Manages code changes and collaboration.
  - **Branching Strategy**: Organizes development workflows and code versions.

- **Integrated Development Environment (IDE)**
  - **IDE Tools (e.g., Visual Studio)**: Provides coding, debugging, and testing capabilities.
  - **Plugin Support**: Extends functionality with additional tools and features.

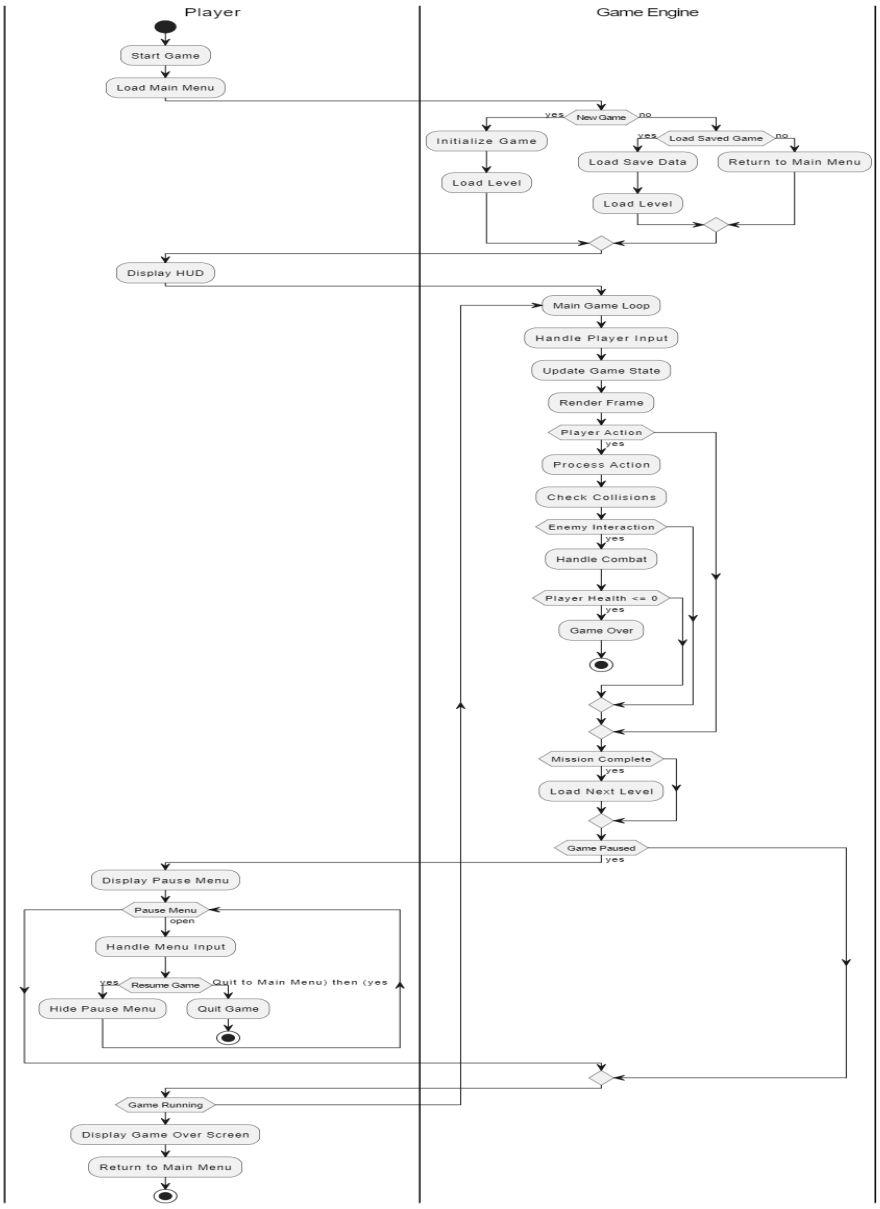

Unity Engine
Core Systems
Event Handling
Game Loop
Render Cycle
Update Cycle
Action Mapping
Input Handling
Input Manager
Rigid Body Dynamics
Physics Engine
Physics Simulation
Collision Detection
Graphics and Rendering
Texture Mapping
3D Models and Textures
Model Loader
Fragment Shaders
Shaders
Vertex Shaders
Bone System
Animation System
Animation Controller
AI and NPCs
Finite State Machines (FSM)
AI System
Behavior Trees
A* Algorithm
Pathfinding
Navigation Mesh (NavMesh)
Audio
Sound Manager
Sound Effects
Audio Clips
Music Manager
Music
Background Tracks
Volume Control
Audio Engine
Spatial Audio
User Interface (UI)
Mission Objectives
HUD
Ammo Count
Health Bar
Pause Menu
Menus
Settings Menu
Main Menu
Game Logic
Quest Tracker
Mission System
Mission Manager
Weapon Stats
Weapon System
Weapon Manager
Damage Calculation
Health and Damage System
Health Manager
Optimization
LOD Models
Level of Detail (LOD)
LOD Manager
Visibility Calculation
Occlusion Culling
Culling Manager
Optimization Reports
Performance Profiling
Profiler Tools
Development Tools
Branching Strategy
Version Control
Version Control System (e.g., Git)
Plugin Support
Integrated Development Environment (IDE)
IDE Tools (e.g., Visual Studio)

4.3 Detailed Activity Diagram for a Game

1. **Player Starts the Game**
    - The player starts the game, leading to the loading of the main menu.
2. **Main Menu**
    - The game engine handles the main menu where the player can choose to start a new game or load a saved game.
3. **New Game or Load Saved Game**
    - If a new game is selected, the game initializes and loads the first level.
    - If a saved game is selected, the game loads the save data and the appropriate level.
4. **Main Game Loop**
    - The main game loop begins, handling player input, updating the game state, and rendering frames.
5. **Player Actions and Events**
    - Within the game loop, the engine processes player actions, checks for collisions, and handles enemy interactions (combat).
    - If the player's health reaches zero, the game over screen is displayed.
6. **Mission Complete**
    - If a mission is completed, the next level is loaded.
7. **Pause Menu**
    - If the game is paused, the pause menu is displayed, allowing the player to resume the game or quit to the main menu.
8. **Game Over**
    - When the player dies, the game over screen is displayed and then returns to the main menu.

4.4 Activity Diagram

Below is the activity diagram for Proximity Assault.

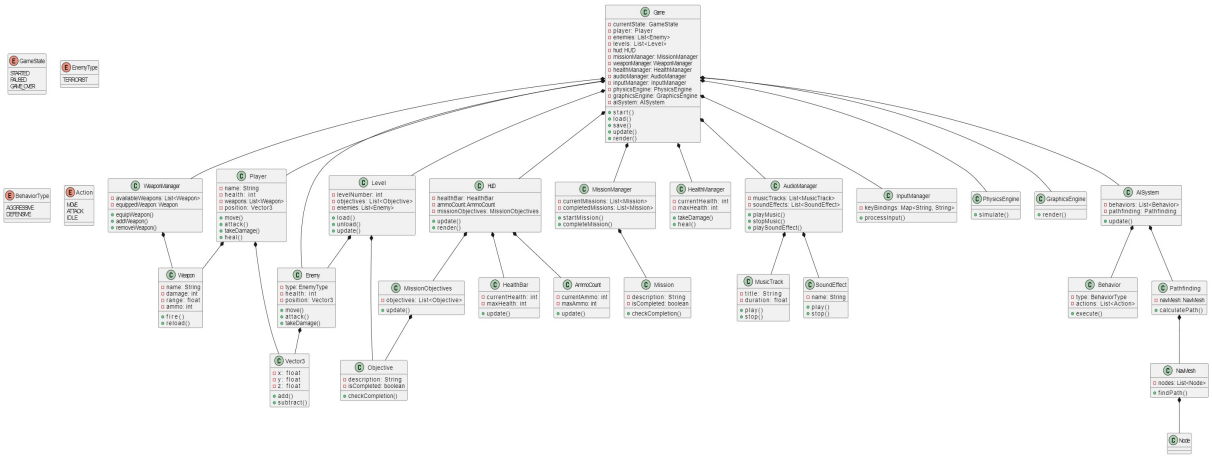


4.5 Class Diagram

A class diagram for a gaming system typically includes classes such as `Game`, `Player`, `Enemy`, `Level`, `Weapon`, `Mission`, and various managers (e.g., `AudioManager`, `InputManager`, `PhysicsEngine`, `GraphicsEngine`, `AISystem`). These classes interact to represent the game's core components, manage gameplay elements, handle player inputs, simulate physics, render graphics, and control AI behaviors. Each class has specific attributes and methods that define their roles and responsibilities within the game.

- **Game Class**: The main class containing the core components and systems.
- **Player and Enemy Classes**: Representing the player and enemies with attributes and methods for movement, attack, etc.

- **Level and Objective Classes**: Handling levels and their objectives.
- **Weapon, WeaponManager, and HUD Classes**: Managing weapons, HUD elements, and their updates.
- **Mission and MissionManager Classes**: Managing missions and their statuses.
- **AudioManager, MusicTrack, and SoundEffect Classes**: Handling audio playback.
- **InputManager Class**: Processing player inputs.
- **PhysicsEngine and GraphicsEngine Classes**: Simulating physics and rendering graphics.
- **AISystem, Behavior, and Pathfinding Classes**: Managing AI behaviors and pathfinding.
- **Vector3 Class**: Representing 3D vectors.
- **Enumerations**: Defining various game states, enemy types, behavior types, and actions.



4.6 Data Flow Diagram

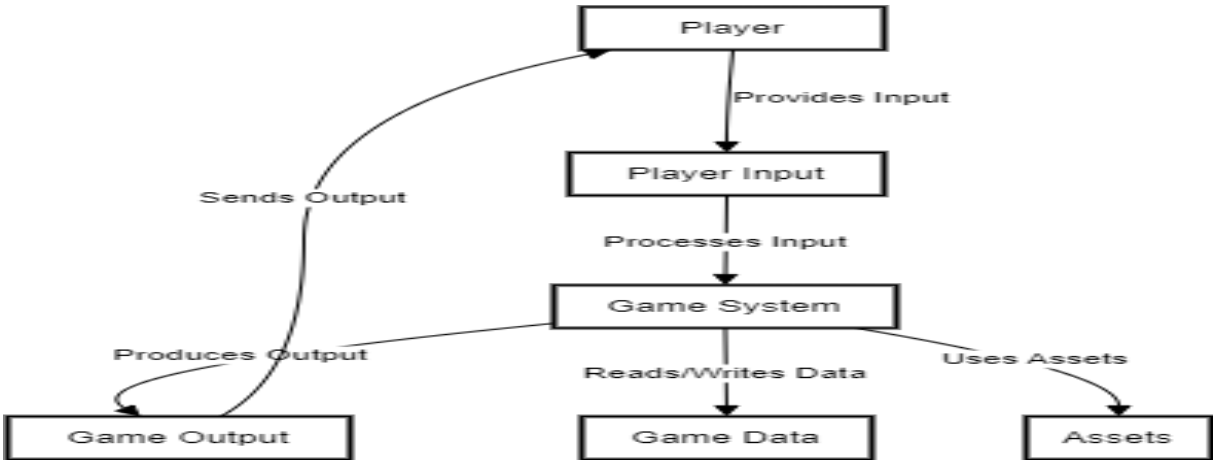
A class diagram for a gaming system typically includes classes such as `Game`, `Player`, `Enemy`, `Level`, `Weapon`, `Mission`, and various managers (e.g., `AudioManager`, `InputManager`, `PhysicsEngine`, `GraphicsEngine`, `AISystem`). These classes interact to represent the game's core components, manage gameplay elements, handle player inputs, simulate physics, render graphics, and control AI behaviors. Each class has specific attributes and methods that define their roles and responsibilities within the game.

4.6.1 Level 0 Data Flow Diagram (DFD)

A Level 0 Data Flow Diagram (DFD), also known as a context diagram, represents the entire system as a single process. It provides a high-level overview of the system's interactions with external entities such as users, other systems, or data sources. Here's a simple explanation:

- **Single Process**: The whole system is shown as one process, representing all the main functions and operations.
- **External Entities**: It includes all the external entities that interact with the system (e.g., users, other systems).
- **Data Flows**: Shows the flow of data between the system and these external entities.
- **High-Level View**: It's a big-picture view that doesn't dive into the details of internal processes or data stores within the system.

A Level 0 DFD provides a basic, overall picture of how the system interacts with its environment.



Explaination

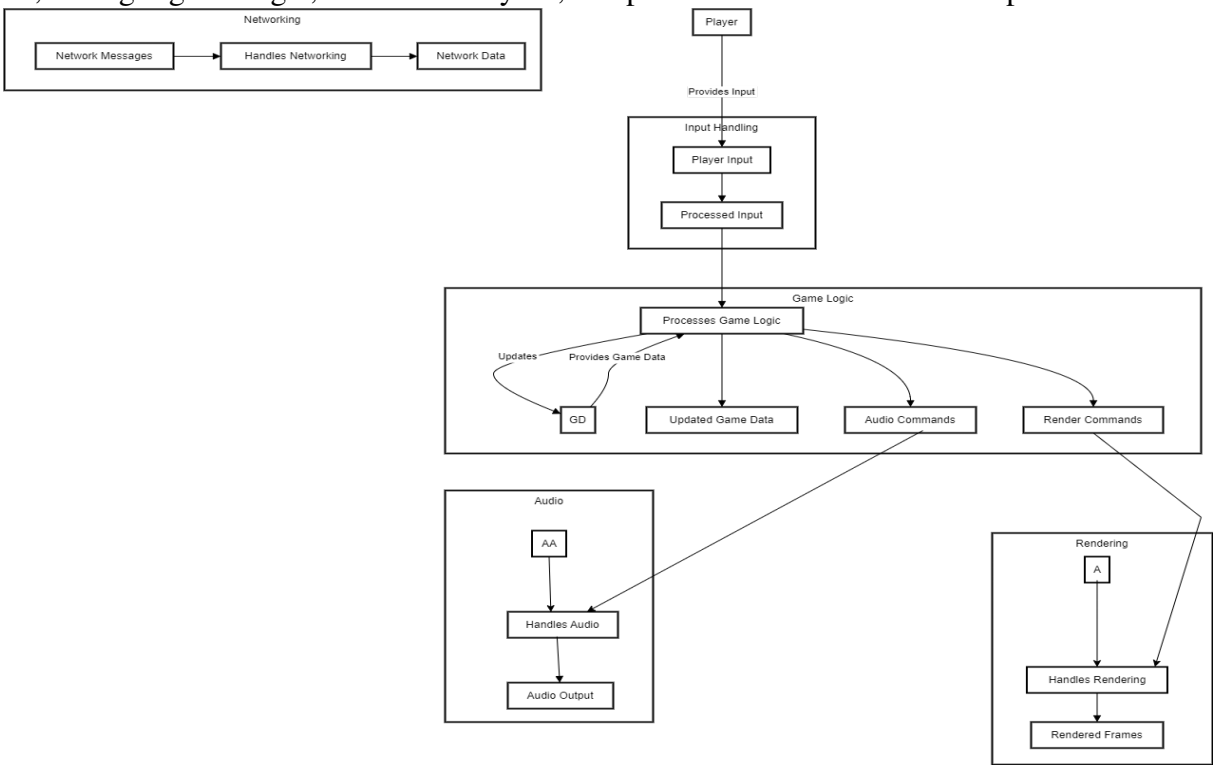
- **Player**: Provides input to the game system and receives game output.
- **Game System**: Processes player input, interacts with game data and assets, and produces game output.
- **Player Input**: Captures and processes input from the player.
- **Game Output**: The result of the game system processing, sent back to the player.
- **Game Data**: Stores and retrieves game-related data.
- **Assets**: Provides game assets like models, textures, and audio.

4.6.2 Level 1 Data Flow Diagram (DFD)

A Level 1 Data Flow Diagram (DFD) for an action shooting game represents a detailed view of the system. It breaks down the single process from the Level 0 DFD into several sub-processes and shows data flows between them.

- **Player Input Handling**: Captures and processes inputs from the player, such as movement commands and shooting actions.
- **Game Logic**: Manages the core mechanics of the game, including player movements, shooting mechanics, collision detection, and health management.
- **AI System**: Manages the behavior of enemies, including their movements, attacks, and decision-making.
- **Rendering Engine**: Handles the visual output of the game, rendering the game world, characters, and effects based on the current game state.
- **Audio Engine**: Manages audio playback, including background music, sound effects, and voiceovers.
- **Game Data Management**: Stores and retrieves game-related data, such as player progress, scores, and game state.

The Level 1 DFD provides a detailed view of how the game system processes player inputs, manages game logic, handles enemy AI, and produces visual and audio outputs.



Explaination

- **Player (P)**: Provides input to the system.

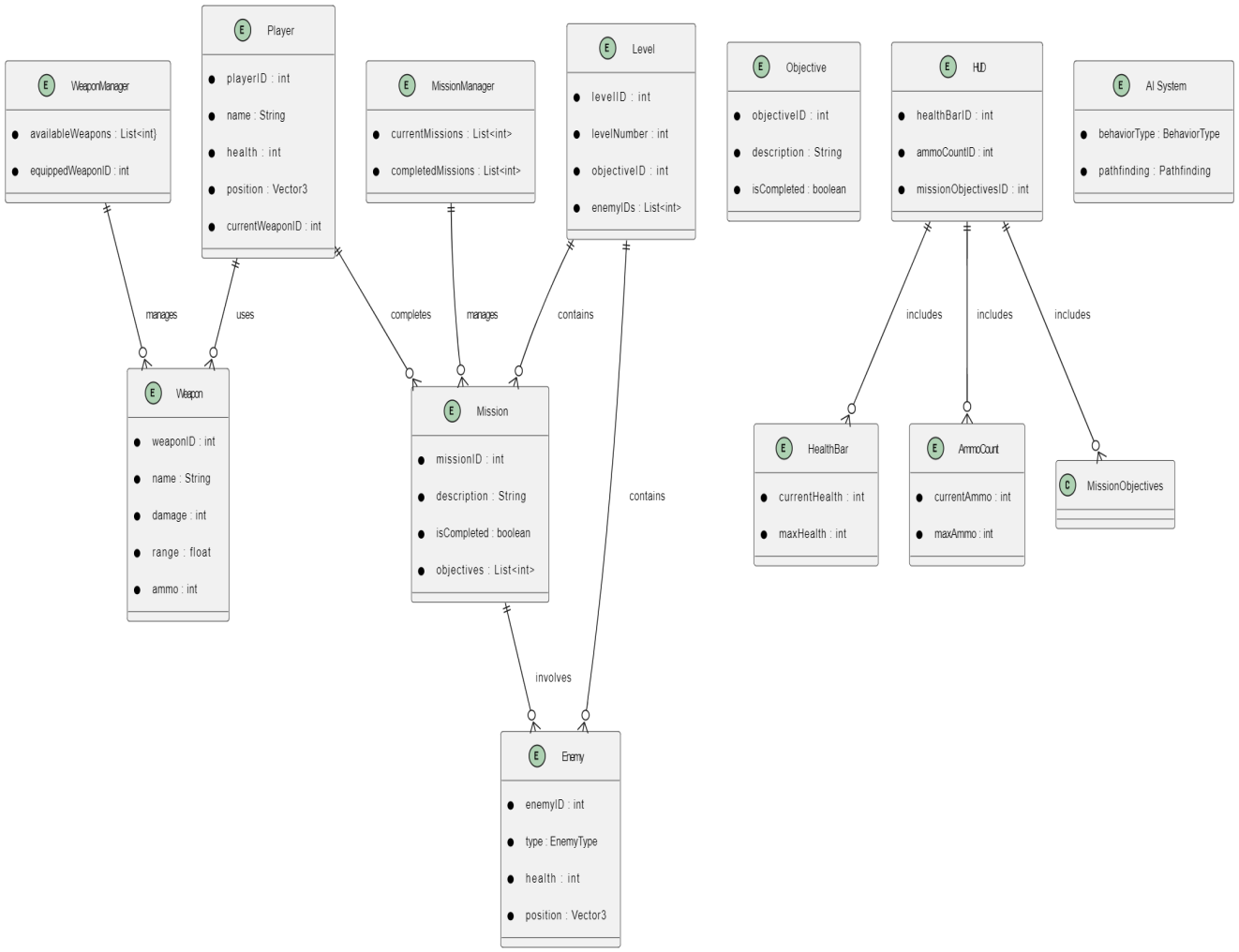
- **Input Handling (IH)**: Captures and processes player input.
- **Player Input (PI)**: Raw input from the player.
- **Processed Input (PIn)**: Input after processing, sent to Game Logic.
- **Game Logic (GL)**: Core logic of the game.
- **Processes Game Logic (GLP)**: Handles game mechanics and logic.
- **Game Data (GD)**: Stores game state data, both read and written by Game Logic.
- **Updated Game Data (UGD)**: Updated state of the game.
- **Render Commands (RC)**: Instructions for rendering the game visuals.
- **Audio Commands (AC)**: Instructions for game audio.
- **Rendering (R)**: Manages the visual output of the game.
- **Handles Rendering (RP)**: Renders frames using provided assets.
- **Rendered Frames (RF)**: Output frames to be displayed.
- **Assets (A)**: Visual assets like models and textures.
- **Audio (AU)**: Manages the audio output of the game.
- **Handles Audio (AP)**: Processes audio commands and assets.
- **Audio Output (AO)**: Output sound to be played.
- **Audio Assets (AA)**: Audio files and sound effects.

Networking Component in DFD

The Networking section in the Data Flow Diagram (DFD) is included as a placeholder for potential future development. While the current version of the game is designed for offline play.

4.7 Data Design

Data design involves structuring and organizing data to efficiently store, manage, and retrieve it within a system. The data design diagram focuses on how the data is structured and managed.



4.7.1 Major Business Entities

Player: Represents the main character. This entity is central to the game, tracking important attributes like health, position, and equipped weapon.

Weapon: Represents the tools player uses to attack enemies.

Enemy: Represents the adversaries the player encounters. Enemies have attributes like type, health, and position, determining their behavior and how challenging they are for the player.

Level: Represents the different stages or environments in the game. Each level contains objectives and enemies, and its structure impacts the game's progression.

Objective: Represents specific goals the player must achieve within a level or mission.

HUD (Heads-Up Display): Represents the on-screen display elements that provide the player with essential information like health, ammo, and mission status.

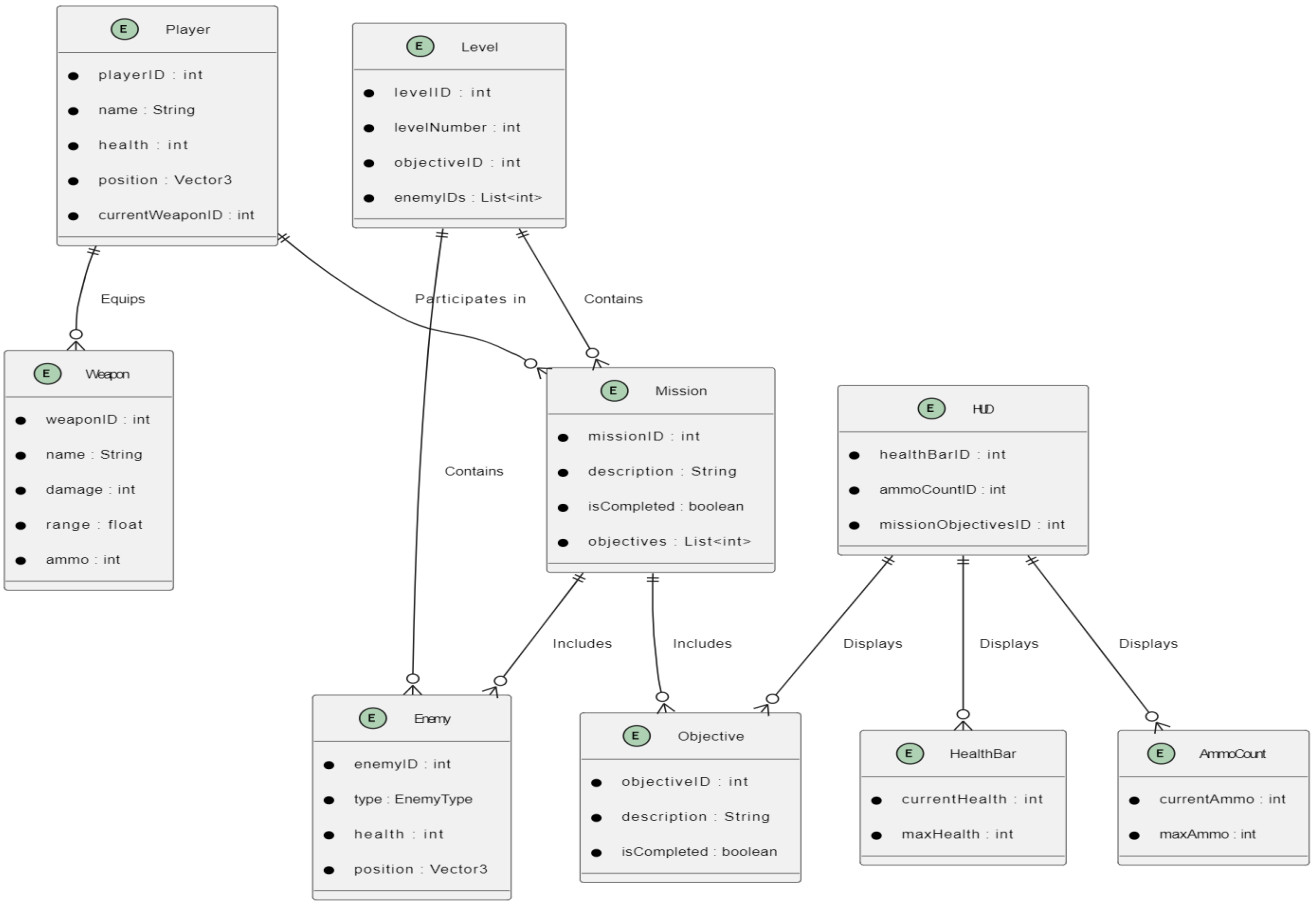
HealthBar A component of the HUD that shows the player's current health.

AmmoCount: Another component of the HUD that displays the current ammo count.

AI System: Represents the system controlling enemy behavior and navigation.

4.7.2 Entity Relationship Diagram

The Entity-Relationship Diagram (ERD) for the game illustrates the main components and how they interact with each other.



Explanation:

Entities and Attributes:

- **Player**: Represents the player with attributes for ID, name, health, position, and currently equipped weapon.
- **Weapon**: Represents weapons with attributes for ID, name, damage, range, and ammo count.

- **Enemy**: Represents enemies with attributes for ID, type, health, and position.
- **Level**: Represents game levels with attributes for ID, level number, objectives, and list of enemies.
- **Objective**: Represents mission objectives with ID, description, and completion status.
- **Mission**: Represents missions with ID, description, completion status, and associated objectives.
- **HUD**: Represents the Heads-Up Display with IDs for health bar, ammo count, and mission objectives.
- **HealthBar and AmmoCount**: Display current and maximum health and ammo, respectively.

Relationships:

- **Player - Weapon**: Players equip weapons.
- **Player - Mission**: Players participate in missions.
- **Mission - Objective**: Missions include objectives.
- **Level - Mission**: Levels contain missions.
- **Level - Enemy**: Levels contain enemies.
- **Mission - Enemy**: Missions involve enemies.
- **HUD - HealthBar**: HUD displays health bar information.
- **HUD - AmmoCount**: HUD displays ammo count information.
- **HUD - Objective**: HUD displays mission objectives.

4.7.3 User-Interface Design

UI Design Decisions

A minimalist approach was used to design the game UI and just implement the main features to make it functional. The color scheme of the game was set to be a light color so that it can be more adjusted to the user's eyes. The UI was completed implemented from the unity asset store and no tools such as Figma or Adobe Packages were used for designing the interface.

UI Implementation

The UI was implemented using the the unity game engine. Below are some of the screenshots that can be referenced for more details. include figures

Chapter 5

Implementation and Testing

5.0.1 Implementation Environment

Programming Tools:

Unity Hub & Unity Engine were used as primary development tools for the game, where game assets, scenes, physics and animations were handled.Visual Studio Community version was employed for scripting and debugging C# scripts. The Unity Asset Store leveraged for acquiring free assets, such as character models, environments, and particle effects to enrich the game's visuals and functionalities.

Repository Setup & Version Control:

Git & GitHub was used as a version control system to track changes and maintain back-ups of the project.

Integration:

The game integrated multiple components like 3D models from the Asset Store, sound assets, and physics libraries within Unity. Testing & Debugging was performed through Visual Studio and Unity's built-in play testing features. In Unity, the Play Mode allows you to run the game within the Unity Editor itself. This lets you test the game in real time without needing to build and deploy it to a separate platform.

Team Development Tools

To make a plan and for easy management of the project, Trello was used.

5.0.2 High Level Description Major Program Modules:

The high-level program modules consist of the main classes that play a crucial role in the core functionality of the system. These key classes manage essential operations such as game logic, user interaction, and overall flow. While there are additional supporting classes that handle specific tasks (e.g., utilities, minor features)..

5.0.3 PlayerController.cs

- **Description:** This module handles player input, movements, and interactions with the environment. It defines how the character moves in response to user input and interacts with in-game objects.
- **Data Tables:** Keeps track of player stats such as health, position, inventory, and weapons.
- **Error Handling:** Includes checks for invalid inputs and collision detection to avoid bugs related to character movement.

5.0.4 EnemyAI.cs

- **Description:** Controls enemy behavior, including patrol routes, detection of the player, and attack patterns. The enemy reacts when the player enters its detection zone.
- **Data Tables:** Enemy health, position, and state (idle, attack, patrol).
- **Error Handling:** Implements fail-safes to reset enemy states if unexpected behavior occurs, such as getting stuck in the environment.

InventorySystem.cs

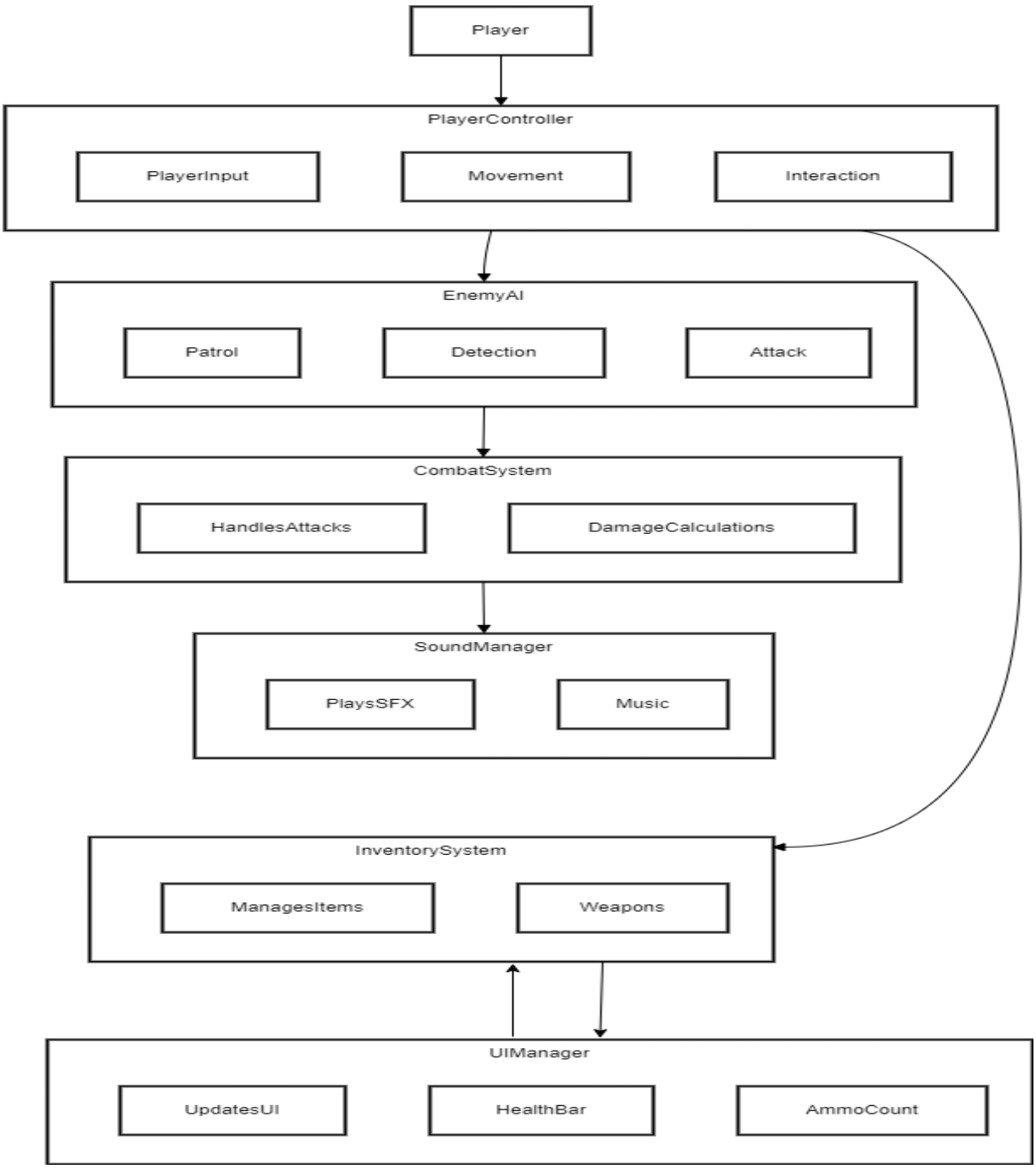
- **Description:** Manages the player's inventory, including adding, removing, and using items. Links the player's interactions with the in-game shop.
- **Data Tables:** Item IDs, quantities, and descriptions.
- **Error Handling**: Prevents overflow (carrying more items than allowed) and invalid item usages.

5.0.5 UIManager.cs

- **Description**: Manages the UI elements like the HUD, menus, and in-game notifications. Ensures that UI elements update dynamically during gameplay (health bars, ammo count).
- **Data Tables:** UI elements' visibility, text, and buttons.
- **Error Handling:** Detects and fixes null references in UI elements to prevent crashes.

5.0.6 Design of Implemented Module Structure:

Below is the diagram for Module Structure



PlayerController:

- Manages user input, character movement, and player interactions with the environment.
- Links to `InventorySystem` to update inventory when picking up or using items.
- Also links to `UIManager` to ensure the user interface reflects player status (e.g., health, ammo).

InventorySystem:

- Manages items and weapons.
- Communicates with `UIManager` to display the correct items and quantities on the screen.

UIManager:

- Updates the user interface dynamically based on player actions and status changes.

EnemyAI:

- Handles enemy movement, patrol behavior, and attacks.
- Communicates with the `CombatSystem` to calculate and apply damage when the player and enemy interact.

CombatSystem:

- Coordinates attack logic and damage calculations between the player and enemies.

SoundManager:

- Plays sound effects (SFX) and music based on game events such as attacks, item pickups, or changes in player state.

5.0.7 Test Strategies

The testing approach employed during the game development process was centred on both automated and manual testing in order to guarantee seamless gaming, error-free operation, and an intuitive user experience. Important phases of testing were:

- **Unit Testing:** Every essential feature, such as inventory management, adversary AI behaviour, and player controls, was tested separately. The behaviour of scripts such as PlayerController.cs and EnemyAI.cs, which handle erroneous input, boundary conditions, and stress testing, was verified by testing.
- **Integration Testing:** Each module's ability to function alone was checked, and then it was examined how well it interacted with the others (for example, connecting the PlayerController to the InventorySystem and UIManager).
- **Play Testing (Manual Testing):** Frequent playtests were carried out to find flaws that could only be discovered during runtime, difficulty balancing, and unexpected enemy AI behaviours. Playtests also paid attention to the user experience.
- **Performance Testing:** Load testing was done to ensure the game performed well on a range of devices, testing for frame rate drops, memory usage, and lag. The game's performance was checked with higher numbers of enemies, effects, and environmental details to ensure it could handle complex scenarios without performance degradation.
- **Regression Testing:** Every time a new feature or bug fix was introduced, regression testing ensured that previously working features were still functioning as intended. The version control system (GitHub) was used to track and merge changes, helping avoid conflicts that could introduce bugs. Several times Git has helped to travel back in time and fix issues in code.

5.0.8 White Box Testing

Unit Testing the Individual Classes and Methods:

Software testing that involves testing individual program modules or components separately is known as unit testing [1]. A function, method, or class is the smallest tested component that makes up a ”unit” in an application. Unit testing makes sure that every single portion of the code runs correctly without interference from outside sources.

- **AI.cs Unit Test:**
  AI behavior based on conditions (e.g., proximity to the player, attack conditions).

```
[Test]
public void Test_AI_Movement()
{
  // Arrange
  AI ai = new AI();
  ai.followSnds = new AudioClip[1]; // Assuming AI has
     audio clips for movement sound
  // Act
  ai.Move(); // Call the movement function
  // Assert
  // Add assertions depending on the conditions of Move,
     like if it moves to the correct destination
}
```

- **CharacterDamage.cs Unit Test:**
  This script involves handling damage to NPCs and tracking hit points. Important methods of this class include taking damage and removing the character upon death.
- **TakeDamage():** How the character’s hit points are reduced.
- **Death event():** Whether the body is removed when health reaches zero.

```
    [Test]
    public void Test_Character_TakesDamage()
    {
      // Arrange
      CharacterDamage character = new CharacterDamage();
      character.hitPoints = 100f;
      // Act
      character.TakeDamage(30f); // Simulate 30 points of
         damage
      // Assert
      Assert.AreEqual(70f, character.hitPoints,
         "Character should have 70 hit points left after
         taking 30 damage.");
    }
    [Test]
    public void Test_Character_Dies_When_Health_Depletes()
```

```
    {
      // Arrange
      CharacterDamage character = new CharacterDamage();
      character.hitPoints = 50f;
      // Act
      character.TakeDamage(60f); // Damage exceeds
         current hit points
      // Assert
      Assert.AreEqual(0f, character.hitPoints, "Character
         hit points should not go below zero.");
      // Additional checks for death behavior can be
         added, such as checking if the onDie event is
         called.
    }
```

- **MoveTrigger.cs Unit Test:**
  This script manages NPC movement triggers. It detects when an NPC (Non-Player Character) should move or rotate towards a target.

```
    [Test]
    public void Test_NPC_Moves_On_Trigger()
    {  // Arrange
      MoveTrigger moveTrigger = new MoveTrigger();
      NPC npc = new NPC(); // Assuming an NPC class
         exists and npcToMove refers to it
      moveTrigger.npcToMove = npc;
      // Act
      moveTrigger.OnTriggerEnter(); // Simulate entering
         a movement trigger
      // Assert
      Assert.IsTrue(npc.followed, "NPC should follow the
         player when the move trigger is activated.");
    }
```

- **RemoveBody.cs Unit Testing:**
  This script is responsible for removing NPC bodies after a certain time (bodyStayTime) and removal of weapons.

```
  [Test]
  public void Test_Body_Removed_After_Delay()
  {  // Arrange
    RemoveBody removeBody = new RemoveBody();
    removeBody.bodyStayTime = 5f; // Set stay time to 5
       seconds for test purposes
    GameObject body = new GameObject();
    // Act
    removeBody.Start(); // Start timer
    removeBody.FixedUpdate(); // Simulate time passing
    // Assert
```

```
    // Check if the body is removed after the specified
       delay (this would require simulating time)
}
```

5.0.9 Integration Testing

In integration testing we check how the code for different functionalities integrates with other functionalities. After manual testing of each and every feature the project is working fully and the code is free from any large scale bugs.

- **Full Integration Test: Combining AI, CharacterDamage, and RemoveBody** This comprehensive test checks the full interaction: AI attacking the player, the player taking damage and dying, and the body being removed.

```
using NUnit.Framework;
using UnityEngine;
using System.Collections;
public class FullIntegrationTest : MonoBehaviour
{  private AI ai;
  private CharacterDamage player;
  private RemoveBody removeBody;
  [SetUp]
  public void Setup()
  {  // Setup AI, player, and remove body in the scene
    GameObject aiObject = new GameObject();
    ai = aiObject.AddComponent<AI>();
    GameObject playerObject = new GameObject();
    player =
       playerObject.AddComponent<CharacterDamage>();
    removeBody =
       playerObject.AddComponent<RemoveBody>();
    // Assign player to AI and configure settings
    ai.player = player;
    ai.playerProximity = 5f; // AI will attack within
       5 units
    player.hitPoints = 100f; // Player starts with
       100 hit points
    removeBody.bodyStayTime = 2f; // Body will be
       removed after 2 seconds
  }
  [UnityTest] // Use UnityTest for time-based
     operations
  public IEnumerator
     Test_AI_Attacks_Player_And_Body_Removed()
  {  // Simulate AI getting close to the player and
     attacking
    ai.transform.position = new Vector3(0, 0, 0);
    player.transform.position = new Vector3(3, 0, 0);
       // Player within attack range
```

```
        ai.Update(); // Update AI to check proximity and
            attack
        // Assert: Player took damage
        Assert.Less(player.hitPoints, 100f, "Player
            should take damage from AI attack.");
        // Simulate player dying
        player.TakeDamage(150f); // Lethal damage to
            player
        // Wait for bodyStayTime to pass
        yield return new
            WaitForSeconds(removeBody.bodyStayTime);
        // Assert: Player's body is removed
        Assert.IsNull(player.gameObject, "Player's body
            should be removed after bodyStayTime.");
    }
}
```

- **Explanation** To make sure that various game elements—such as AI, player health, and movement triggers—interact as intended during actual game play, Unity integration testing is essential. The tests offered confirm how certain parts function together:

  1. **AI attacking the player:** AI that attacks the player makes sure that damage-dealing and proximity detection work as they should.
  2. **Character death and body removal:** Game realism is increased by character death and body removal tests that determine whether the body object is properly removed after a predetermined amount of time.
  3. **NPC movement triggers:** NPC movement triggers verify whether the NPC responds to triggers correctly by moving to designated positions.
  4. **Full Integration Test:** A full integration test simulates every step of the gaming flow, from AI attack to player death and body removal, by combining several different components.
     These integration tests guarantee that the game mechanics function as a cohesive unit by covering crucial script-to-script interactions. These tests are adaptable to the unique gameplay situations and game features.

5.0.10 Black Box Testing

Black box testing is a type of software testing where the tester assesses an application's functioning without looking inside its core components or operations.

Techniques Used in Black Box Testing:

- **Equivalence Partitioning:** Equivalence Partitioning splits input data into segments that are valid and invalid in order to minimize the number of test cases.
- **Boundary Value Analysis:** Boundary Value Analysis tests the limits of the partitions, including the points directly below, at, and above the boundaries.

- **Decision Table Testing:** Decision Table Testing represents combinations of inputs and their matching outputs using a table.
- **State Transition Testing:** State Transition Testing evaluates the system's ability to change states in response to input circumstances.
- **Use Case Testing:** Use Case Testing uses use cases and user scenarios to validate the functionality.

Test Case ID	Component	Input	Expected Output	Result
TC1	AI.cs	Player is within proximity of AI	AI should attack the player and reduce the player's hit points.	Pass
TC2	AI.cs	Player is outside AI proximity	AI should not attack the player, and player's hit points remain unchanged.	Pass
TC3	CharacterDamage.cs	Player takes 30 damage when hit	Player's hit points should reduce by 30 from the initial value.	Pass
TC4	CharacterDamage.cs	Player takes damage exceeding health	Player's health should not drop below zero, and the death event should trigger.	Pass
TC5	CharacterDamage.cs	Player is dealt no damage	Player's hit points should remain the same.	Pass
TC6	MoveTrigger.cs	NPC enters trigger zone	NPC should move to the designated location defined in the `movePosition`.	Pass
TC7	MoveTrigger.cs	NPC does not enter trigger zone	NPC should remain at its original position.	Pass
TC8	RemoveBody.cs	Character dies and `bodyStayTime` is set	Character's body should be removed after the `bodyStayTime` passes.	Pass
TC9	RemoveBody.cs	`bodyStayTime` set to 0	Character's body should be removed immediately after death.	Pass
TC10	RemoveBody.cs	Player remains alive	Character's body should not be removed as the player is still alive.	Pass
TC11	AI + CharacterDamage (Integration Test)	Player is within AI range, and AI attacks	Player's hit points should reduce, and after enough damage, the player should die.	Pass
TC12	AI + MoveTrigger (Integration Test)	AI reaches target position via trigger zone	AI should correctly navigate and stop at the target position defined by the trigger.	Pass
TC13	Full Integration (AI + CharacterDamage + RemoveBody)	Player is killed by AI, `bodyStayTime` triggers	Player's health should drop to zero, triggering death, and the body should be removed after a delay.	Pass

Table 5.1: Black-box testing results for game scripts

5.0.11 Usability Report For Proximity Assault

1. **Title Page‘**
   - **Title:** Usability Report for Proximity Assault
   - **Project:** First-Person Shooting Game
   - **Prepared By:** Muhammad Atif
2. **Executive Summary** The purpose of this usability test was to assess the player's experience in the game, in which terrorists have taken over a facility and are getting ready to produce nuclear weapons. The player's goal is to use a variety of weaponry to destroy every enemy so that a group of scientists can disassemble the nuclear apparatus.
3. **Introduction**
   - **Purpose:** The goal of the test was to find problems with the controls, navigation, and degree of difficulty of the gaming experience.
   - **Scope:** The first two stages of the game, the weapon swapping mechanism, enemy encounters, and mission objectives.
4. **Methodology**
   - **Participants:** 4 participants, aged 18-25 with a mix of gaming experience.
   - **Environment:** The test was conducted on the users phones.
   - **Task:** The user navigate through the scenes, switched weapons and tried different weapons.
   - **Data Collection:** Data was collected through screen recording.

Table 5.2: Task Completion Rates and Findings

Task	Success Rate	Average Time on Task	Error Rate
Navigate through the facility	90%	4 minutes	5%
Switch weapons and eliminate enemies	75%	6 minutes	20%

5. **Results**

    - **Task Completion Rate**
    - **Usability Issues:**
    - **Weapon Switching Difficulty:**
        - 60% of players found it difficult to switch between weapons quickly.
        - **Feedback:** "I couldn't switch weapons fast enough to keep up with the enemies. It was frustrating, especially during intense battles."
    - **Enemy AI Behavior:**
        - Players noted inconsistent enemy AI. Some enemies were too easy, while others felt overpowered.
        - **Feedback:** "The enemies seemed random in their difficulty. Some stood there doing nothing while others were nearly impossible to beat."
    - **Objective Clarity:**
        - 50% of players had difficulty understanding when and how to protect the scientists during the mission.
        - **Feedback:** "I wasn't sure what I was supposed to do after defeating the enemies."

6. **Analysis**

    - **Weapon Switching Mechanism:** The present weapon swapping system is clumsy and takes away from the intense fighting.
    - **Enemy AI:** Players became frustrated with the inconsistent behaviour of the enemies, which decreased their level of enjoyment with the game overall.
    - **Mission Objectives:** The lack of clarity in communicating objectives resulted in a decrease in task success rates and confusion.

7. **Recommendations**

    - **Improve Weapon Switching**
    - **Enhance AI Behavior**
    - **Clearer Mission Guidance**

8. **Conclusion** Overall, the results of the usability test indicated that while participants found the concept and ambiance of the game engaging, they found certain elements difficult to grasp, such as switching between weapons and mission objectives.

Chapter 6

Deployment

When deploying a unity project for Android and IOS the major deployment task are grouped into some categories below.

6.0.1 Major Deployment Tasks

1. **Project Setup and Configuration**
    - **Unit Project Setting:**
        - Set up the project in Unity's Build Settings for both iOS and Android.
        - Configure the bundle identifier, version number, orientation, etc.
        - Optimize the Graphics Settings for optimal performance
    - **SDK and Build Tools:**
        - For **Android**: Install Android **SDK, NDK,** and **JDK** via Unity Hub.
        - For **iOS**: Ensure **Xcode** is installed and up to date for building the game.
2. **Testing and Optimization**
    - **Mobile Specific Testing:**
        - Test the game on a variety of Android and iOS devices
        - Keep an eye on battery life, memory utilization, and frame rates.
        - Examine input handling, touch controls, and UI responsiveness.
    - **Performance Optimization:**
        - Optimize graphics
        - Implement Level of Detail (LOD) for 3D models to enhance performance.
        - Use Profiler and Memory Management tools in Unity to optimize code and improve loading time.
3. **Platform Specific Features Integration**
    - **Android:**
        - Implement the Google Play Services
        - Integrate with Google Play In-App Purchases

- **iOS:**
  - Implement the Game Center for achievements, leaderboards, cloud saves
  - Set up Apple In-App Purchase
  - Integrate with Apple's Push Notification Service

4. **App Store Compliance**
   - **Android (Google Play):**
     - Prepare the Google Play Console with all required game assets.
     - Ensure APK/Bundle meets Google Play guidelines.
     - Perform testing through Google Play's pre-launch report for testing.
   - **iOS (Apple App Store):**
     - Upload the iOS build through **Xcode** or **Transporter** to **App Store Connect**.
     - Ensure the app meets Apple's **App Store Guidelines**.
     - Prepare **App Store Connect** with all required assets.

5. **Distribution of the Product:**
   - **Android (Google Play):**
     - Generate **APK** or **AAB (Android App Bundle)** using Unity's build options.
     - Enable **64-bit support** (required by Google Play).
     - Sign the APK or AAB with your **release key** for deployment on Google Play.
     - Set up **beta testing** via the **Google Play Beta Testing** program for feedback before full release.
   - **iOS (Apple App Store):**
     - Build and export the game to an **IPA** file through Xcode.
     - Ensure the app is signed with the appropriate **provisioning profiles** and **certificates**.
     - Set up **TestFlight** to beta test the game and gather feedback before public release.

6. **Analytics, Ads, and Monetization Setup**
   - Integrate **Unity Analytics, Google Analytics**, or a third-party service to track user behavior, retention, and engagement.
   - Implement In-App Purchases (IAP) and set up necessary stores (Google Play and Apple App Store) for microtransactions.
   - Ensure ads and monetization strategies comply with both Google Play and App Store guidelines.

6.0.2 Deployment Diagram

The deployment diagram illustrates the architecture of the Proximity Assault for Android and iOS platforms. The game application includes essential features such as a local inventory system, an in-game AI system, a dummy shop system, and two levels that are stored locally on the device. Future enhancements include the integration of an external server for the shop system to facilitate transactions and Firebase for tracking user data and analytic. Currently, the game does not have cloud-based features or multiplayer support, as depicted in the diagram.

Bibliography

[1] FOWLER, M. Unit test. https://martinfowler.com/bliki/UnitTest.html, 2021. Accessed: 2024-09-17.

[2] HOCKING, J. *Unity in action: multiplatform game development in C*. Simon and Schuster, 2022.

[3] KLEPPMANN, M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc., Sebastopol, CA, 2017.

[4] SINGH, G., AND VANGAVOLU, V. V. S. S. R. Effect of resolution on player performance and experience in virtual reality low-fidelity first-person shooting games, 2023.