

ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

Σχολή Μηχανικών Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Πρόγραμμα Σπουδών – Μηχανικών Πληροφορικής Τ.Ε.

THESIS

Creation and design of a multiplayer game using Unity 3D

Δημιουργία και σχεδίαση παιχνιδιού πολλών παικτών με τη χρήση Unity 3D

GKAIDATZIS NIKOLAOS - AM4707

KOSTANTINOS KONTOS - AM4738

Επιβλέπων Καθηγητής: Παχουλάκης Ιωάννης

JULY 2024

Acknowledgments

We would like to express our thanks to our professor, Dr. Pachoulakis Ioannis, for not only accepting our thesis but also providing guidance and insights during its finalization. His expertise and support have been instrumental, especially in the development of our game. Additionally, we extend our deepest gratitude to our families, whose support has been significant not just through the writing of this paper, but throughout our entire university journey. Their encouragement and belief in us have been vital in reaching this milestone.

Abstract

This thesis presents the process of designing and developing 'Avalon's Conviction,' a multiplayer game taking advantage of the capabilities of Unity 3D. Drawing inspiration from classic strategy and deception games like 'Town of Salem,' 'Mafia,' and 'Werewolf,' this project emphasizes creating an engaging multiplayer experience centered around the mechanics of the role assigned to the player like ‘Vote, Heal, Kill’. The game's dynamic environment, influenced significantly by player actions, features an evolving user interface that reflects various times of the day, aligning with the game's thematic elements.Central to the development process was the implementation of Unity's advanced features, including Netcode for GameObjects for seamless multiplayer interactions, Unity's Lobby Manager and Relay Services for efficient game session management, and Plastic SCM for version control and collaborative development. The thesis provides a detailed account of the game design, highlighting the strategic depth and immersive aspects of the gameplay. Additionally, it discusses the technical challenges and solutions encountered in crafting a multiplayer game environment that is both visually appealing and functionally resilient.

1Introduction....3
1.1Summary of the Game. 4
1.2Motivation to Conduct the Project. 4
1.3Purpose and Objective of the Project. 5
2Concepts and Technologies....6
2.1What is Game Development....6
2.2What is Game Design 6
2.3Basic Principles of Game Design 6
2.4What is a Social Deduction and Strategy Game....7
2.5What is a Game Engine?....8
2.6What is Unity Game Engine 8
2.6.1Unity Tools 9
2.6.2Game Objects....9
2.6.3Prefabs....9
2.6.4Scriptable Objects 10
2.6.5Programming Patters and Principles 10
2.6.5.1Design Patterns 11
2.6.5.2Creational Patterns....11
2.6.5.2.1Structural Patterns 11
2.6.5.2.2Behavioral Patterns....11
2.6.5.3SOLID Principles....11
2.6.5.4Other Important Principles and Patterns....11
2.6.6Unity Services....12
2.6.6.1Netcode for GameObjects 12
2.6.6.2Lobbies....12
2.6.6.3Relay 13
2.6.6.4Unity DevOps....13
2.6.6.5Plastic SCM....14
2.6.6.6Player Authentication 14
2.6.6.7Cloud Save....14
2.6.6.8Player Management 15
3Implementation Methodologre....15
3.1Version Control 16
3.2Workflow Diagram 16
4Game Design....20

4.1Login Scene 20
4.1.1Error Panel 20
4.1.2Loading Panel 20
4.2Main Menu Scene 20
4.3Lobby Scene. 21
4.3.1Host Game Panel. 22
4.3.2Lobby List Panel 22
4.3.3In Lobby Panel 22
4.3.4Join Lobby with Code Panel 22
4.4Game Scene 22
4.4.1Fortune Wheel Panel 23
4.4.2In Game Panel 23
4.4.3Judgement Panel 23
4.4.4Announcement Panel 23
5Game Development – Code Implementation 24
5.1Core Utility Scripts 24
5.1.1Loader 24
5.1.2Read Only Attribute and Drawer 24
5.2Login 25
5.2.1Authentication Manager 26
5.2.2Cloud Save Manager 26
5.2.3Loading Manager 26
5.2.3.1Loading UI 27
5.2.4Error Manager 27
5.2.4.1Error UI 28
5.3Main Menu 28
5.3.1Main Menu UI 29
5.3.2Main Menu Cleanup 29
5.4Lobby 29
5.4.1Lobby Manager 30
5.4.2Lobby List UI 30
5.4.3Lobby List Single UI 30
5.4.4Lobby Join Code UI 31
5.4.5Lobby Create UI 32
5.4.6Lobby Time Settings UI 33
5.4.7Lobby UI 33

5.4.8Lobby Single UI....34
5.5Game....35
5.5.1Player....35
5.5.2Role....35
5.5.3Role Manager....36
5.5.4Role Information....37
5.5.5Game Network....37
5.5.6Vote Manager....38
5.5.7Player Data....39
5.5.8Action Data....40
5.5.9Message Data....41
5.5.10Time Manager....41
5.5.11Game Manager....42
5.5.12Time State Manager....43
5.5.13Picker Wheel....44
5.6Achievements....44
5.6.1AchievementManager....45
5.6.2AchievementDisplay....46
5.6.3AchievementChecker....48
5.7Audio....48
5.7.1Audio in game....48

1 Introduction

1.1 Summary of the Game.

"Avalon's Conviction" is a dynamic 2D online multiplayer game that merges the intrigue of social deduction with strategic gameplay, inspired by classics like Werewolf and Town of Salem. Players are thrust into a suspense-filled world where survival hangs in balance from start to finish. At the game's onset, each participant is secretly assigned one of three role teams: Town-people, Assassins, or Neutral, each with distinct objectives. The Assassins aim to eliminate the Town-people, the Town-people strive to identify and eliminate the Assassins, and Neutral roles pursue their own unique goals, which may support or hinder the other factions.

The game progresses through rounds of night and day phases. At night, players use their roles' special abilities, such as protection or identity discovery, while Assassins have the opportunity to kill. The day phase reveals the identities of any players killed overnight and serves as a time for survivors to debate, accuse, and decide the fate of suspected Assassins through a vote. A guilty verdict results in the lynching of the accused, revealing their role. The game ends when one group fulfills their victory conditions, ensuring a continuous cycle of deduction, strategy, and suspense.

1.2 Motivation to Conduct the Project.

The genesis of our project stemmed from a desire to enhance our gaming experiences with friends, particularly around the social deduction game Werewolf. This game, akin to a board game, often led to debates over its rules due to the variations in understanding and interpretation among our group. Such disagreements frequently resulted in frustration and disengagement, with some friends opting to divert their attention to their phones and social media, rather than participating in the game. Recognizing the potential to unify our gaming experience and retain engagement, we were inspired to embark on creating a digital version of Werewolf. Our aim was to incorporate the foundational elements of the traditional game, along with influences from its online adaptation, Town of Salem. By doing so, we envisioned a platform where the rules were standardized, eliminating disputes over gameplay mechanics and fostering a more inclusive and focused gaming environment. This digital adaptation would not only streamline the experience but also rekindle the enthusiasm among our friends, ensuring everyone played by the same rules and remained fully engaged in the game, without the distractions of the outside digital world.

1.3 Purpose and Objective of the Project.

The primary aim of this project was to develop a digital incarnation of the classic game Werewolf, utilizing the Unity game engine as the foundation of our creation. A significant part of our endeavor was dedicated to mastering Unity and its programming language, ensuring we had a firm grasp of the tools and techniques necessary for game development. Additionally, an essential objective was to delve into the networking aspects of video game creation, a critical component for enabling multiplayer functionality in our game.

To address the networking requirements, we employed Unity's Netcode for GameObjects along with its Lobby Manager and Relay services. These tools were instrumental in facilitating the hosting of our game, ensuring players could connect and interact seamlessly in a shared online environment.

Another vital aspect of our project was the creation of the game's visual elements. We decided to craft the graphics content from the ground up, utilizing Gimp. This approach allowed us to tailor the game's aesthetic to our vision, ensuring a unique and engaging player experience. Through this project, our objective was not only to recreate the Werewolf game in a digital format but also to acquire a comprehensive understanding of game development processes, from programming and networking to graphic design, using industry-standard software and practices.

While compesating if we should use our own database and handle all the data ourselves, we came across Unity’s cloud services, which consists of many useful services which not only gives

2 Concepts and Technologies

2.1 What is Game Development

Game development is a multifaceted process that entails the creation of video games, a form of interactive entertainment. At its heart, game development is the art of combining various elements such as storytelling, design, and programming to create an engaging and immersive experience for players. Unlike other forms of media, games are interactive and allow players to engage directly with the content, making the development process unique and complex.

In the subsequent stages, programming and asset creation come into play. Programmers work on coding the game, bringing the mechanics and interactions to life, while artists and designers create the visual and audio elements. This phase often involves using specific software and tools, like game engines, which provide the necessary framework for building the game. Collaboration is key in this phase, as different expertise from programmers, artists, and designers come together to realize the vision of the game.

The final stages involve testing and refining the game to ensure it is free from bugs and provides a smooth user experience. This phase is critical as it ensures the quality and playability of the game. Once testing is complete, the game is launched and made available to the public. Post-release, developers may continue to update the game, fixing any issues and sometimes adding new content to keep the game engaging for its audience.

2.2 What is Game Design

Game design is the craft of creating the rules and structures that define a game. It involves conceptualizing the game's theme, story, characters, objectives, mechanics, and challenges, then bringing them together to create a compelling, engaging, and enjoyable experience for players. A well-designed game balances complexity and simplicity, ensuring it is easy to learn but hard to master, thereby keeping players engaged over time.

In the context of "Avalon's Conviction," game design would encompass the creation of the game's setting—a blend of myth and mystery inspired by classics like "Town of Salem," "Mafia," and "Werewolf." It involves devising strategic elements that encourage player interaction and decision-making, such as the actions to "Kill, Vote, Heal," which are central to gameplay. The design process also includes planning how these actions affect the game's outcome and player relationships, contributing to a dynamic and unpredictable game environment.

Moreover, game design for "Avalon's Conviction" involves creating a user interface (UI) that adapts to different phases of the game, enhancing the player's immersion and interaction with the game world. It also includes integrating Netcode for GameObjects for multiplayer functionality, ensuring a smooth and responsive online experience, and using Unity's Lobby Manager and Relay Services to facilitate player matchmaking and connectivity.

In summary, game design is a multifaceted discipline that combines creativity, technical skill, and a deep understanding of player psychology to create memorable and engaging gaming experiences. For "Avalon's Conviction," this means crafting a game that captivates players with its strategic depth, narrative intrigue, and social dynamics, making every playthrough a unique adventure.

2.3 Basic Principles of Game Design

To discuss the basic principles of game design, it's crucial to understand that game design is the craft of applying rules, mechanics, and story to create an engaging and enjoyable experience for players. Here are some key principles that guide game designers in creating compelling games:

- Player-Centered Design: The player's experience is at the heart of game design. Designers focus on making games enjoyable, engaging, and accessible to their target audience. This involves considering player feedback, playtesting, and adjusting the game to ensure it meets the players' needs and expectations.
- Clear Objectives: Games need to have clear, understandable objectives that guide players on what they need to achieve. These objectives give players direction and a sense of purpose, whether it's reaching the end of a level, defeating a boss, or solving a puzzle
- Balanced Challenge: A well-designed game offers a balanced level of difficulty that is neither too hard nor too easy for its target audience. This balance helps keep players engaged and motivated to progress through the game. Designers often use adaptive difficulty, challenges that scale with the player's skill level, to maintain this balance.
- Feedback and Rewards: Providing players with immediate feedback and rewards for their actions is crucial. Feedback can be visual, auditory, or through game mechanics, and it helps players understand the consequences of their actions. Rewards, such as points, new abilities, or story progression, motivate players to continue playing.
- Interesting Choices and Strategy: Games should offer players meaningful choices that affect the game's outcome, encouraging strategic thinking and planning. These choices can create a richer gaming experience and increase the game's replay value.
- Narrative and World-Building: A compelling story and a well-developed game world can greatly enhance the player's experience. Narrative and world-building provide context to the players' actions and make the game more immersive.
- Simple and Intuitive Controls: Game controls should be easy to learn but hard to master. Complex controls can be a barrier to entry for new players, while intuitive controls can make the game more accessible
- Aesthetic Consistency: The game's visuals, sounds, and overall aesthetics should be consistent and support the game's theme and mood. This consistency helps in creating an immersive world that players can get lost in.
- Playtesting: Regular playtesting with target players is essential to identify issues and opportunities for improvement. Playtesting provides valuable feedback that can be used to refine the game design.
- Innovation and Creativity: While it's important to follow proven design principles, innovation and creativity can set a game apart from others. Exploring new mechanics, stories, and designs can lead to unique and memorable gaming experiences.

By adhering to these principles, game designers can create engaging, enjoyable, and memorable games that stand the test of time.

2.4 What is a Social Deduction and Strategy Game

Social deduction and strategy games are distinguished by their dynamic social interactions, where communication, bluffing, and negotiation play pivotal roles. These games createtense experiences that combine logical problem-solving. The social aspect of these games can lead to the formation of temporary alliances or rivalries, adding layers of complexity and depth to the gameplay. This genre appeals to a wide range of players, offering a rich and varied experience that can be both intellectually challenging and emotionally engaging, making it a popular choice for those who enjoy games that require both strategic thought and social interaction. Strategically, these games demand a high level of planning and decision-making. This involves not just a tactical understanding of the game's rules but also an ability to predict and manipulate opponents' actions. Strategy in these games is multifaceted, encompassing the management of resources, timing of actions, and psychological warfare. Players are often required to adapt their strategies in response to the evolving game state and the actions of other players.

2.5 What is a Game Engine?

A game engine is a software framework used for the development and creation of video games. It provides a comprehensive set of tools that developers use to write and manage game code, create graphics, and integrate sound. Essentially, a game engine acts as the foundational architecture for a game, enabling developers to focus more on the creative aspects of game development without getting bogged down by the technical intricacies of programming every small detail from scratch.

Game engines typically consist of several key components. These include a rendering engine for 2D or 3D graphics, a physics engine that simulates the laws of physics realistically within the game, and audio features for sound management. Additionally, most game engines provide tools for animation, artificial intelligence, networking, and memory management. These integrated components work together seamlessly, allowing for the efficient creation of complex and interactive gaming worlds.

One of the primary advantages of using a game engine is its ability to significantly streamline the game development process. By offering pre-built modules and reusable components, game engines reduce the amount of time and effort required to develop basic game functionalities. This efficiency allows developers, especially smaller teams or individual creators, to produce games more quickly and with fewer resources. Moreover, game engines often come with user-friendly interfaces and support scripting in common programming languages, making them accessible to a wide range of users with varying skill levels.

Game engines can be used to develop a variety of game types, from 2D to 3D adventures. Some engines can be cross-platform, meaning they allow for the development of games that can run on multiple systems as PC, consoles, and mobile devices. This gives opportunities for game developers to bring their creative visions to life and reach a diverse audience.

2.6 What is Unity Game Engine

Unity is a popular game engine that was released in 2005. Unity Technologies was the team that developed Unity and it is known for being versatile, meaning it can be used for making not just games but also other interactive simulations and even training programs. You can create both 2D and 3D games not only for a PC but for smartphones and consoles alike.

Many Developers prefer Unity due to its user-friendly interface and its simplicity. The primary programming language is C#, but it can also support a variety of programming languages. This makes Unity a suitable option for both beginners and experienced developers. There are a ton of tutorials on the web due to its popularity and a lot of tools as well. Another advantage is that Unity offers certifications in using the engine for game development or programming and it is one of few recognized credentials available for game development in general.

2.6.1 Unity Tools

Having many options to choose from Unity's marketplace, we used not only those that would help us with the actual development of the game, but with the version control of the project so we would have a good collaborative process. For most of our UI elements we used, the TextMeshPRO which helps with the creation of rich, detailed, and highly customizable text elements. Some of the objects we created are the buttons used for interacting and executing click-based events and input for the name of the user. Another tool we used is BG tools, which give the option to modify the playerprefs of the user from the Unity editor which helps with the fast testing of features which use those values. Lastly, for our version control we used plastic SCM which is a Version Control System which was designed to handle large binary files and large-scale projects efficiently.

2.6.2 Game Objects

Game Objects in Unity serve as the building blocks for creating game environments, housing components that determine their look and behavior. Each Game Object comes with a "Transform" component to set its position, rotation and size in the scene. Developers can customize Game Objects by adding components such as visuals like meshes or sprites well as functional scripts to design characters, props, cameras and user interface elements. These Game Objects can be structured hierarchically to establish parent child connections. Can be saved as Prefabs" for efficiency. Tags and layers play a role in organizing and managing Game Objects, within Unity projects enhancing their utility in crafting scenes.

2.6.3 Prefabs

In Unity a prefab is a flexible tool that allows developers to make set up and save a template of Game Objects with specific components and property values. Once you create a prefab you can use it several times in different or even in the same scene making it easier to replicate complex objects. This feature simplifies and helps with the development time, by reusing Game Objects but also automatically applies any changes made to the prefab to all instances making it effortless to update and manage game elements. We created many prefabs which were used for different purposes, from the achievements which pop up when the Player meets specific conditions to our role wheel picker that shows the role assignment visually.

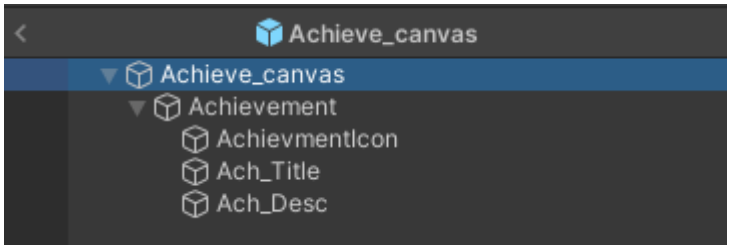


Figure 1



Figure 2

2.6.4 Scriptable Objects

In Unity, a Scriptable Object is a data container that enables developers to store numerous shared data separately, unlike MonoBehaviour scripts, which need to be attached to GameObjects, ScriptableObjects are not attached to them and can be utilized to create reusable assets via the Unity Editor. This asset can store data like configurations, settings, character stats, or game logic. This data can be used throughout a game without duplication, which is beneficial for managing game data, enhancing code flexibility and ease of modification, and reducing unnecessary memory usage. ScriptableObjects aid in the creation of assets for storing information that can be reused multiple times in various game components.

2.6.5 Programming Patters and Principles

Programming patterns and principles are foundational concepts that guide software developers in creating efficient, maintainable, and scalable code. By adhering to these patterns and principles, developers can avoid common pitfalls and enhance the quality of their software.

2.6.5.1 Design Patterns

Design patterns are typical solutions to common problems in software design. They represent best practices used by experienced developers and are categorized into three groups:

2.6.5.2 Creational Patterns

- Singleton: Ensures a class has only one instance and provides a global point of access to it.
- Factory Method: Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.

1.1.1.1.1 Structural Patterns

- Adapter (or Wrapper): Allows objects with incompatible interfaces to collaborate.
- Composite: Composes objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly.

1.1.1.1.2 Behavioral Patterns

- Observer: Defines a dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically.
- Strategy: Enables selecting an algorithm's runtime, encapsulating each one, and making them interchangeable.

2.6.5.3 SOLID Principles

SOLID is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable:

- Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one job or responsibility.
- Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
- Interface Segregation Principle (ISP): No client should be forced to depend on methods it does not use. ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

2.6.5.4 Other Important Principles and Patterns

- DRY (Don't Repeat Yourself): Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. This principle is aimed at reducing repetition of software patterns.

- KISS (Keep It Simple, Stupid): Systems work best if they are kept simple rather than made complex; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.
- YAGNI (You Ain't Gonna Need It): A developer should not add functionality until deemed necessary. YAGNI is part of the extreme programming (XP) practice and is aimed at reducing adding unnecessary features to a system.
- Composition Over Inheritance: Favoring object composition over class inheritance helps in achieving more flexible software architectures, making it easier to change the behavior of objects on the fly and avoid problems of deep inheritance hierarchies.

By understanding and applying these programming patterns and principles, developers can significantly improve the quality, maintainability, and scalability of their code.

2.6.6 Unity Services

Unity Services offer a suite of tools and features designed to support game developers through various stages of game development and management. These services include Unity Ads for monetization, Unity Analytics for understanding player behavior, Unity Cloud Build for streamlining build processes, and Unity Multiplayer Services for creating and managing online multiplayer experiences. Unity's Multiplayer Services, such as Netcode for GameObjects and the Lobby Manager, are particularly relevant for developing multiplayer games like "Avalon's Conviction." They provide the necessary infrastructure to handle multiplayer dynamics, from matchmaking and communication to managing game sessions. This *comprehensive ecosystem simplifies the technical complexities involved in game development,* allowing developers to focus more on creative aspects and gameplay mechanics, ensuring a seamless and engaging player experience.

2.6.6.1 Netcode for GameObjects

Netcode for GameObjects is a Unity package designed to facilitate the development of multiplayer games by providing a reliable and scalable network framework. It abstracts the complexity of network programming, allowing developers to focus on game mechanics rather than the intricacies of network management. With Netcode for GameObjects, developers can synchronize game objects across multiple clients in real-time, ensuring that player actions and game states are consistently updated across all connected sessions. This is crucial for multiplayer games where real-time interactions and decisions play a significant role in the gameplay experience. The package supports authoritative server architectures, reducing the potential for cheating and improving overall game fairness. Netcode for GameObjects also includes features for networked object instantiation, RPC (Remote Procedure Calls), and custom messaging, making it a versatile tool for building rich and interactive multiplayer game environments.

2.6.6.2 Lobbies

Unity's Lobby service is a crucial component for multiplayer games, offering a structured system for players to organize, join, and wait in pre-game lobbies before starting a session. This service simplifies the process of matchmaking, allowing players to either join existing lobbies based on specific game criteria or create new ones where they can invite friends or wait for others to join. It's particularly effective in games requiring team formation or in genres where player coordination before game starts is essential. For a game like "Avalon's Conviction," Unity's Lobby service would enable players to gather, discuss strategies, or simply socialize before diving into the game's strategic and deception-based gameplay. The lobby

system also supports custom lobby properties, which developers can use to filter and sort lobbies according to game modes, player skill levels, or other criteria, enhancing the overall player experience by ensuring more balanced and engaging game sessions.

2.6.6.3 Relay

Unity's Relay service is designed to enhance the multiplayer experience by facilitating seamless and secure connections between players across the globe. It operates by routing game traffic through Unity's servers, ensuring that players can connect with each other regardless of their network environments, including those behind strict NATs (Network Address Translators). This service is particularly beneficial for developers who want to ensure a smooth and accessible multiplayer experience without the need for complex network configuration. For a game which relies on real-time player interactions and connectivity, Relay provides a vital backbone, ensuring that players can easily join games and interact with one another with minimal latency and connection issues. Moreover, Unity's Relay is integrated with Unity's Matchmaker service, allowing for efficient pairing of players based on skill levels, preferences, or other custom criteria, further enhancing the multiplayer gaming experience.

2.6.6.4 Unity DevOps

Unity DevOps, formerly known as Unity Cloud Build, represents a suite of tools aimed at streamlining the development pipeline for Unity projects. It provides continuous integration and delivery services, enabling teams to automate the build process, test applications, and manage deployments efficiently. By leveraging Unity DevOps, developers can ensure that their builds are consistently tested across multiple platforms, reducing errors and improving overall game quality. This is particularly valuable for complex multiplayer games where consistent performance across various devices is crucial for maintaining a fair and competitive environment. Unity DevOps also facilitates collaboration among development teams by allowing for easy sharing of builds and rapid iteration on feedback, ensuring that the game development process is as smooth and efficient as possible. This comprehensive approach to game development and deployment helps teams focus on creativity and gameplay, knowing that the technical aspects of build and release management are handled effectively.

2.6.6.5 Plastic SCM

Plastic SCM is a version control system developed by Unity Technologies, designed to handle the complexities of game development workflows, particularly those involving large binary files and assets common in game projects. It offers robust tools for branching, merging, and versioning, enabling teams to work concurrently on different features, assets, or game areas without conflict. For developers working on multiplayer games Plastic SCM proves invaluable in managing the iterative process of game design, allowing for seamless integration of new content, mechanics, and fixes. Its GUI and distributed version control capabilities make it accessible for both technical and non-technical team members, ensuring that artists, designers, and programmers can effectively collaborate and track changes throughout the game development lifecycle. By leveraging Plastic SCM, teams can enhance productivity, reduce integration issues, and ensure that all aspects of the game development process are aligned and versioned correctly, facilitating a smoother path from development to deployment.

2.6.6.6 Player Authentication

Player Authentication in Unity is a critical mechanism designed to verify and manage the identities of players interacting with a game. This process is fundamental for ensuring a secure, personalized gaming experience, particularly in games that feature online components such as multiplayer functionalities, leaderboards, or cloud-based save systems. By implementing player authentication, developers can safeguard user data, streamline the player experience across various platforms, and prevent unauthorized access. Unity's robust suite of services supports a variety of authentication methods, including traditional email and password logins, social media integrations, and the use of Unity Accounts. This flexibility allows developers to choose the authentication strategy that best fits their game's needs, ensuring both security and ease of access for players around the globe.

In our game developed with Unity, player authentication plays a pivotal role in ensuring a secure and tailored gaming experience for each user. Recognizing the importance of flexibility and ease of access for our players, we have implemented two key methods of authentication. Firstly, we offer an anonymous login option, catering to players who wish to dive straight into the game without the upfront commitment of creating an account. This method is ideal for those prioritizing quick access while still maintaining a level of personalized interaction with the game. Secondly, for players integrated into the Google ecosystem, we support authentication via Google Play Games Services. This method not only streamlines the login process for users already using Google services but also enhances security and provides a seamless cross-platform gaming experience. By incorporating these authentication strategies, we aim to offer our players a blend of convenience, security, and personalization, right from their first interaction with our game.

2.6.6.7 Cloud Save

Unity Services' Cloud Save feature is a powerful tool designed to enhance the gaming experience by securely storing players' game progress and data in the cloud. This service allows game developers to offer a seamless gaming experience across different platforms, enabling players to pick up their game from exactly where they left off, regardless of the device they're using. Whether it's saving high scores, unlocked levels, character progression, or custom settings, Cloud Save ensures that player data is safely stored and easily retrievable. By

integrating Cloud Save into their games, developers can provide a more engaging and user-friendly experience, encouraging longer play sessions and greater player retention. Moreover, the Cloud Save feature is built with flexibility in mind, supporting a range of game types and sizes, from casual mobile games to more complex console or PC gaming experiences, making it an indispensable tool in modern game development with Unity.

2.6.6.8 Player Management

Unity Services' Player Management feature is designed to streamline the process of managing and understanding player interactions within your game. This comprehensive tool offers developers a robust framework for tracking player activities, behaviors, and engagement patterns, providing invaluable insights that can be used to enhance the gaming experience. Through Player Management, developers gain access to detailed analytics and data segmentation capabilities, enabling them to categorize players based on various criteria such as gameplay behavior, spending habits, or progression milestones. This not only aids in personalizing the player experience but also helps in identifying key areas for improvement, optimizing game features, and tailoring marketing strategies to increase player retention and satisfaction. Moreover, Player Management supports a seamless integration with other Unity Services like Cloud Save and Authentication, ensuring a cohesive and secure environment for both developers and players. By leveraging Player Management, developers can more effectively engage their audience, foster a loyal player base, and drive the success of their game in the competitive gaming market.

3 Implementation Methodologre

3.1 Version Control

Version control is a crucial technology that developers must thoughtfully implement and utilize. There are numerous tools available that proficiently manage the files and scripts of a project. Given Unity's existing partnership with Plastic SCM, we selected it primarily for its integration with the Unity game engine.

Version control systems generally offer a range of core features that enhance project management and collaboration. One such feature is branching, which allows developers to create separate branches which are derived from the main project, enabling them to work on different features or fixes without affecting the main codebase. This is particularly useful for testing new features without disrupting the functioning of existing code. Merging is another critical feature, which facilitates the integration of changes from different branches back into the main project, ensuring that all updates are unified in a controlled and efficient manner.

Additionally, version history is a fundamental aspect of version control, providing a detailed log of changes, including who made each change and when. This is typically managed through git commits, which are snapshots of the project at specific points in time. Each commit includes a message describing the change, making it easier to track progress, audit changes, and revert to previous versions if necessary. With these tools, teams can manage their development processes more effectively, reduce errors, and improve overall productivity.

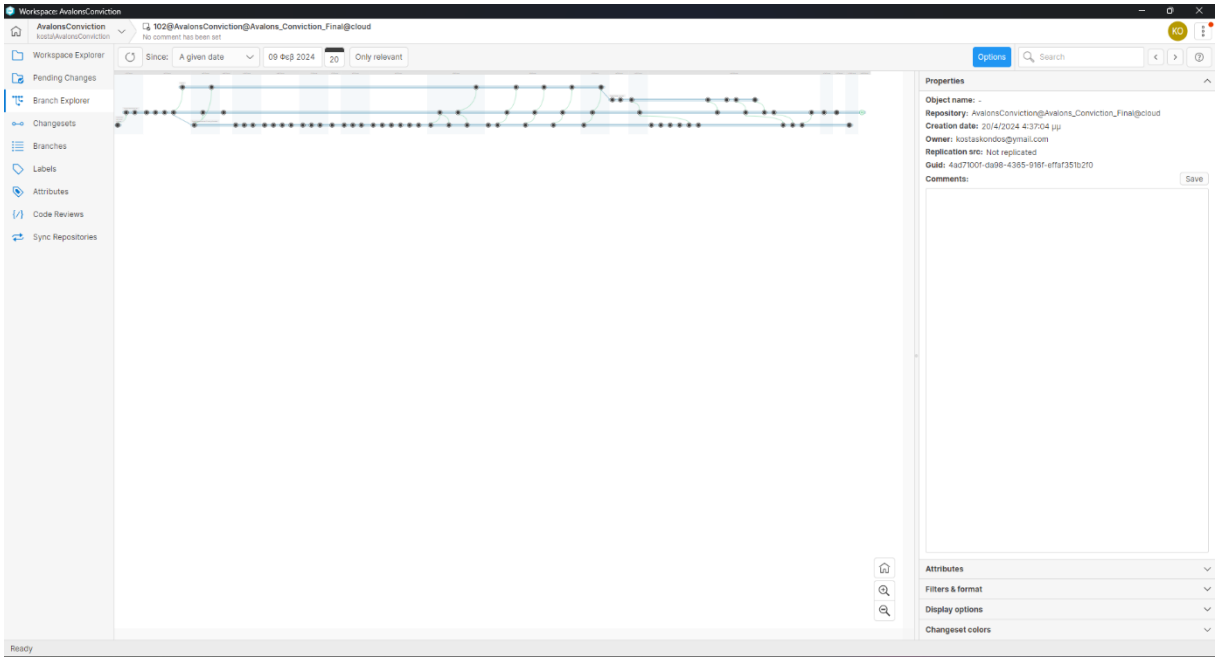


Figure 3

3.2 Workflow Diagram

To start developing a video game we must have a clear image of how the UX will be on the hands of the player, a diagram which can explain each important element of the game can help with the flow of development as well keeping a clear goal.

Most selections within the main menu are designed to present overlay panels, rather than transitioning to separate scenes, which streamlines navigation and allows for quick, uninterrupted access to game features. This approach underscores a user-centric design philosophy, facilitating an intuitive and hassle-free interface that enhances player engagement and underpins a focused development trajectory.

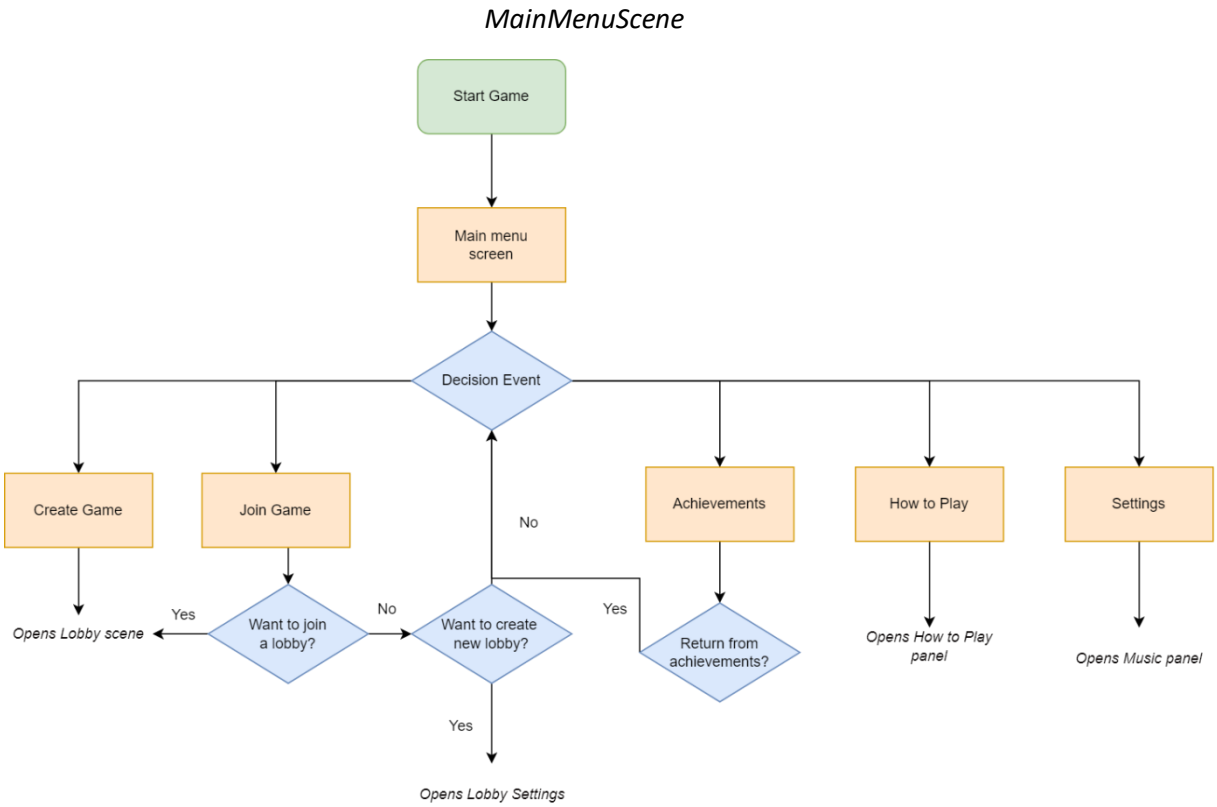


Figure 4

Lobby Scene

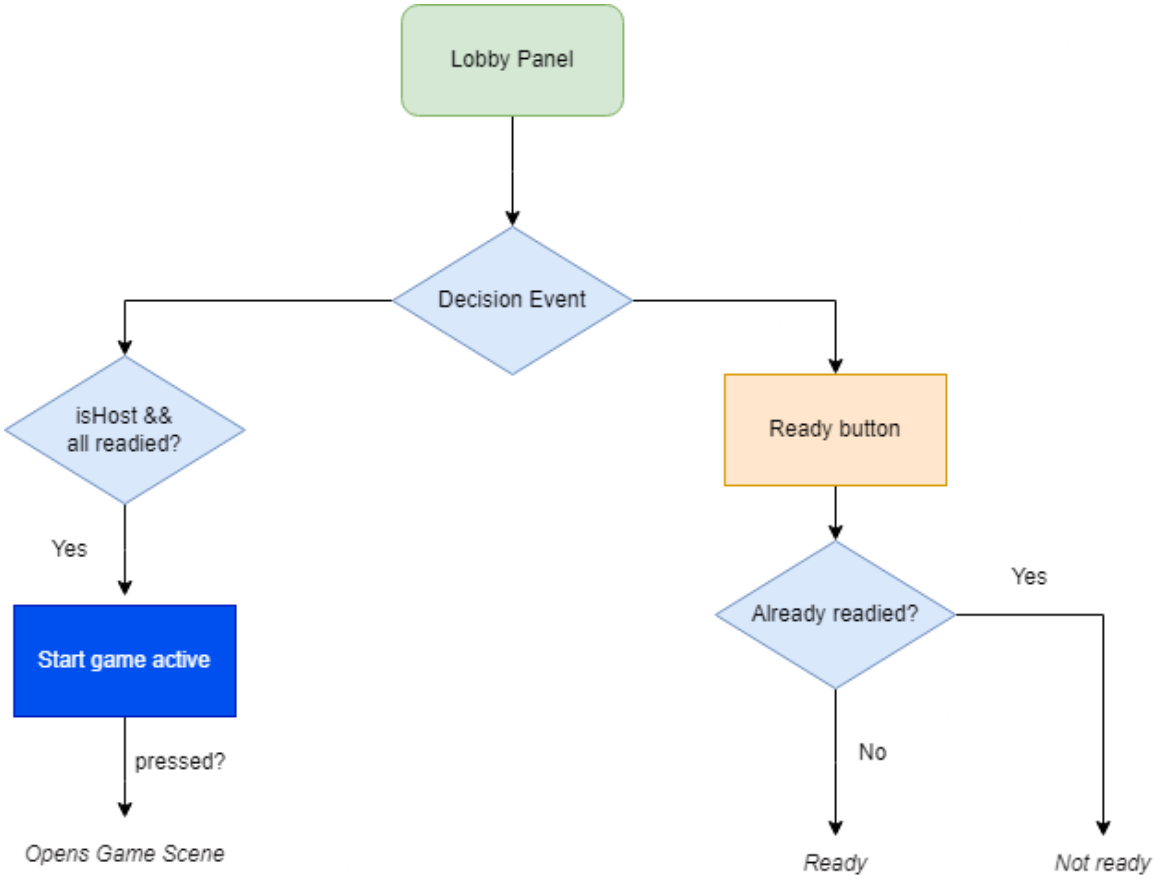


Figure 4

Game Scene

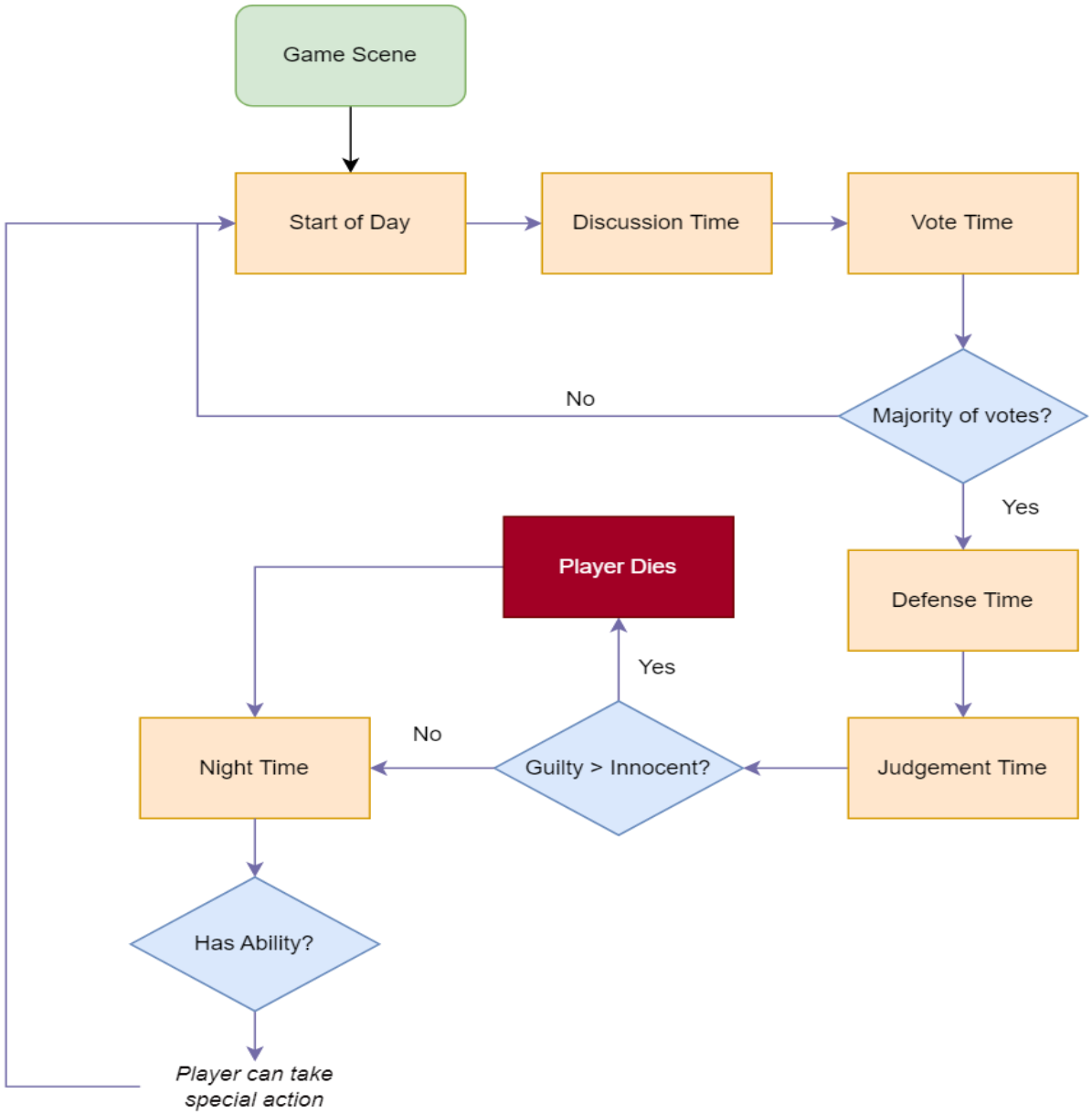


Figure 6

4 Game Design

In designing the game's UI, we leveraged the Unity Editor along with Gimp, utilizing their vast array of features such as buttons, images, canvases, and texts. This combination allowed us to craft all scenes and menus with precision and organization. Aesthetics played a crucial role in enhancing the game's appeal, with background and key visual elements sourced from the public domain to enrich the game's visual experience. The design process followed a structured sequence to ensure a cohesive and engaging user interface.

4.1 Login Scene

In our game, the first touchpoint for players upon launch is the login scene, a pivotal interface designed to facilitate a smooth and secure entry into the gaming world. Within this scene, players are presented with two distinct choices for logging in: they can either opt for Unity's Anonymous Services for a swift and hassle-free entrance without the need for personal information, or they can choose to log in via Google Play Games Services for a more integrated experience, especially beneficial for those already embedded within the Google ecosystem. This flexibility in login methods caters to a wide range of player preferences, ensuring ease of access while maintaining security and personal data protection. Furthermore, the login scene is equipped with options for players to logout, switch between login services, or proceed directly to the main menu scene, offering a seamless navigation experience. This approach not only respects the player's autonomy in choosing their preferred method of engagement but also sets the tone for a user-friendly and personalized gaming journey right from the start.

4.1.1 Error Panel

This panel plays a vital role in enhancing user experience by promptly notifying players of various errors, such as an invalid lobby name or network-related issues (e.g., incorrect lobby code, full lobby). It simplifies troubleshooting by displaying a clear error message and includes a back button for easy panel closure.

4.1.2 Loading Panel

During transitions and data processing moments, players are presented with a thoughtfully designed loading panel. This panel prominently displays the word "Loading" at its center, serving as a clear indicator of ongoing processes behind the scenes. Complementing this message is an animated hourglass, meticulously crafted to rotate smoothly, adding a visually engaging and aesthetically pleasing element to the waiting time. The rotating hourglass not only serves as a timeless symbol of the passage of time but also enhances the user experience by providing a dynamic visual cue that progress is being made. This careful integration of animation within the loading panel reflects our commitment to maintaining a seamless and visually appealing gaming experience, even during necessary loading intervals.

4.2 Main Menu Scene

The main menu interface is not merely a transitional screen but a pivotal element of the user experience. Upon launch, the player will be greeted with a visually engaging background, specifically designed to showcase the game's thematic essence.

This interface includes six strategically placed buttons, each with a distinct function. The 'Host button' and 'Join Game' buttons are fundamental to multiplayer dynamics, enabling players to either create or enter a game lobby. Notably, if a player attempts to proceed without a valid lobby name, a pop-up window emerges, guiding them to rectify the input. This feedback mechanism is integral to maintaining a fluid user experience and preventing potential frustrations.

The 'Achievements' button is a testament to the game's engagement strategy. Upon interaction, it reveals a comprehensive list of achievements, both unlocked and pending, serving as a motivational tool for players to continue their journey in the game.The remaining three buttons, aesthetically designed as wax stamps, contribute the thematic consistency of the interface.

The 'How to Player' button is crucial for orienting new players, offering an accessible overview of the game rules.

The 'Options' button allows players to personalize their experience, notably through volume control, demonstrating the game's commitment to accommodating individual preferences.

Finally, the 'Close' button, while functional in nature, is essential for providing players with a straightforward means to exit the game.

Overall, the main menu was designed to balance aesthetic appeal with functional efficiency. It exemplifies the game's commitment to providing an intuitive, player-centered experience, ensuring a smooth introduction to the game's mechanics and atmosphere.

4.3 Lobby Scene.

In the multiplayer game's lobby scene, three primary panels facilitate player interaction and game setup: the Host Game Panel, Lobby List Panel, and In Lobby Panel. This structured presentation enhances understanding by detailing the function and interaction mechanisms within each panel, emphasizing the user interface's role in facilitating game setup and player readiness before starting a game session.

4.3.1 Host Game Panel.

The Host Game Panel serves to set up the game lobby. Here, the host names the lobby and decides its privacy settings, choosing between a public lobby open to all or a private one accessible via a lobby code. Additional settings include timers for various game phases and a player count, each adjustable through increment and decrement buttons. A reset option and two stylized buttons for navigation—a wax-sealed back button and another to create the lobby—are included. Transitioning from this panel to the In Lobby Panel occurs upon lobby creation, making the lobby visible in the Lobby List Panel.

4.3.2 Lobby List Panel

The Lobby List Panel displays available lobbies, distinguishing private ones with an icon. It features a refresh button to update the lobby list, a scroll view showing each lobby's details, including player counts, and buttons to initiate hosting a game or to return to the main menu.

4.3.3 In Lobby Panel

The In Lobby Panel reveals details of the selected lobby, including its name, player count, and lobby code—with an option to copy the code to the clipboard. It lists connected players, their readiness status, and provides the host with the authority to remove players. A ready button toggles the player's readiness state, visually indicated by a splash of blood when ready. The host has access to a start game button—visible only when all players are, ready and the player count matches the host's setting—which transitions players to the game scene.

4.3.4 Join Lobby with Code Panel

Designed for seamless entry into private lobbies, this panel features an input field for the lobby code, a join button to facilitate access upon entering a valid code, and a back button for panel navigation. If an error occurs, such as an incorrect code, the previously mentioned Error Panel will appear, ensuring a user-friendly interface for lobby access.

4.4 Game Scene

The final scene of the game, pivotal for seamless gameplay, integrates six critical panels: the Fortune Wheel, the In-Game Panel, the Announcement Panel, the Judgement Panel, the Character Details Panel, and the Settings Panel. This arrangement ensures players have access to essential information and functionalities throughout the game.

4.4.1 Fortune Wheel Panel

The interaction begins with the Fortune Wheel, a captivating feature that randomly assigns roles to players. As the wheel spins and gradually halts, the designated role is highlighted beneath an arrow, accompanied by a textual indication of the role at the screen's bottom. This moment of anticipation transitions smoothly as the wheel fades out, bringing the In-Game Panel into focus. Each panel within the final scene is meticulously designed to facilitate a coherent, immersive gameplay experience, seamlessly integrating game mechanics, player interactions, and narrative progression.

4.4.2 In Game Panel

Central to the gameplay experience, the In-Game Panel displays the current game cycle's name and remaining time at the top, followed by a scrollable interface listing player-names, their respective vote counts, and a voting button. This panel adapts dynamically, showing role-specific UI elements for interaction based on the game's current phase and the player's role.

Below the scroll view, two prominent wax buttons provide additional functionalities. The left button reveals the Character Details Panel, offering insights into the player's assigned role, objectives, and a brief role description. The right button opens the Settings Panel, allowing players to adjust sound effects and music volume, enhancing the auditory experience of the game.

4.4.3 Judgement Panel

During voting phases, the Judgement Panel becomes the focal point, presenting the name of the player under scrutiny alongside two decisive options: 'Innocent' and 'Guilty.' A guilty verdict removes the player's ability to see or interact with others' voting buttons, visually indicating their elimination with a line through their name. This panel disappears once the judgement phase concludes, ensuring a fluid transition between game stages.

4.4.4 Announcement Panel

The Announcement Panel serves as the game's narrative vehicle, conveying the outcomes of critical events, such as the fate of players targeted by assassins at night. These announcements enrich the game's story, providing players with crucial updates and maintaining engagement.

5 Game Development – Code Implementation

5.1 Core Utility Scripts

5.1.1 Loader

The loader script is designed to manage scene transitions within a multiplayer game environment, handling the loading of different scenes based on player actions or game states. It includes functionality to switch between the main menu, game, and lobby scenes, facilitating the game's flow from one section to another. Additionally, it provides support for networked game sessions by using a network manager to load scenes in a single mode, ensuring that players are synchronized when transitioning between game phases. The script also allows for setting the state of the lobby panel, enabling dynamic UI changes that reflect the current game context or player decisions. Through this, it plays a critical role in navigating the game's structure, enhancing player experience by seamlessly integrating various game components.

```
public static class Loader
{
    2 references
    private static Scene targetScene;
    3 references
    public static LobbyPanelState NextPanelState { get; set; }
    8 references | 1 reference | 2 references | 3 references
    public enum LobbyPanelState { None, HostGamePanel, LobbyListPanel }
    9 references | 2 references | 1 reference | 3 references
    public enum Scene { MainMenuScene, GameScene, LobbyScene }
    4 references
    public static void Load(Scene targetScene, LobbyPanelState nextPanelState = LobbyPanelState.None)
    {
        NextPanelState = nextPanelState;

        Loader.targetScene = targetScene;

        SceneManager.LoadScene(targetScene.ToString());
    }
    1 reference
    public static void LoadNetwork(Scene targetScene) =>
        NetworkManager.Singleton.SceneManager.LoadScene(targetScene.ToString(), LoadSceneMode.Single);
    0 references
    public static void LoaderCallback() => SceneManager.LoadScene(targetScene.ToString());
}
```

Figure 7

5.1.2 Read Only Attribute and Drawer

The ReadOnlyDrawer script is responsible for creating a custom property drawer in the Unity Editor that makes serialized fields read-only while still allowing them to be visible. This functionality is achieved by disabling GUI editing for the specific property field, ensuring that its value cannot be changed in the editor. Using the #if UNITY_EDITOR directive ensures that this editor-specific code is only compiled and executed within the Unity Editor environment,

```
#if UNITY_EDITOR
using UnityEditor;
using UnityEngine;

[CustomPropertyDrawer(typeof(ReadOnlyAttribute))]
0 references
public class ReadOnlyDrawer : PropertyDrawer
{
    0 references
    public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
    {
        GUI.enabled = false; // Disable editing
        EditorGUI.PropertyField(position, property, label, true);
        GUI.enabled = true; // Re-enable editing
    }
}
#endif
```

Figure 8

not affecting the game's runtime logic or performance in the final build.

Finally, the ReadOnlyAttribute is used for marking certain fields with this custom attribute, indicating that these fields should be displayed as read-only in the Unity Editor. This approach allows developers to expose specific variables in the editor for viewing without permitting modifications, maintaining the integrity of predefined values while supporting debugging and informational purposes.

5.2 Login

```
using UnityEngine;

2 references
public class ReadOnlyAttribute : PropertyAttribute {}
```

Figure 9

5.2.1 Authentication Manager

Upon the game's launch, the script initializes Unity Services and sets up listeners for UI buttons that facilitate different authentication actions: signing in as a guest, signing in with Google Play Games, continuing after successful authentication, signing out, and exiting the game. The authentication method used by a player is stored using PlayerPrefs, ensuring that upon returning to the game, users are greeted with a personalized welcome message based on their previous session, enhancing the user experience by providing continuity.

The script also includes functionality to handle the transition between signing in and accessing the game's main content. If the player signs in successfully, the script displays a "continue" button, which leads to the loading of game data, potentially from cloud storage,

before proceeding to the game's main content. This process is visually supported by a loading panel, although the specific implementation of this panel is handled elsewhere.

For Google Play Games authentication, the script activates the Play Games platform and handles the authentication process, including error handling and storing the authentication token. If switching from Unity Services authentication to Google Play Games, it links the existing Unity Services account with the Google Play Games account, ensuring that the player's data remains consistent across different login methods.

5.2.2 Cloud Save Manager

This manager allows for saving data under specific keys, loading all saved cloud data into a local dictionary for quick access, and provides methods for retrieving data in various forms (e.g., string, int, float) based on the key. It ensures that any saved data is uploaded to the cloud when the application quits, making it a crucial component for games that rely on cloud storage for saving player progress, settings, and other data.

By leveraging Unity's CloudSave service, this script streamlines the process of handling player data, providing a smoother experience for both developers and players. It encapsulates the complexity of directly interacting with cloud services, offering straightforward methods for saving and loading data, which can be used across different game sessions and even across different devices, provided the player uses the same authentication method to access their game account.

5.2.3 Loading Manager

The LoadingManager class is designed to work within the lifecycle of the scenes it's part of, respecting the broader game's state management that handles object persistence across scenes. The class exposes public methods to control the display of loading screens, indicating various game states like joining a lobby, hosting a game or loading the game. Each method, such as StartJoiningLoading, StartHostingLoading, StartLoadingGame, and StartLoadingPlayers, is responsible for displaying the loading panel with a context-specific message. This is achieved by instantiating a LoadingPanelPrefab within the game's canvas and updating the displayed message to inform the player of the current action.
The 'Awake' method, typically used in Unity to initialize components, is simplified here to only contain necessary initial setup relevant to this class's scope without handling instance persistence. This reflects a shift to a more localized management approach, where the lifecycle of the loading screen is tightly coupled with the scene it belongs to.
The 'StopLoading' method remains straightforward, tasked with removing the loading panel from the display, which is a necessary action after completing the loading process or when it's no longer needed. Event handlers like OnJoinLobby and OnFailedToJoinGame are connected to gameplay events to hide the loading screen at the appropriate times, ensuring a seamless experience for the player.

5.2.3.1 Loading UI

The LoadingUI class is part of a system that manages the display of loading messages on the user interface. It employs a singleton pattern, signified by the static Instance property, to ensure there is only one active instance throughout the game, providing a centralized point of access for updating the loading message.

In the Awake method, the class assigns its own instance to the Instance property. This ensures that as soon as the object this script is attached to is initialized, Instance will refer to this specific LoadingUI object, allowing other scripts in the game to easily access and modify the loading message.

The core functionality of the LoadingUI class is encapsulated in the SetLoadingMessage method. This method accepts a string parameter message, which it uses to update the text of the loading message displayed on the screen. This mechanism facilitates real-time updating of the loading message to inform players of the current loading status, enhancing the interactive experience during game operations like loading scenes or processing data.

5.2.4 Error Manager

The ErrorManager class manages the error display within the game. It follows the singleton pattern to ensure there is only one instance of this manager throughout the game's runtime, accessible via the ErrorManager. This pattern facilitates easy access to the error management functionality from anywhere in the game. In the Awake method, the class sets its Instance property to reference the current instance of ErrorManager and calls DontDestroyOnLoad to ensure that the error manager persists across different scenes in the game. This persistence is important for a consistent error handling experience, as errors can occur in any part of the game.

The ShowErrorPanel method is responsible for displaying the error panel when an error occurs. It first calls LoadingManager.Instance.StopLoading to stop any ongoing loading processes, as displaying the error should take precedence over other screen activities. Then, it finds the first Canvas object in the scene to use as a parent for the error panel, ensuring the error message is visible on the UI layer.

The errorPanelPrefab is instantiated as a child of the canvas, making the error panel appear on the screen.Finally, the method uses SetErrorMessage(errorMessage) to set the actual error message on the error panel. This ensures that the specific error encountered by the player is displayed, allowing for better understanding and handling of the issue.

5.2.4.1 Error UI

The ErrorUI class handles the error messaging system, establishing a centralized mechanism to show and manage error messages through a singleton instance, accessible via ErrorUI.Instance. This setup ensures that error handling is streamlined and can be invoked from anywhere in the application.

Upon initialization in the Awake function, the class assigns itself to the Instance property, facilitating this universal access. Additionally, it sets up a listener for the backButton, so when the button is clicked, the error message display is terminated, enhancing user control over the UI elements.

The method SetErrorMessage is where the actual content of the error message gets updated. By passing a string to this method, the text property of the errorMessage component is set, effectively changing the message displayed to the user. This function exemplifies the class's primary role in updating and managing the content of error messages within the application's UI.

ErrorUI is structured to provide an efficient and user-friendly way to display and manage error messages, with straightforward mechanisms for setting the message and allowing the user to dismiss it. This design ensures that the error handling process is not only effective but also intuitive for the user.

5.3 Main Menu

5.3.1 Main Menu UI

The MainMenuUI is designed to manage the user interface for the main menu of a multiplayer game. It includes functionality to handle player input for names, alongside creating or joining a game session. The script employs several UI elements from the Unity Engine and TextMeshPro for improved text rendering. Specifically, it initializes listeners for create and join game buttons, validating player names against specific criteria—such as length and character content—before proceeding. Player names are sanitized to remove spaces and stored using Unity's PlayerPrefs to persist across sessions. If the player's name doesn't meet the validation criteria, an error message is displayed on an error panel, informing the player of the validation failure. This ensures that player names adhere to the game's requirements, improving the user experience and maintaining consistency in player identification.

5.3.2 Main Menu Cleanup

The MainMenuCleanUp is designed to ensure a clean state before the main menu of a game, likely "Avalon's Conviction," is loaded. It targets singleton instances of various manager classes such as NetworkManager, GameMultiplayer, LobbyManager, RoleManager, AchievementManager, and AudioManager. For each of these, if an instance exists, it destroys the GameObject to which the instance is attached, effectively resetting these systems. This approach is particularly relevant for multiplayer games, where lingering network connections, game state information, or audio settings from a previous session could interfere with the new game session. The script takes special care with AchievementManager and AudioManager by first checking for multiple instances and only destroying the GameObject if more than one instance is found, preserving a single instance of each. This ensures that essential services like achievements tracking, and audio management are maintained without duplication, supporting a clean and efficient game initialization process.

```
public class MainMenuCleanUp : MonoBehaviour
{
    0 references
    private void Awake()
    {
        if (NetworkManager.Singleton != null)
        {
            Destroy(NetworkManager.Singleton.gameObject);
        }
        if (GameMultiplayer.Instance != null)
        {
            Destroy(GameMultiplayer.Instance.gameObject);
        }
        if (LobbyManager.Instance != null)
        {
            Destroy(LobbyManager.Instance.gameObject);
        }
        if(RoleManager.Instance != null)
        {
            Destroy(RoleManager.Instance.gameObject);
        }
        if(AchievementManager.Instance != null)
        {
            var achievementManagers = FindObjectsOfType<AchievementManager>();
            if(achievementManagers.Count() > 1)
            {
                Destroy(AchievementManager.Instance.gameObject);
            }
        }
        if(AudioManager.Instance != null)
        {
            var audioManagers = FindObjectsOfType<AudioManager>();
            if(audioManagers.Count() > 1)
            {
                Destroy(AudioManager.Instance.gameObject);
            }
        }
    }
}
```

Figure 10

5.4 Lobby

5.4.1 Lobby Manager

The Lobby Manager script is a comprehensive solution for managing multiplayer lobby operations within Unity, integrating with Unity Services like Authentication, Lobbies, and Relay. It serves as the central hub for creating, joining, updating, and leaving lobbies, ensuring a smooth multiplayer experience. The script implements Singleton pattern to provide a globally accessible instance, facilitating easy access from other scripts.

The script handles a variety of key operations:

- Authentication: It initializes player authentication anonymously, allowing players to interact with online services.
- Lobby Operations: It provides functionality for creating lobbies with customizable settings, including lobby name, maximum players, and privacy settings. Players can join existing lobbies using a lobby code, and the host has the ability to kick players or start the game.
- Lobby List Management: Periodically refreshes the list of available lobbies and updates lobby information to reflect current states like player readiness or lobby availability.
- Networking and Relay Services: Integrates with Unity's Relay service to manage network traffic, ensuring players can connect to lobbies seamlessly and securely. The script handles the allocation of network resources and establishes connections for gameplay.
- UI Interaction: It interacts with various UI elements to display lobby information, error messages, and loading states, providing feedback to players during lobby operations.
- Event Handling: The script dispatches events for significant lobby actions, such as joining or leaving a lobby, updating lobby state, or kicking players, allowing other components to react accordingly.

Overall, the Lobby Manager script encapsulates the complexities of lobby management and network setup, abstracting them into a user-friendly interface that supports the dynamic nature of multiplayer game lobbies. (see Appendix 1)

5.4.2 Lobby List UI

The LobbyListUI is tailored for managing the lobby list UI in a multiplayer game environment. It is responsible for displaying available game lobbies to players, allowing them to join an existing game or refresh the list to see newly created lobbies. The script uses several serialized fields to link UI elements like loading panels, buttons, and a template for displaying individual lobbies within a container. It initializes by setting up button listeners for refreshing the lobby list and exiting back to a main menu or previous UI state.

Upon starting, the script subscribes to events from the LobbyManager to handle lobby-related actions such as joining, leaving, or being kicked from a lobby, as well as updating the lobby list when changes occur. The lobby list is dynamically updated through the UpdateLobbyList method, which processes new, existing, and removed lobbies, instantiating or destroying UI elements as needed to reflect the current state of available lobbies. This dynamic update mechanism ensures that the lobby list UI always presents the most current information to the player.Functionality to clear the lobby list UI upon joining a lobby and to show or hide the UI based on player interaction is also included, providing a seamless navigation experience. The script concludes with a method to exit lobbies, cleanly handling the process of leaving a lobby, shutting down the network manager, and transitioning back to the

main menu scene, ensuring a smooth and intuitive user flow within the game's multiplayer ecosystem. (see Appendix 2)

5.4.3 Lobby List Single UI

The LobbyListSingleUI script is designed to manage the display and interaction logic for a single lobby entry within a multiplayer game's lobby list UI. It dynamically updates the UI elements to show the lobby's name, the current number of players versus the maximum allowed, and an indicator for whether the lobby is private (password-protected) or public. The script associates each lobby entry with a button, enabling players to attempt joining the selected lobby upon clicking.

When a player attempts to join a lobby, the script checks if the lobby requires a password. If so, it triggers the display of a UI component (LobbyJoinCodeUI) that prompts the player to enter the password. For public lobbies without a password, the script directly initiates the join process, displaying a loading UI (LobbyLoadingUI) while the join operation is in progress, leveraging asynchronous programming to await the completion of the lobby joining process.

This approach ensures a user-friendly experience by providing immediate visual feedback and intuitive interaction mechanisms. Players can easily understand the status of each lobby and take

```
public class LobbyListSingleUI : MonoBehaviour
{
    1 reference | 1 reference
    [SerializeField] private TextMeshProUGUI lobbyNameText, playersText;
    1 reference | 0 references
    [SerializeField] private GameObject privateLobbyImage, lobbyGameObject;
    0 references
    public const string KEY_RELAY_JOIN_CODE = "RelayJoinCode";
    5 references
    private Lobby lobby;
    0 references
    private void Awake()
    {
        GetComponent<Button>().onClick.AddListener(async () =>
        {
            if(lobby.HasPassword)
            {
                LobbyJoinCodeUI.Instance.SetLobby(lobby);
                LobbyJoinCodeUI.Instance.Show();
            }
            else
            {
                LobbyLoadingUI.Instance.StartJoiningLoading();
                await LobbyManager.Instance.JoinLobby(lobby);
            }
        });
    }
    1 reference
    public void UpdateLobby(Lobby lobby)
    {
        this.lobby = lobby;

        privateLobbyImage.SetActive(lobby.HasPassword);

        lobbyNameText.text = lobby.Name;

        playersText.text = lobby.Players.Count + "/" + lobby.MaxPlayers;
    }

    1 reference
    public Lobby GetLobby()
    {
        return lobby;
```

Figure 11

appropriate actions to join, enhancing the overall usability and accessibility of the game's multiplayer lobby system.

5.4.4 Lobby Join Code UI

When the player clicks to join a locked lobby, the join lobby code panel appears, providing a user interface to enter a unique code for accessing specific lobbies. This script, attached to the lobby join code UI, facilitates the process by linking the join button to the JoinLobby method. Upon clicking the join button, the script initiates an attempt to join the specified lobby using the code entered in the input field. It leverages asynchronous programming to wait for the lobby join process to complete, showing a loading indicator in the meantime. If the join attempt is successful, it reveals the lobby's game object, signifying the player has successfully entered the lobby. Conversely, if an error occurs, such as entering an incorrect code or experiencing a network issue, the script halts the loading indicator and displays an error message. The script also includes methods to show or hide the lobby code panel and to set the current lobby, ensuring the UI responds appropriately to player actions and the game's state.

```
public class LobbyJoinCodeUI : MonoBehaviour
{
    4 references
    public static LobbyJoinCodeUI Instance { get; private set; }
    2 references
    [SerializeField] private Lobby lobby;
    1 reference
    [SerializeField] private Button joinButton;
    1 reference
    [SerializeField] private TMP_InputField codeInputField;
    2 references | 0 references
    [SerializeField] private GameObject lobbyCodePanel, lobbyGameObject;

    0 references
    private void Awake()
    {
        Instance = this;

        joinButton.onClick.AddListener(JoinLobby);
    }
    1 reference
    public async void JoinLobby()
    {
        try
        {
            await LobbyManager.Instance.JoinLobbyByCode(lobby, codeInputField.text);
        }
        catch (LobbyServiceException e)
        {
            LoadingManager.Instance.StopLoading();
            ErrorManager.Instance.ShowErrorPanel(e.Message);
        }
    }
    1 reference
    public void SetLobby(Lobby lobby) => this.lobby = lobby;
    1 reference
    public void Show() => lobbyCodePanel.SetActive(true);
    1 reference
    public void Hide() => lobbyCodePanel.SetActive(false);
}
```

Figure 12

5.4.5 Lobby Create UI

The Lobby Create UI script manages the user interface for creating a new lobby. It provides players with the ability to configure their lobby's settings before hosting it. The script interacts with various UI elements, such as input fields for the lobby name, buttons to set the lobby as public or private, and controls to adjust the maximum number of players allowed in the lobby.

Upon initialization, the script sets up listeners for its UI components. The create lobby button, when clicked, triggers a validation of the lobby name entered by the player. If the name is deemed valid—meeting criteria such as length and character restrictions—the script proceeds to save the lobby name and player count settings using PlayerPrefs. It then initiates the lobby creation process with these settings, displaying a loading panel while awaiting completion. If the validation fails, an error message is displayed to the user.

The public and private buttons toggle the lobby's accessibility, affecting whether a lobby code is needed to join. The script also allows for the dynamic adjustment of the maximum

number of players through plus and minus buttons, with built-in limits to prevent setting a player count outside of the allowed range. This count is displayed to the user and updated in real-time as adjustments are made.

Additionally, the script includes functionality to hide the lobby creation UI when a player successfully joins a lobby, ensuring a seamless transition from lobby setup to lobby participation. The lobby name validation checks for acceptable length and absence of special characters or numbers, ensuring that names are suitable for all players and systems. Overall, this script plays a crucial role in facilitating the creation of customizable, player-hosted lobbies, enhancing the multiplayer experience by allowing players to set up game sessions according to their preferences. (see Appendix 1)

5.4.6 Lobby Time Settings UI

The Lobby Time Settings UI script facilitates the customization and management of timer settings for various aspects within a multiplayer lobby environment. It defines a TimerSetting class to hold references to UI elements associated with each timer, including buttons for increasing and decreasing the time, a text display for the timer, and a preference key for saving the timer's value. The script initializes by adding listeners to the increase and decrease buttons for each timer setting, allowing for fine-tuned adjustments in fixed increments. Additionally, it incorporates continuous change functionality when a button is held down, accelerating the time adjustment process.

The script also features a reset button to revert all timers to their default values, an exit button to leave the current screen, and a start button to save the adjusted values into the game's settings. These functionalities are supported by coroutines that enable smooth and responsive UI interactions, enhancing the user experience by providing immediate feedback and a sense of control over the game's timing aspects. Upon starting, the script initializes timer settings based on saved preferences, offering a personalized setup for returning players. Overall, this script plays a crucial role in allowing players and hosts to tailor the pace and flow of the game to their preferences, contributing to a more engaging and customized gaming experience. (see Appendix 2)

5.4.7 Lobby UI

The Lobby UI is integral to managing the lobby user interface (UI) in a multiplayer game, offering a comprehensive suite of functionalities to enhance player interaction within the lobby. At its core, the script initializes a singleton pattern for itself to ensure a single instance controls the lobby UI across the game. It handles various UI elements like displaying the lobby's name, player count, and a unique lobby code, alongside player templates for dynamically showing each player's ready state and options.

The script activates the lobby UI, allowing players to start the game, leave the lobby, toggle their ready state, and copy the lobby code to the clipboard for sharing. It reacts to significant events like joining or leaving a lobby, updating lobby details, player ready states, and handling the removal of players from the lobby—either through kicking or voluntary departure. These actions trigger updates across the UI to reflect changes in real-time, such as adjusting the displayed player count, updating the ready state of players, and managing the visibility of the start game button based on the readiness of players.

Additionally, the script incorporates methods to clear the lobby UI when leaving and to dynamically create or update player slots as participants join, leave, or change their ready state within the lobby. This dynamic adjustment ensures the lobby accurately represents the current

state of the game, player readiness, and participation, fostering a seamless multiplayer experience. (see Appendix 3)

5.4.8 Lobby Single UI

The Lobby Single UI is designed to manage the individual UI elements for players in a multiplayer lobby, including displaying player names, their readiness status, and the option to kick players from the lobby. Upon initialization, it sets up a listener for the kick player button, allowing the host or authorized players to remove participants from the lobby. The script can dynamically update the UI to reflect each player's current state, showing their name and whether they are ready or not, with visual cues (green for ready, red for not ready) for quick understanding. It also provides functionality to indicate empty slots in the lobby by displaying a "Waiting for player" message and hiding the kick button and readiness status, maintaining a clean and informative lobby interface. This script is integral to enhancing user interaction within

```
public class LobbyPlayerSingleUI : MonoBehaviour
{
    3 references
    [SerializeField] private Button kickPlayerButton;
    2 references | 4 references
    [SerializeField] private TextMeshProUGUI playerNameText, playerReadyStateText;
    5 references
    private Player player;
    0 references
    private void Awake() => kickPlayerButton.onClick.AddListener(KickPlayer);
    2 references
    public void SetKickPlayerButtonVisible(bool visible) => kickPlayerButton.gameObject.SetActive(visible);
    4 references
    public void UpdatePlayer(Player player)
    {
        this.player = player;

        if(player != null)
        {
            playerNameText.text = player.Data[LobbyManager.KEY_PLAYER_NAME].Value;

            playerReadyStateText.text = player.Data[LobbyManager.KEY_PLAYER_READY_STATE].Value.Equals("True") ?

            "<color=green>Ready</color>" : "<color=red>Not Ready</color>";

            playerReadyStateText.gameObject.SetActive(true);
        }
    }
    3 references
    public void SetEmptySlot()
    {
        player = null;
        playerNameText.text = "Waiting for player";
        playerReadyStateText.text = "<color=red>Not Ready</color>";
        kickPlayerButton.gameObject.SetActive(false);
        playerReadyStateText.gameObject.SetActive(false);
    }
    1 reference
    private void KickPlayer()
    {
        if (player != null)
        {
            LobbyManager.Instance.KickPlayer(player.Id);
        }
    }
    7 references
```

Figure 13

the lobby, ensuring that players have the necessary information and controls for a smooth multiplayer experience.

5.5 Game

5.5.1 Player

The Player class is a comprehensive representation of a player entity, encompassing both gameplay mechanics and UI interactions. It synchronizes player states across the network, ensuring that each player's status is consistently updated in real-time for all participants in the game.

Central to the class are NetworkVariables, which automatically sync data like player ID, name, alive status, and various game-specific states (e.g., protection, defense, blocking, vote counts) across the network. These variables trigger callbacks such as OnIsAliveChanged, OnVoteCountChanged, and OnJudgementChoiceChanged when their values change. This system allows for immediate visual feedback on the game's current state to all players.

Moreover, the class implements methods for role management and player interactions, such as voting and judgement decisions, which are central to the game's logic. The ability to track and modify the player's role, along with their voting targets, enables complex game mechanics and player strategies to unfold within the game. Additionally, methods like GetAssassinsTarget showcase the class's capability to execute game-specific logic, determining the target of an assassin's vote based on the game's rules. This blend of networked state management, UI interaction, and gameplay logic.

5.5.2 Role

The Role abstract class, designed as a ScriptableObject, represents the distinct roles a player can assume in the game, each with unique abilities, objectives, and alignment. Roles are differentiated by characteristics such as special abilities, team alignment, and winning conditions, and they dictate the player's interactions and objectives within the game. By providing methods for different times of the game day, such as DefenseTime and NightTime, it allows for dynamic role-based behavior that integrates smoothly with the game's progression and phases. This system not only enhances the depth and variety of player interactions but also directly influences the game's complexity.

The class’s implementation includes lifecycle methods for each game phase, enabling roles to execute phase-specific logic that can affect the game state, player decisions, and the overall flow of the game. These mechanisms are essential for creating a dynamic and engaging multiplayer experience, where players' decisions and role-specific actions significantly impact the unfolding game narrative and outcomes.

5.5.3 Role Manager

The RoleManager class acts as a central hub for managing and distributing player roles in a multiplayer setting. It employs Unity's MonoBehaviour to integrate seamlessly with the Unity Engine's lifecycle and utilizes the singleton pattern to ensure there's a single, globally accessible instance of the manager throughout the game. This class is crucial for assigning roles to players, encompassing various roles including townspeople, special characters like assassins or gravekeepers, each with unique alignments and team affiliations.

Upon initialization, RoleManager loads all available roles from a specified directory within the project's resources, categorizing them based on their alignment (good, neutral, evil) and specific characteristics (e.g., whether they are part of the assassin's team). This pre-categorization facilitates efficient role assignment based on game dynamics and player count, ensuring a balanced distribution of roles that align with the game's design and the intended player experience.

The role assignment process dynamically calculates the number of players to be assigned to each category based on the total number of players participating in the game session. This process includes the addition of special roles and ensures that the game remains engaging and balanced, with each player receiving a role that contributes to the game's strategic depth. The RoleManager also includes functionality to shuffle the role pool, ensuring random distribution of roles for each game. This comprehensive management of player roles is pivotal for the game's progression, directly influencing gameplay mechanics, player interactions, and the overall narrative development.

5.5.4 Role Information

The RoleInformation class allows players to access detailed information about their assigned role during gameplay. Integrated within the game's roles system, this class utilizes Unity's UI framework, including Image for role visuals and TextMeshProUGUI for text display, to present players with an informative overview of their role, including the role's name, description, objectives, and an accompanying image.

Upon initialization in the Awake method, the class registers an event listener to a button, roleInformationButton, designed to reveal the role information panel when clicked. This design approach ensures that detailed role information is readily accessible to players at any point during the game, enhancing player engagement and understanding of their role's significance and expected contributions to the game's dynamics.

5.5.5 Game Network

The GameNetwork class is a comprehensive script managing the networking aspects of the game. It integrates with Unity's networking system, ensuring smooth communication and state synchronization across different clients connected in the game session. This class maintains a list of registered players and their states, such as whether they are ready or have loaded into the game. The use of dictionaries like playerReadyDictionary and playerLoadedPlayersDictionary allows for efficient tracking of each player's status.

Additionally, network variables like LoadedRole and LoadedPlayers are employed to synchronize game, ensuring all clients have consistent views of the game's progress. Event handling methods such as StartClient, StopClient, StartHost, and StopHost manage the lifecycle of the network session, handling the connection and disconnection processes, as well as the initialization of the host.

These methods use callbacks from the NetworkManager to respond to network events like client connections and disconnections, ensuring that the game's network state is always current.

The role assignment and player management logic are embodied in functions like AssignRolesToPlayers, RemovePlayerFromList, and SetPlayerReadyAssigned RoleServerRpc. These methods deal with the game's logic, such as assigning game roles to players and managing their ready status.

In summary, GameNetwork is a critical component that handles the networking infrastructure of the game, managing player connections, synchronizing states across the network, and controlling game session flow. Its design reflects a blend of network management and game logic functionalities, facilitating a cohesive multiplayer game experience.

5.5.6 Vote Manager

The VoteManager class manages and processes player votes, particularly in judgment scenarios like determining guilt or innocence of a player. Its design leans heavily on Unity's Netcode for GameObjects to facilitate robust and synchronized network interactions. It ensures that actions such as casting votes and resolving judgments are coordinated across the network, maintaining game state integrity across different client-server instances. The class uses server-side Remote Procedure Calls (RPCs) for critical operations, allowing for the execution of game logic on the server while reflecting changes to all clients, a necessary feature for real-time multiplayer games.

Key functions like CastVote, SendJudgementGuilty, and SendJudgementInnocent outline the class's capability to handle the complexity of vote dynamics within the game. These methods, alongside others for vote count adjustments, provide a mechanism for players to influence the game's outcome through democratic choice, directly impacting the narrative or player standings based on collective or individual decisions.

Moreover, the CalculateVoteResults and ResolveJudgement methods demonstrate the class's utility in concluding voting phases by analyzing the collected votes to enforce game consequences. The ability to reset votes ensures that each voting session starts afresh, preventing carry-over effects that could skew the fairness or outcome of subsequent rounds. Through the efficient use of network variables and RPCs, VoteManager exemplifies a tailored approach to integrating complex voting mechanics seamlessly into the multiplayer framework.

5.5.7 Player Data

The PlayerData script is designed as a struct to hold essential information for each player in a multiplayer game using Unity's Netcode for GameObjects. This data structure includes the player's ID and name, both of which are stored as FixedString64Bytes to ensure they are efficiently serialized across the network. Additionally, it tracks the player's client ID, a unique identifier used by the network to manage connections, and the player's role ID, which can be used to assign specific roles or characteristics within the game.

The script implements the IEquatable<PlayerData> interface, allowing for easy comparison between two PlayerData instances. This is particularly useful for determining if two players are the same based on their client ID, role ID, player name, and player ID.

Moreover, the PlayerData struct implements the INetworkSerializable interface, making it ready for serialization and deserialization over the network. The NetworkSerialize method is defined to serialize or deserialize the player's data using a BufferSerializer<T>, which is a generic parameter that can handle both reading and writing operations. This method explicitly serializes the clientId, roleId, playerName, and playerId fields, ensuring that all necessary player information is accurately transmitted across clients and the server.

Overall, the PlayerData script is a compact and efficient way to handle player information in a networked Unity game, facilitating easy data management and synchronization across the network.

```
using System;
using Unity.Collections;
using Unity.Netcode;

12 references
public struct PlayerData : IEquatable<PlayerData>, INetworkSerializable
{
    8 references
    public FixedString64Bytes playerId;
    5 references
    public FixedString64Bytes playerName;
    10 references
    public ulong clientId;
    5 references
    public int roleId;
    0 references
    public bool Equals(PlayerData other)
    {
        return
            clientId == other.clientId &&
            roleId == other.roleId &&
            playerName == other.playerName &&
            playerId == other.playerId;
    }

    0 references
    public void NetworkSerialize<T>(BufferSerializer<T> serializer) where T : IReaderWriter
    {
        serializer.SerializeValue(ref clientId);
        serializer.SerializeValue(ref roleId);
        serializer.SerializeValue(ref playerName);
        serializer.SerializeValue(ref playerId);
    }

}
```

Figure 14

5.5.8 Action Data

The ActionData struct is a concise, yet versatile data structure designed to encapsulate the details of an action. This struct not only stores essential action-related information—such as the player initiating the action , the action type , the target player of the action , and the nature of the action —but it also adheres to both IEquatable<ActionData> and INetworkSerializable interfaces, ensuring it can be easily compared for equality and serialized over the network.

Implementing IEquatable<ActionData> allows for comparison between two instances of ActionData, which is crucial for validating actions within the game's logic to check if two actions are identical or to ensure an action's uniqueness. The equality check is comprehensive, considering all aspects of the action data—identifying player, action type, targeted player, and the action's nature—thereby ensuring a high level of precision in comparing actions.

The inclusion of INetworkSerializable with a custom NetworkSerialize method enables ActionData to be serialized and deserialized across the network efficiently. This is particularly important in multiplayer games where actions initiated by a player need to be communicated to the server and then to other players. By serializing the player IDs, action types, target player IDs, and the action's nature, the game can maintain a synchronized state across all clients, ensuring that every player has a consistent view of the game world and its ongoing activities. This serialization process is important for the smooth and coherent operation of networked multiplayer games, allowing for the accurate and timely transmission of game state changes.

5.5.9 Message Data

The MessageData struct is specifically designed for handling chat messages in Unity's multiplayer environments, utilizing Netcode for GameObjects. It encapsulates the essentials: the player's ID, the message content, and a boolean flag indicating whether the message should be highlighted in red, used for announcements.

By implementing IEquatable<MessageData>, it allows for an efficient comparison mechanism, ensuring that announcements can be uniquely identified and compared based on their content, and stylistic attributes like color.

Incorporating INetworkSerializable with a custom NetworkSerialize method. This aspect is crucial, as it enables the struct to be serialized and deserialized when sending chat messages across the network, ensuring that all players receive and view messages as intended. The serialization process includes the player ID, the message itself stored as a FixedString512Bytes to optimize memory usage and network bandwidth, and the boolean flag for message color, providing a compact yet flexible format for chat data transmission. This design highlights the struct's utility, offering both functionality and efficiency in displaying information to the player.

5.5.10 Time Manager

The TimeManager script is designed to control the flow of time within a multiplayer game environment. This script is responsible for managing game phases and their durations, displaying the current phase and countdown timer to players, and handling the transition between different phases of the game.

Upon initialization, TimeManager set’s itself as a singleton instance to ensure there is only one manager controlling the game’s timing aspects. It uses serialized fields to link UI elements for displaying the current game phase and the countdown timer, and a button to start or stop the game, enhancing user interaction.

The script maintains a boolean flag isGameRunning to track the game's state and employs NetworkVariable<float> and NetworkVariable<FixedString32Bytes> to synchronize the remaining time and the name of the current phase across the network. This synchronization is crucial for ensuring that all players have a consistent view of the game's timing, regardless of their individual network conditions.

A list of phaseKeys and dictionaries for phaseDurations and phaseDisplayNames are used to define the various game phases, their durations, and display names. This setup allows for flexible configuration of each phase's length and how it is presented to the players.

The script features methods to start and stop the countdown, with server RPCs (ServerRpc) to allow these actions to be triggered from client-side code in a networked context. It includes logic to load phase durations from player preferences, providing a means to customize phase lengths easily.

The core functionality revolves around the countdown coroutine, which decrements the time remaining in the current phase and triggers the transition to the next phase upon completion. This transition is handled smoothly through client RPCs (ClientRpc), which notify clients about phase changes, ensuring the game's state is updated across all connected players.

Additionally, utility functions are provided to update the UI elements based on the timer's current state and to clean up event listeners upon the destruction of the TimeManager instance, preventing potential memory leaks or unwanted behavior when the game scene changes.

In essence, the TimeManager script plays a critical role in managing the temporal aspects of the game, facilitating a structured and synchronized progression through its various phases. (see Appendix 7)

5.5.11 Game Manager

The GameManager class serves as a vital component in the network part. Its role includes orchestrating game state management, player readiness, and the game's outcome based on winning conditions. Utilizing Unity's networking infrastructure, the class inherits NetworkBehaviour to enable network-related functionalities. It embodies a singleton pattern through the static Instance property, ensuring there is only a single global access point for game management.

The GameManager controls game states (WaitingToStart, Loaded, GameOver) via a NetworkVariable, allowing synchronized state changes across the network. It employs event-driven mechanisms to react to state changes and player actions, such as players becoming ready in the lobby or the UI loading players. Player readiness is tracked through a dictionary, playerReadyDictionary, mapping player identifiers to their readiness status, which is pivotal for transitioning the game state to Loaded when all players are ready.

Additionally, it contains a list of registered players (_registeredPlayers), facilitating player management and the evaluation of winning conditions through role-specific logic encapsulated within player objects. Winning conditions are checked through a series of server and client remote procedure calls (RPCs), enabling a distributed decision-making process to determine the outcome of the game.

This process culminates in the CheckEndGame method, which aggregates winning conditions across various player roles (e.g., Assassin, Jester, Town, Neutral) to declare the game's result, showcasing a flexible and extensible framework for defining and assessing game objectives in a multiplayer context. (see Appendix)

5.5.12 Time State Manager

The TimeStateManager class encapsulates the management and transition between different time states or phases within the game, such as NightTime, AnnouncementTime, and DiscussionTime, among others. This class is a part of the game's network layer, as indicated by its namespace Assets.Scripts.Game.Network, and extends MonoBehaviour, allowing it to leverage Unity's game object system for lifecycle management and coroutine support.

At its foundation, the TimeStateManager employs, like the other managers, a singleton pattern, ensuring a single global instance is accessible throughout the game for state management. This pattern facilitates a centralized approach to managing time-related events and transitions, crucial for synchronizing game logic across networked multiplayer sessions. The class introduces a method for handling the start of each time state, HandleTimeStart, which triggers specific behaviors in the local player's role depending on the current time state. Similarly, the HandleTimeEnd method manages the conclusion of each time state, executing role-specific cleanup or transition logic.

Moreover, the class features a coroutine, DisplayAnnouncements, designed to sequentially display game announcements to players, illustrating the use of Unity's coroutine system for timing and asynchronous UI updates. Through its structured approach to handling time states, including both the initiation and conclusion of these phases, alongside the management of game announcements, the TimeStateManager plays a pivotal role in the

orchestration of game dynamics and player interaction within a networked multiplayer environment. (see Appendix)

5.5.13 Picker Wheel

The PickerWheel script is a key component of the UI system designed to visually represent and select game roles through a spinning wheel animation. Structured within the EasyUI.PickerWheelUI namespace, it integrates with Unity's UI system, utilizing elements like GameObject, Transform, TMP_Text, and AudioSource to create an interactive and dynamic experience.

At the script's core, it initializes a list of roles from the RoleManager, ensuring each role is unique. The wheel is then dynamically generated to match the number of roles available, adjusting the size of each segment to fit the wheel properly. This setup phase includes generating visual lines to divide the wheel into segments, each corresponding to a different role, and assigning the roles' names and images to these segments.

The spinning functionality is triggered by calling the Spin method, which uses DOTween, a popular Unity animation library, for smooth transitions and rotations. The wheel spins to randomly select a role, highlighted to the user at the end of the animation. The script plays a ticking sound during the spin, enhancing the user's engagement and anticipation as the wheel slows down to reveal the chosen role.

Events are used to signal the start and end of the spin, allowing for custom logic to be executed at these points, such as updating the UI or processing the result of the spin. Additionally, the script includes a method to disable the picker wheel UI after a delay, further integrating the component into the broader game flow.

In summary, the PickerWheel script is a visually engaging way to select roles within a game, providing an intuitive and customizable user interface component that enhances the game's interactivity and visual appeal. (see Appendix 8)

5.6 Achievements

In the context of video game design, achievements serve as a form of player reward system, offering tangible goals outside of the main gameplay loop. These milestones are designed to extend the playtime, promote replayability and increase player engagement by rewarding specific actions. Achievements can range from simple tasks as playing as a specific class a number of times, to complex challenges as killing three or more players without dying. Implementing an achievement system can signifivantly enchance the game's appeal ensuring that there is more the base gameplay than it seems. This mechanism not only enriches the player's experience but also fosters a sense of accomplishment and proggression, making it a critical component in modern game design.

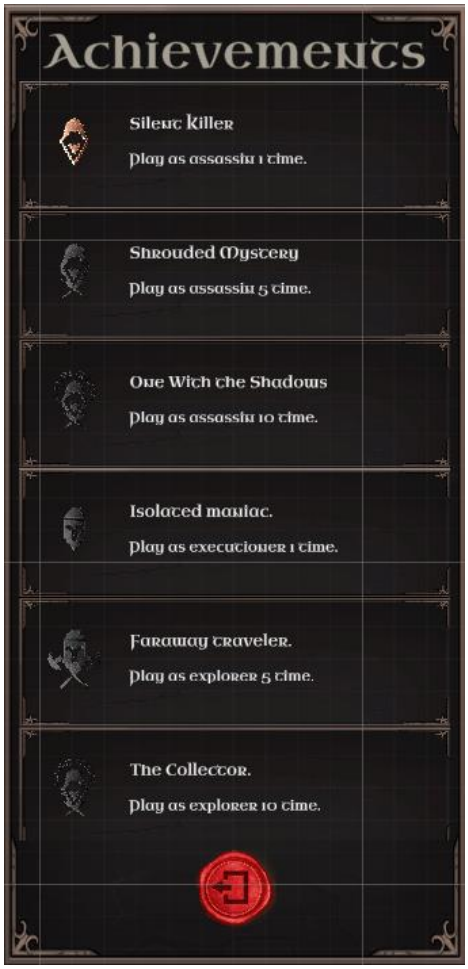


Figure 15

5.6.1 AchievementManager

The AchievementManager class is central to the game's achievement system. It adheres to the Singleton design pattern, ensuring that only one instance of the class exists and can be accessed throughout the game. This simplifies the management and tracking of achievements. At initialization, the class retrieves a set of predefined achievements and organizes them in a dictionary for convenient access. It also contains functionality for saving and loading achievements, enabling their persistence across game sessions. To ensure efficient retrieval and updates, each achievement is assigned a unique ID.

Furthermore, the class incorporates a queue mechanism for displaying achievements subsequently if more than one is achieved simultaneously from the player.

As we can see from the above images, when the class AchievementManager is initiated, first it loads the achievements that have been created, and then updates the DisplayAchievements class with the correct state of each achievement of the player according to the value of playerPrefs of achievements.

```
0 references
private void Awake()
{
    if (Instance == null)
    {
        Instance = this;

        achievementList = Resources.LoadAll<Achievement>("Achievments").ToList();

        DontDestroyOnLoad(gameObject);
        achievements = new Dictionary<string, Achievement>();
        foreach (var achievement in achievementList)
        {
            achievements.Add(achievement.achievementID, achievement);
        }
        LoadAchievements();
    }
    else
    {
        Destroy(gameObject);
    }
}
```

Figure 16

```
1 reference
private void LoadAchievements()
{
    foreach (var achievement in achievements.Values)
    {
        string key = "Achievement_" + achievement.achievementID;
        bool isUnlocked = PlayerPrefs.GetInt(key, 0) == 1;
        achievement.isUnlocked = isUnlocked;
    }
}

1 reference
private void SaveAchievements()
{
    foreach (var achievement in achievements.Values)
    {
        string key = "Achievement_" + achievement.achievementID;
        int value = achievement.isUnlocked ? 1 : 0;
        PlayerPrefs.SetInt(key, value);
    }
    PlayerPrefs.Save();
}
```

Figure 17

5.6.2 AchievementDisplay

The AchievementDisplay class is dedicated to the visual presentation of achievements. It is used to link the game's UI with the data of the achievement's scriptable object, ensuring that each achievement's status is accurately represented on-screen. The Setup method initializes the display with details like title, image and id, while the UpdateDisplay method dynamically adjusts the visual elements to reflect any changes in the achievement's status, such as transitioning from a locked to an unlocked state. This dynamic update mechanism is important for maintaining an engaging and responsive user interface, where players can see the results of their in-game actions reflected in real-time.

```
2 references
public void UpdateDisplay()
{
    iconImage = transform.GetChild(0).gameObject.GetComponent<Image>();
    titleText = transform.GetChild(1).gameObject.GetComponent<TextMeshProUGUI>();
    descriptionText = transform.GetChild(2).gameObject.GetComponent<TextMeshProUGUI>();
    disableAch = transform.GetChild(3).gameObject.GetComponent<Image>();

    if (achievement != null)
    {
        titleText.text = achievement.title;
        descriptionText.text = achievement.description;
        iconImage.sprite = achievement.achievementIcon;
        disableAch.sprite = achievement.disableAch;
        if (achievement.isUnlocked)
        {
            GameObject disableAchObj = iconImage.gameObject;
            disableAchObj.SetActive(true);
            disableAchObj = disableAch.gameObject;
            disableAchObj.SetActive(false);
        }
    }
    else
    {
        Debug.LogError("Achievement is not assigned!");
    }
}
```

Figure 18

As shown in Figure 18, whenever we create the achievement object, in the achievement window list or as a pop-up when unlocking one, the DisplayAchievement class gets the achievement prefab and populates it with values based of its own data.

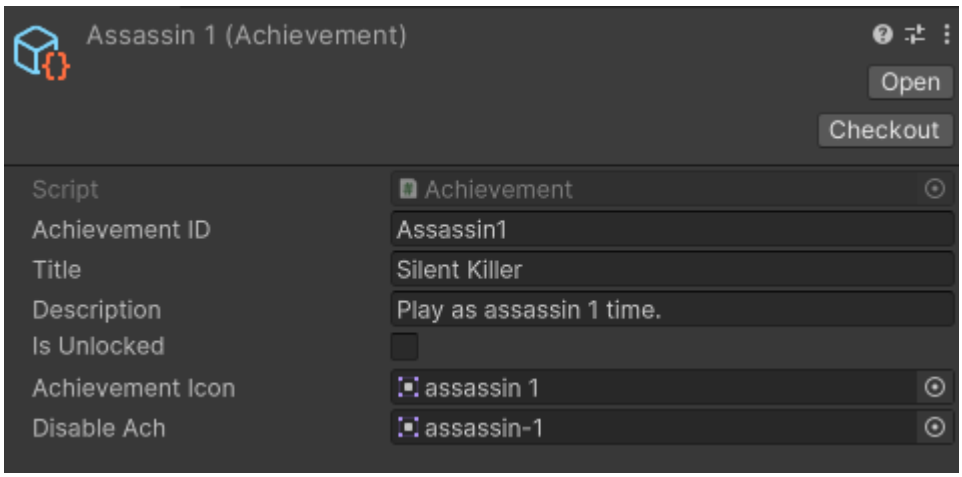


Figure 19

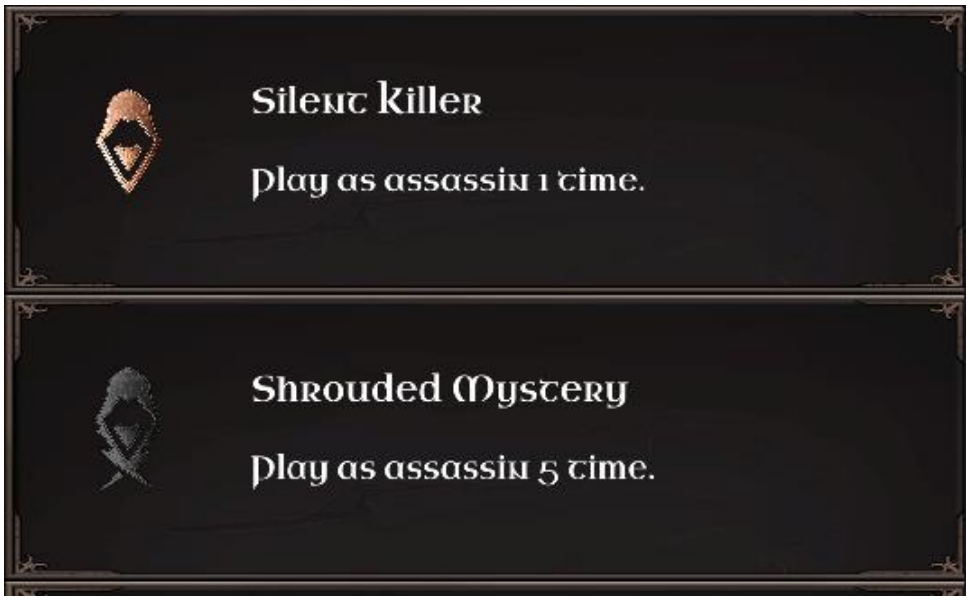


Figure 20

As you can see in the achievements list whenever we iterate to our achievements list, we call every time the method UpdateDisplay to populate each achievement prefab that's been created.

5.6.3 AchievementChecker

The AchievementChecker class in the achievement system is a proactive one as it continuously tracks down the game events and actions of players in order to know when the achievements should be unlocked. This class contains the code for certain achievement criteria, looking at the game state and player stats to check if the conditions for different achievements have been reached. For example, it could check whether the player has killed a certain number of enemies, finished a particular mission, or reached some other goals in the game that are connected to the achievement. Once the AchievementChecker has met the conditions for an achievement, it talks directly to the AchievementManager, calling methods to unlock the corresponding achievement. The achievements unlock on a direct basis with the in-game behavior. This in turn increases game engagement and depth, enabling players to explore other paths of the game, and this way they will get a reward according to their hard work.

5.7 Audio

The AudioManager class is desinged as a central hub for managing all audio-related functionalities within a game, in order to have a consistent audio implementation across all scenes. Like the AchievementManager, it utilizes the Singleton pattern to create a unique instance that persists across the game. The class is equipped to handle sound effects and music tracks, with dedicated audio source for each one, and integrates with Unity's AudioMixer to fine-tune audio levels for different audio groups. The StartSFX and StartMusic coroutines initiate playback of the initial values of sounds of SFX and music, ensuring they will start as

soon as the game starts. This class ensures that players have an immersive audio experience, with easy control over audio levels to suit their preferences.

5.7.1 Audio in game

The ExitToMainLobby method within the VolumeControl class signifies a thoughtful addition to the in-game audio settings interface, addressing the practical need for players to seamlessly transition back to the main lobby. This function encapsulates a critical piece of logic distinguishing in-game settings management from its main menu counterpart.

Upon invocation, the method first discerns the player's network role—whether they are hosting the game (server) or participating as a client. This distinction is crucial, as the action required to disconnect a player from the current session differs based on their role. For a player hosting the game, the method triggers GameNetwork.Instance.StopHost(), effectively shutting down the server and thereby the multiplayer session for all connected clients. Conversely, for players who joined the session, it is called GameNetwork.Instance.StopClient(), which disconnects them from the server while leaving the session active for others. Following the disconnection logic, the method proceeds to navigate the player back to the main menu, leveraging a scene loader utility to ensure a smooth transition.

```
1 reference
29  public void ExitToMainLobby()
30  {
31      if (Player.LocalInstance.IsServer)
32      {
33          GameNetwork.Instance.StopHost();
34      }
35      else
36      {
37          GameNetwork.Instance.StopClient();
38      }
39      Loader.Load(Loader.Scene.MainMenuScene);
40  }
```

Figure 21

Bibliography and Sources

[1] What is unity and why unity source:
https://gamedevacademy.org/what-is-unity/
[2] Feather Asset URL:
https://www.onlygfx.com/7-feathers-png-transparent/
[3] Blood Splash Asset URL:
https://www.freepik.com/free-photo/overhead-view-red-color-powder-white-background_3590117.htm
[4] Wax Asset URL:
https://www.freepik.com/free-psd/sealing-wax-text-effect_9071236.htm#query=wax&position=0&from_view=search&track=sph&uuid=080366d5-ea60-42ef-8b79-7d81c7d4ad16
[5] Menu Border URL:
[6] https://mungfali.com/post/901EB307151F48538A04C6666F1F93529E2B6398
[7] Paper and Key Assets URL:
https://walls1080.blogspot.com/2013/02/old-paper-on-key-hd-wallpapers.html
[8] Book popup Asset URL:
https://www.pinterest.com/pin/952159546197879836/
[9] Compass Asset URL:
https://photooboi.com.ua/en/wallpapers/1503
[10] Background Photos URL:
https://propnomicon.blogspot.com/2014/02/travelers-tales.html
[11] Disable Field on the Inspector:
https://forum.unity.com/threads/how-do-you-disable-inspector-editing-of-a-public-variable.142100/
[12] Make animation from sprite sheet in under 1 minute Unity 2D:
https://www.youtube.com/shorts/pllu3PMcZs4/
[13] COMPLETE Unity Multiplayer Tutorial (Netcode for Game Objects):
https://www.youtube.com/watch?v=3yuBOB3VrCk
[14] Responsive Fortune Spin Wheel - Unity Tutorial:
https://www.youtube.com/watch?v=jZ_cuxTPPsA
[15] Dynamic scroll view for achievements URL:
https://www.youtube.com/watch?v=oOox4JNQp_M
[16] How to : Integrate Google Play login into your Unity project:
https://www.youtube.com/watch?v=dbLpA2YB6vU
[17] Wind sound SFX
https://pixabay.com/sound-effects/epic-storm-thunder-rainwindwaves-no-loops-106800/
[18] Fireplace sound SFX
https://pixabay.com/sound-effects/fireplace-with-crackling-sounds-2-min-rk-178392/

Appendices