Design and Implementation of RPG Game Based on Unity

by

Yikai KONG

204603104268

A thesis submitted to the faculty of

College of Global Talents

at Beijing Institute of Technology, Zhuhai

in partial fulfillment of the requirements for the degree of

Bachelor of Engineering

in

Computer Science and Technology

College of Global Talents

Beijing Institute of Technology, Zhuhai

March 2024

Supervisor: Zhicheng YIN

ABSTRACT

This research paper aims to provide a comprehensive overview of the development process of creating an RPG (Role-playing Game). RPGs are a popular gaming industry genre characterized by immersive storytelling, character progression, and rich gameplay mechanics. Understanding the intricacies of RPG game development can help game developers and enthusiasts gain insights into this complex and evolving field. The paper begins by examining the historical background and evolution of RPG games. It explores the origins of RPGs, their transformation from tabletop games to digital platforms, and the influence of early RPG titles on the genre.

The paper delves into the key components of RPG game development. It discusses the importance of world-building, character design, narrative development, and gameplay mechanics. World-building includes creating immersive environments, designing non-player characters, and establishing a cohesive lore. Character design encompasses creating diverse and relatable player characters, non-player characters, and their progression systems. Narrative development involves crafting engaging storylines, quests, and branching paths that enrich player experiences. Gameplay mechanics range from combat systems to skill progression and exploration mechanics, shaping the core gameplay experience.

Keywords: RPG Game, Gameplay Mechanics, Character design

Contents

Figure List

Table List

Chapter 1 INTRODUCTION

1.1 Research Background

Most RPG Games consist of a big game world, immersing game stories, and a complex interactive system. Players will experience the game entirely in an immersive atmosphere. After completing the main storyline, a single-player game will lose its novelty. Most people will not choose to repeat the same game flow because the old content cannot attract players.

Therefore, the game's ability to attract players depends on the integrated game flow and the high playability of game mechanics. The game's various logical interactions or complex systems, such as the quest and backpack systems, are necessary. Including the art style, the design of the scene, and the reasonableness of the numerical values between the different characters are also essential parts of the game.

In many famous and successful RPG games, the studio will make different stories and details in the new game plus. After finishing the main storyline, many small games, extra awards, and other content in its enormous open world. It is another way to improve players' game experience. This project will provide a 2D pixel-style exploration single-player game with an open world. After the player passes the game's main storyline, they can still look back and find detailed content in the development world. They can look for side quests and Easter eggs and gradually unlock various small games independent of the main storyline to enrich the gaming experience.

1.2 Significance

This project targets gamers who like to play arcade mini-games and gamers who like adventure. The

first thing is to make sure that the game has enough adventure elements as well as enough mini-game elements. It also needed to make sure that the game was easy to pick up to appeal to players of all ages. In addition, the game runs smoothly with no lagging or crashing issues is also important. The vast majority of games on the market nowadays are overly concerned with picture quality and art while neglecting the device's adaptability and the game's fun. Interaction and gameplay are consistently among the most essential attributes of a game. Many games with complicated rules cannot bring relaxation to today's stressed young people. Similarly, games without much innovation can cause severe visual fatigue and contribute to player exhaustion. This project incorporates some easy elements into the game, hoping to differentiate it from most of the games on the market and bring a unique gaming experience to the players.

1.3 Contents

The game has a 2D-pixel world that will serve as the main world and contain various areas and scenes. The player will have a character in the main world who can do quests, level up, fight monsters, acquire equipment and leveling, and acquire equipment and props, and this is the player's main character. Players can obtain special quests or props in the main world, and players can unlock other mini-games by acquiring special quests or props in the main world. By clearing various mini-games, they can obtain higher-level equipment and skills. The player can unlock other mini-games by obtaining special quests or props in the main world. This is the core gameplay of our game, through various adventures to help players feel the joy of the game. Players can feel the joy of the game through a variety of adventures.

Chapter 2 CORE TECHNOLOGY

2.1 Unity 3D Engine

Unity 3D is an efficient and versatile game development engine that has rapidly gained acclaim among game creators for creating 2D and 3D titles across various platforms. Unity Technologies first unveiled their engine back in 2005, and since then it has grown into a comprehensive suite of tools designed to enable developers to bring their ideas to fruition. User-friendly interface and full range of features: built-in physics engine, real-time rendering engine, animation tools and an extensive scripting ***API*** (Application Programming Interface) [1]. Unity offers support for multiple programming languages, such as C Sharp and JavaScript, which allow developers to write customized scripts that control game behavior. One of Unity's greatest assets is its cross-platform support: this has made it popular among both individual developers and larger studios alike.

2.2 C# Language

C# (C sharp) is an object-oriented programming language created by Microsoft for their.NET initiative and released in 2000. Since its release, it has enjoyed widespread adoption for developing desktop, web, game and enterprise solutions from desktop apps, games and enterprise solutions all the way down to games for desktop PCs. As it offers strong typing support, automatic memory management through garbage collection as well as modern constructs like ***lambdas***, ***LINQ*** and ***async*** for asynchronous programming [2], C# provides powerful solutions.

C# is known for its simplicity and readability, making it accessible for both beginner and veteran developers alike. Its syntax resembles other C-based languages such as C, C++ and Java making learning it relatively straightforward for developers familiar with these other C-languages.

C#'s greatest strength lies in its integration with the***.NET*** framework, which offers a vast selection of class libraries for everyday tasks like file ***I/O***, networking and cryptography. Developers can utilize these libraries to quickly build robust applications.

2.3 Visual Studio

Visual Studio developed by Microsoft, is an integrated development environment used for developing various applications ranging from web apps and mobile applications, desktop programs and cloud services. Visual Studio's tools and features assist developers with writing, debugging and testing code efficiently; including its code editor, debugger tools, version control system and multi-language support; this powerful and versatile ***IDE*** (Integrated Development Environment) has become the go-to option among software engineers worldwide when developing high-quality software apps [3].

2.4 *Netcode* Network Framework

Unity Technologies recently developed ***Netcode*** for ***GameObjects*** as a networking system designed to simplify creating multiplayer games and applications in Unity. The goal is to make implementation simpler for developers by providing high-level ***APIs*** and tools tailored specifically for networking ***GameObjects*** within Unity projects. Furthermore, ***Netcode*** for ***GameObjects*** handles tasks related to synchronization, interpolation, prediction to ensure smooth multiplayer experiences [4]. This system aims to make adding multiplayer functionality accessible to developers of all levels.

Chapter 3 LITERATURE REVIEW

This paper investigates how scripting tools and innovative gameplay integration techniques can enhance player engagement and the overall gaming experience. By studying Howland's development of an interactive in-game storytelling progression using homemade script cards to aid in creating the game this project gets inspired. Additionally, Norton's [5] use of the C# State Machine to create the game's combat system and Wu's [6] creativity in integrating minigames into his project were researched. These ideas all seek to offer insight into creating compelling game content that transcends traditional boundaries. Developing games entirely through code is a tedious and cumbersome process. Various tools and techniques can significantly improve game development efficiency in the information age of rapid technological advancement. This paper examines similar tools that aid in developing a project. It examines how various tools can be used to improve the gameplay of the game itself and solve various difficulties in the development process..

Howland's [7] invention of Script Cards revolutionizes interactive story creation by eliminating the need to learn complex scripting languages. Users can express story events in any order that suits them, increasing creativity and storyline control while encouraging storytelling creativity. This approach provides a simpler, more accessible means of controlling game narrative, making the experience more engaging for players. Norton's work using C# State Machine in game development demonstrates its immense flexibility and range of combat methods and skill effects which can be realized. By creating a State Machine in C#, players can realize various actions and behaviors in-game, expanding on combat systems to provide a more diverse gaming experience. This method enhances diversity while at the same time offering dynamic combat systems to enhance combat systems for more dynamic combat experiences. Wu's ***Balance Ball*** showcases how mini-games can serve to connect disparate game elements through a main storyline. By emphasizing individual characteristics of each mini-game and integrating them into overall content of Wu's Balance Ball, players are provided with an engaging gaming experience while at the same time inspiring innovation and creativity in gameplay design.

In the paper "Role-Playing Game Studies," Zagal [8] discusses the evolution of role-playing games (RPGs) from their wargaming origins to the popular tabletop RPGs such as ***Dungeons & Dragons*** [9], and further into the realm of live-action role-play and modern computer RPGs like ***Fallout*** [10] and ***World of Warcraft*** [11]. The book delves into various disciplines such as performance studies, sociology, psychology, education, economics, and game design to explore different viewpoints and concepts related to RPGs. It also examines broader themes like transmedia worldbuilding, immersion, and player-character relationships within RPG studies. Focusing on the role-playing style, this project in Pixel-style games emphasizes gameplay and innovation over visual aesthetics. ***Guardian Tales*** [12], for instance, showcases how an RPG hand game can excel through its engaging plot and intricate gameplay design despite limitations in artwork quality. It engages players by including simple but engaging mini-games within its storyline, creating positive reinforcement for progression and keeping players interested. ***Hollow Knight*** [13], another pixel RPG, stands out for its clear visuals and direct gameplay mechanics - its well-crafted map design, impactful feedback mechanisms and robust combat system make for an immersive gaming experience for its players.

In summary, the insights from scripting tools, innovative gameplay integration, and role-playing game research are invaluable as strategies to drive game development. It is also an approach that needs to be utilized and mastered during the development of this project. These methods are incorporated into the design to create an engaging and unique game experience for the player. The future of game development lies in blending creativity, technology, and player engagement to deliver unforgettable gaming experiences.

Chapter 4 DEMAND ANALYSIS

4.1 Feasibility Analysis

Feasibility analysis is an assessment of the practicality and potential success of a proposed project or business venture. It measures various technical, economic, legal and time constraints to determine whether it can be pursued without risk to stakeholders and misallocation of resources. Through feasibility analysis, stakeholders are better equipped to make informed decisions and allocate resources efficiently. As an iterative process, evaluation of participant responses to the intervention is part of a feasibility study. It is not the end point but enables the researchers to make a decision about whether to proceed with a more controlled, larger study. Within the context of a feasibility study, however, the extent to which one can examine outcomes quantitatively and calculate effect sizes will depend on study design, sample size, and how many adaptations have been made to the study protocol [14].

4.1.1 Social Feasibility

Social feasibility analysis of creating an RPG game involves determining if its development matches up with the social needs, values and trends of its target audience and society. Below are some points to keep in mind during an RPG game development social feasibility assessment.

Target Audience and Social Trends. Identify and assess whether there is demand from your target demographic for RPG games; factors that are taken into consideration are age, gender, interests and gaming habits of this group. Social Trends means looking into current gaming industry social trends like popularity of RPG games, player preferences and emerging themes to make sure your game concept resonant with them; storyline and themes must also meet culturally appropriate appeal and

appeal to target audiences as part of this evaluation process. Consider how the game's narrative can engage players and create a social impact. The last is Ethical Considerations. Evaluate the game's content for ethical considerations, such as violence, discrimination, or sensitive topics. Ensure the game adheres to ethical standards and promotes a safe and inclusive gaming environment.

4.1.2 Technical Feasibility

Creating an RPG game requires a specific development environment based on it. The game engine used in this project is Unity, which has powerful functions to realize various interactions and systems in the game. In addition, at any time of the development of science and technology, the performance of computers is getting higher and higher, and the development of tools is also increasing. Various functions in the game can be quickly realized by using various plug-ins. Similarly, the quality of graphics and artwork in the game is getting higher and higher, and the improvement in the computer configuration makes it easy to run the game on the device.

4.2 Functional Analysis

4.2.1 Player

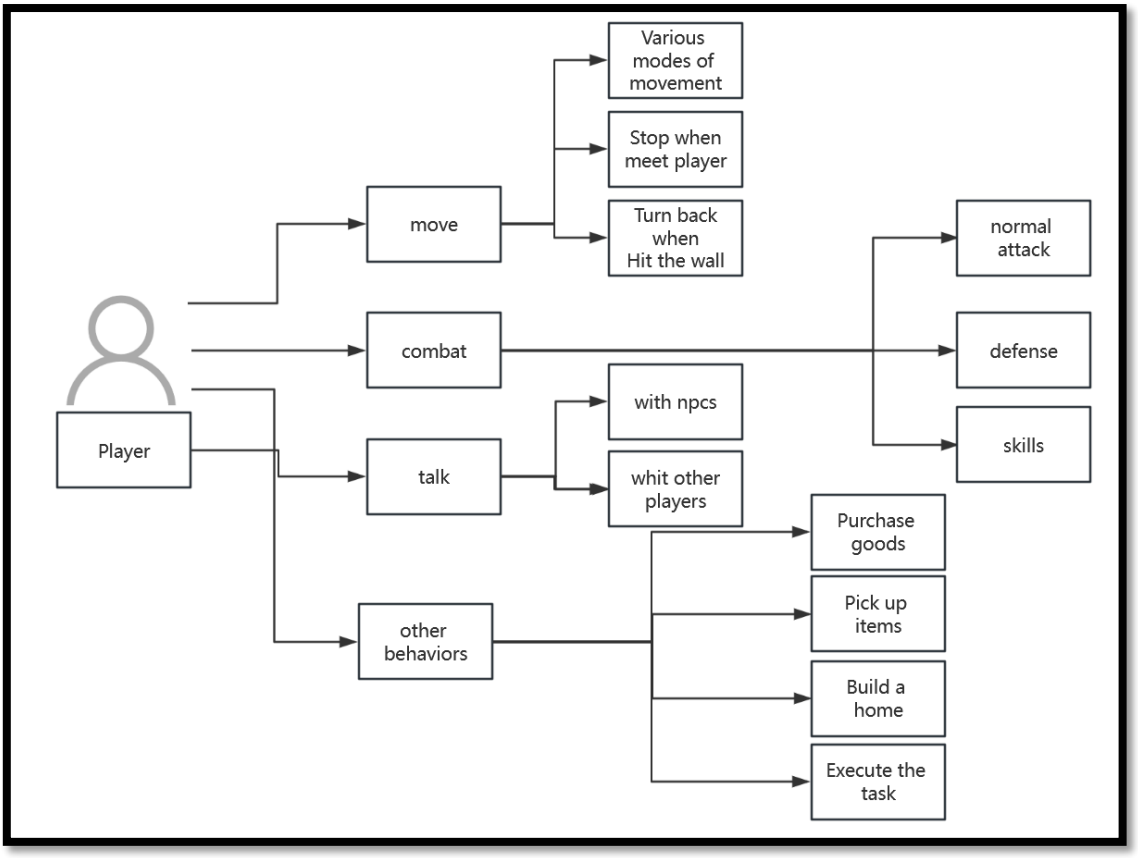

Figure 4.2.1 Player Function Diagram

This project provides the player with a variety of actions and choices to make as they navigate the game world, engage in combat, interact with NPCs, and progress through the game's storyline. Relevant contents as shown in Figure 4.2.1.

- Movement modes: There are various means by which a player can navigate within the game world, including walking, running, jumping and stopping.
- Stop when Meeting Player: This feature signifies that when encountering another player character or NPC (non-player character) within a game, your character will immediately stop moving until further instructions from either them or an NPC are given to do so.

- Turn Back When Hitting Wall: Upon colliding with walls or obstacles, this function allows the player character to automatically reverse direction and continue moving in their opposite direction.
- Normal Attack: This feature allows the player to launch basic attacks and damage enemies or characters in the game. Distinct from Attacks, defense allows the player to protect themselves from enemy attacks while reducing the damage they take from them.
- Skills: Skills are special abilities or special attacks used by the player in battle, and a variety of skills can enrich the game's combat experience.
- Purchase goods: This function allows a player to purchase goods from in-game shops using in-game currency. Purchased items can be viewed or used in the backpack at any time.
- Gather Items: Players can collect and add various props such as life potions, weapons, or quest-related supplies. Props have the potential to be extremely useful in both plot and combat.
- Build a Home: This feature allows the player to build and customize their home in the game world. Players can rest or store items inside, and even grow crops or expand their territory.
- Execute Task: Players are required to complete specific objectives or quests in the game to advance the game's plot. At the same time, the player will be rewarded for completing the mission.

In conclusion, players play a crucial role in shaping the outcome of the game through their choices and actions.

4.2.2 NPC

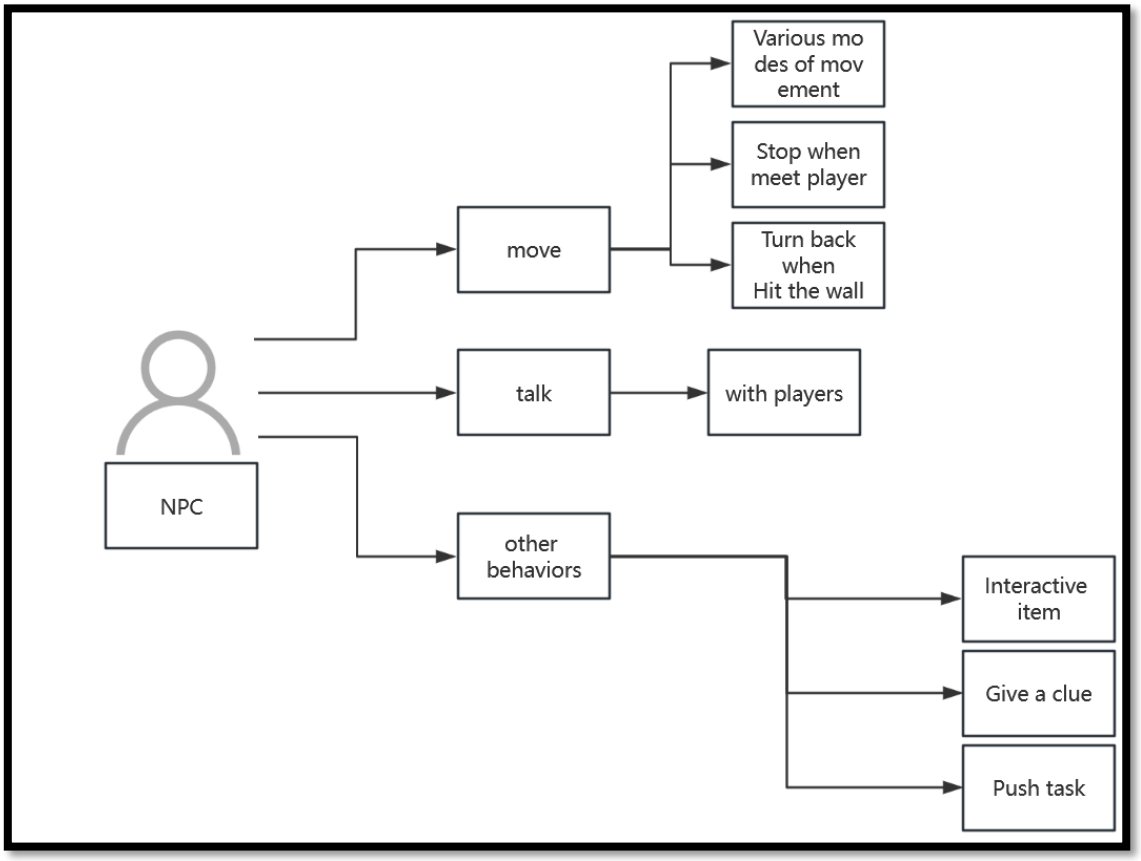

Figure 4.2.2 NPCs Function Diagram

NPCs are essential components of RPG games, providing opportunities for player engagement, storytelling, exploration and progression. Their presence enriches the gaming experience and adds depth and richness to the world they inhabit - such as those shown in Figure 4.2.2.

- ◆ Movement Modes: NPCs in the game will display a variety of movement patterns, from walking, running, patrolling and even moving to specific locations in the game world. These different movement logics add variety to the NPC's behaviors and make them more dynamic when interacting with the player.
- ◆ Stop When Meet Player: This feature instructs the NPC to stop on contact with the player character, possibly initiating dialogue, combat or other interactions. Players will often start new quests or new interactions at this point.

- Turn Back When Hitting Wall: When an NPC encounters an obstacle, this feature instructs them to turn around and randomly move to a route in another direction, rather than being trapped in their current environment. It adds realism to the NPC's behaviors and prevents the NPC from getting stuck in a dead-end loop and creating gameplay bugs.
- Chat With Players: NPCs can interact with the player character, providing useful information, quest clues or just having a simple chat through dialogue. Players chatting with NPCs tends to increase the enthusiasm and richness of the game.
- Interactive Item: NPCs can react differently to specific props in the player's inventory, creating special interactions or unlocking hidden content. The presence of props encourages the player to explore the game world further, searching for hidden clues and discovering secrets. Some props are also essential to the main storyline.
- Give a Clue: NPCs may provide the player with hints or clues to help them solve puzzles, complete quests or traverse challenging areas of the game. This feature guides do not provide complete solutions and encourage critical thinking and exploration through the game.
- Push Task: NPC's may assign quests or tasks to the player character, prompting them to complete certain objectives in order to earn rewards or progress through the game. The quest system serves as one of the main gameplay features and indirectly encourages the player to actively participate in the game world.

All in all. NPCs play an integral part of any game experience, adding depth and creating an exciting, vibrant world for the player to inhabit.

4.3 Nonfunctional Requirements Analysis

Non-functional requirements analysis is critical when developing RPGs, including performance, scalability, security, reliability, user interface design considerations, compatibility, localization, and accessibility considerations. First and foremost, RPG performance is an integral part of the player

experience. As part of the development process, the production team must consider loading speed, frame rate stability, and hardware resource usage when creating games for multiple platforms and devices. Make sure the game runs smoothly on the vast majority of devices. In addition, having unique and rich gameplay is the key to a quality game; the quality of the game should also increase with the number of players and the volume of the game. On top of that, the game system should also be flexible enough to meet changing needs quickly by integrating new features and content and providing modularity for future expansions or modifications. The game frame should possess the necessary flexibility to swiftly adapt to changing requirements by integrating new features and content, while also exhibiting modularity to facilitate future expansion or modification. Protecting players' personal information and game data is paramount in an RPG. Therefore, the development team must ensure that their game system includes adequate safeguards to prevent unauthorized access or data leakage. When developing their game's user interface design, developers should keep adaptability across devices, screen sizes, as well as user-friendly interaction design top of mind so players can easily grasp and operate it.

Chapter 5 SYSTEM DESIGN

5.1 Functional Design

5.1.1 Login Flow Chart

Once in the game, players must enter their IP address, enter the game lobby, and wait. In the game lobby, players can wait for other players; if they do not choose to team up, they can do other preparations, such as filling in their name and choosing a character. After that, players can start the game. Relevant functions design as shown in Figure 5.1.1.

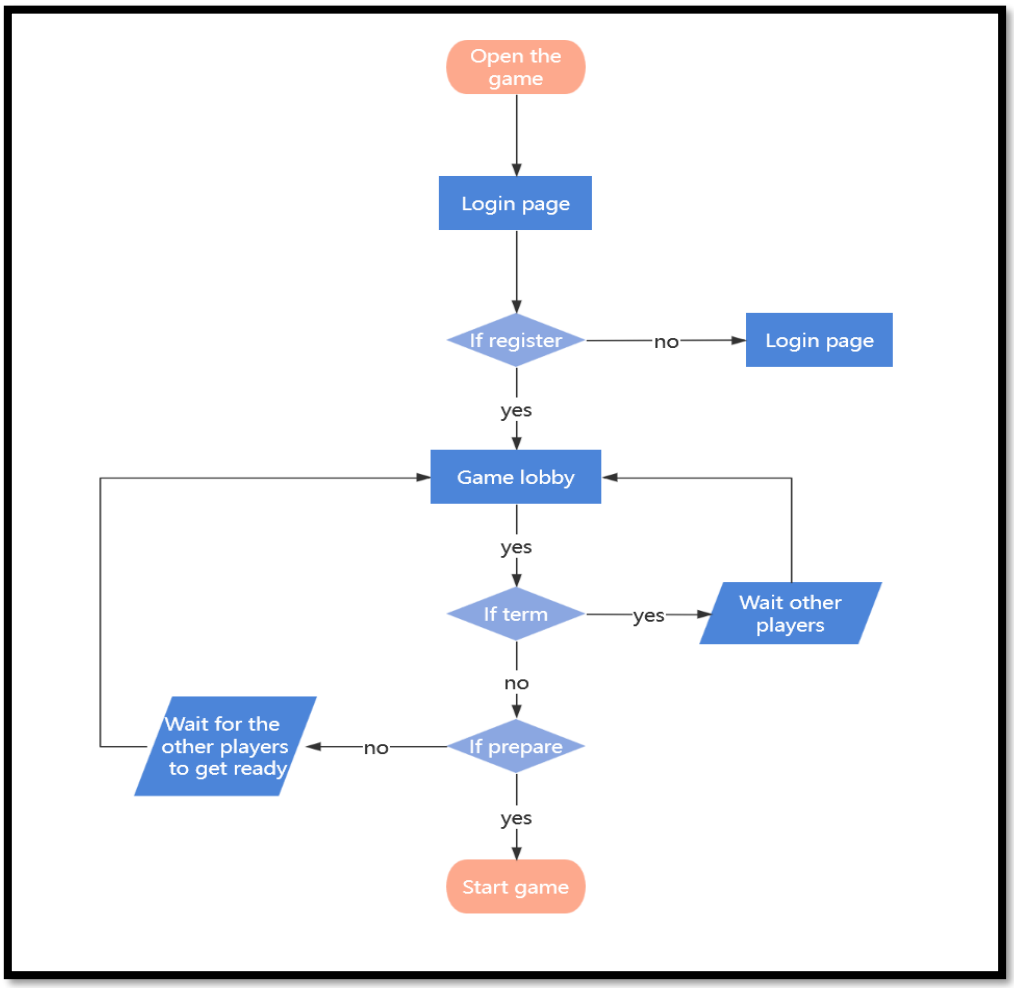

Figure 5.1.1 Login Flow Chart

5.1.2 UI

In the UI interface, players can check the attribute data, they can also continue the game or turn back the game lobby. When the player opens the UI interface to pause the game, the player can also archive and open the shop, and of course, exit the game directly. Relevant functions design as shown in Figure 5.1.2.

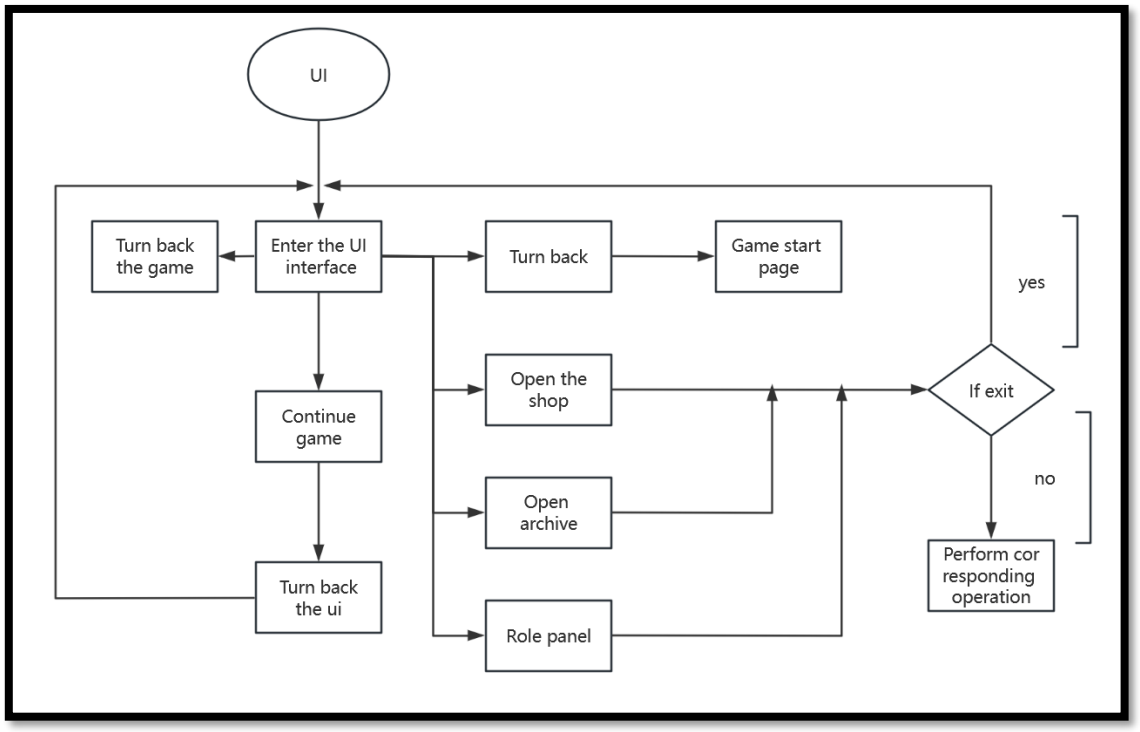

Figure 5.1.2 Game UI Flow Chart

5.1.3 Bag System

Players can open their backpacks to select different categories of compartments to view the weapons

and items they possess and equip or use. At the same time, players can also close their backpacks on each page at any time. Relevant functions design as shown in Figure 5.1.3.

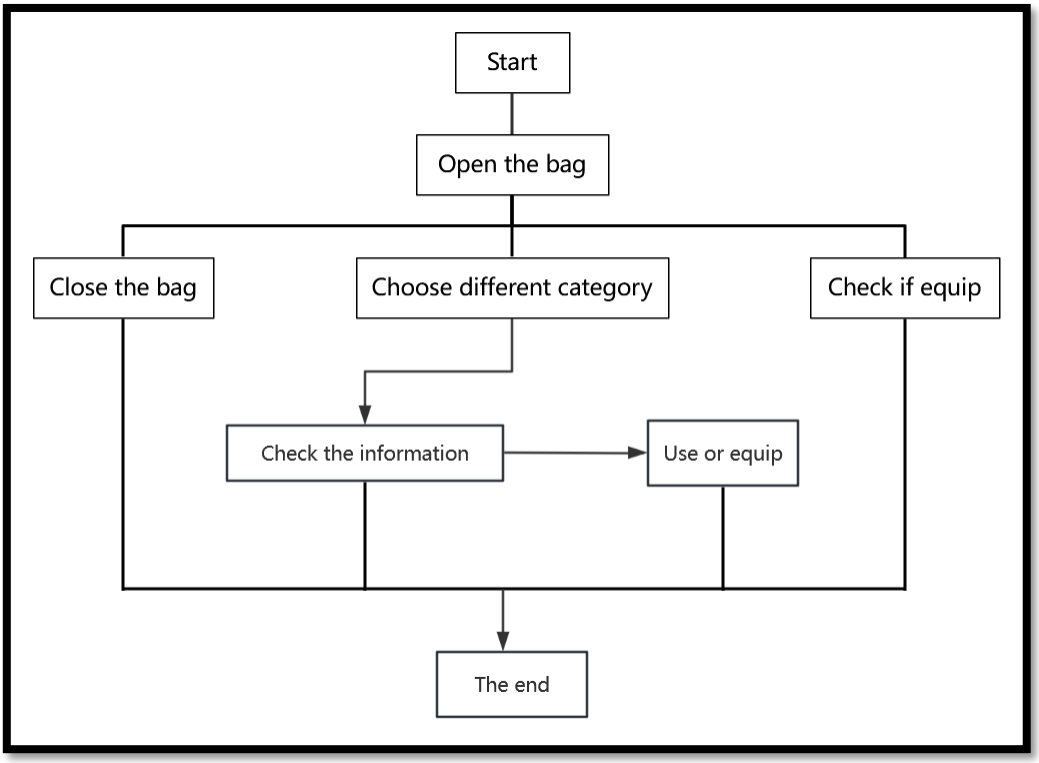

Figure 5.1.3 Flow Chart of Bag System

5.1.4 Shop System

Players can open the shop to select different categories of items, view various rich weapons and items resources along with their descriptions, and purchase them. At the same time, players can close the shop on each page at any time. Relevant functions design as shown in Figure 5.1.4.

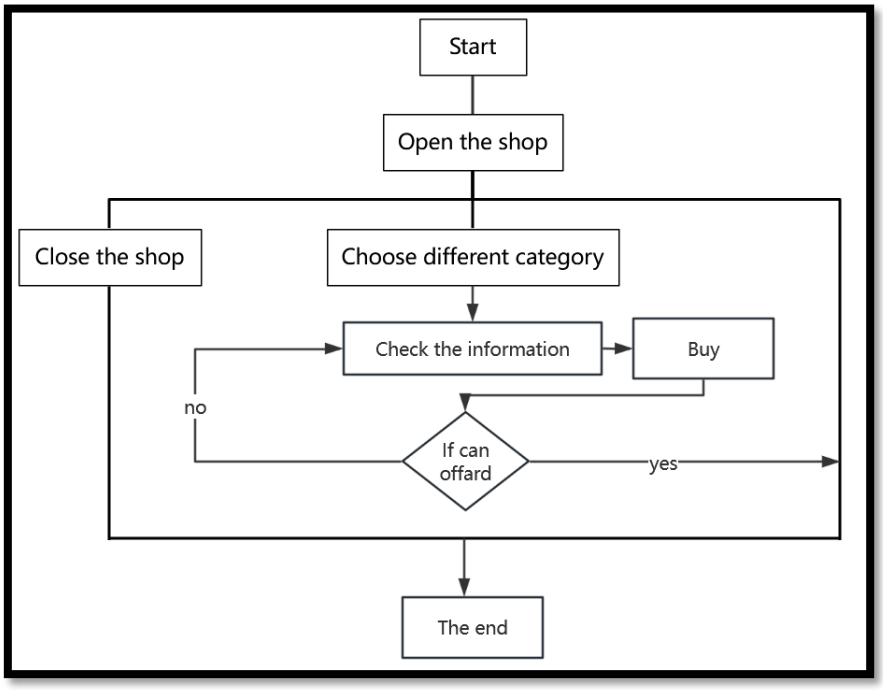

Figure 5.1.4 Flow Chart of Shop System

5.2 Function Module

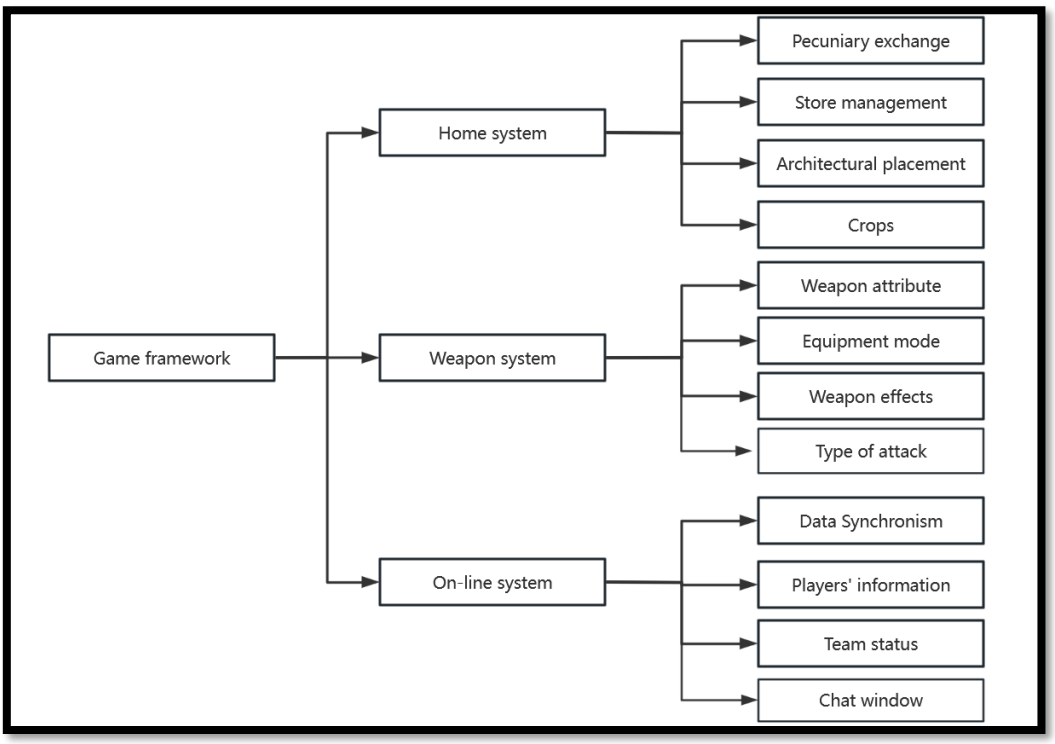

Figure 5.2.1 Function Module Diagram

This project is divided into three major game framework modules: the online, combat, and homestead systems. The scattered functions in the project will be integrated into the game framework. Relevant functions design as shown in Figure 5.2.1.

- Team Status: This function shows the current team status of players. Players initiate a team in the game lobby, and the team page will display each player's name, number, and readiness status. Only when all players are ready can the house owner start the game.
- Player Information: The player's information page provides detailed data about the player, such as the player's life value, defense, speed, intelligence, level, and other attributes. The player can use this information to make strategic decisions about gameplay.
- Data Synchronization: In an online system, the game must ensure that game data is synchronized in real time between all players. It updates the game state, player actions, map events, and the player information mentioned above. This is to maintain consistency between each player's experience and to prevent discrepancies between different players' experiences.
- Chat Window: Players can use the chat window to communicate in real-time. They can use text chat to strategize, coordinate, socialize, or seek help from other players. This promotes social interaction and player cooperation within the game itself.
- Weapon Effects: This item defines the special abilities or effects that different weapons can produce when used in combat, such as dealing extra damage, inflicting status ailments on enemies, granting buffs to players, or triggering unique animations or actions. The game has many weapon effects and a wide variety of weapon-type templates so that players can choose their favorite weapons.
- Equipment Mode: This section represents how the player should use the weapon; different weapons have different combat modes and usage. It may include factors such as whether it needs to be used one-handed or at mid-range or requires special skills or techniques to be used effectively.

- Weapon Properties: The Weapon Properties function defines the inherent characteristics of a weapon, such as damage type (slashing, stabbing, or blunt), range, durability, and weight. It may have elemental properties and those possessed by different enemies or armor types. These attributes determine how well the player performs in different environments with different weapons and how well they interact with various enemies.
- Attack Types: In addition to the weapon system's diversity, the game has multiple categories of attack types. From basic attacks and special skills or magic attacks unique to that weapon to combos and long-range attacks, the player can even have attacks that deal absolute damage.
- Crops: Players can cultivate and manage various crops in the game world. They can plant and water seeds regularly and harvest the fruits to earn game-related resources. Crop management considerations may include soil quality, weather conditions, and the best time to maximize growth.
- Architectural Placement: This feature allows players to design and build buildings, virtual homes, or various terrain structures and decorations. Building elements may include houses, workshops, gardens, fences, and so on. Players are free to build their homes as they like.
- Shop Management: Players can operate a virtual marketplace in the game world to sell or trade items with NPCs or other players. It is also possible to view and manage inventory and filter items they do not need. The shop offers a wide range of goods to meet the needs of players for various occasions in the game.
- Pecuniary: This feature deals with the economic and financial systems in the game. Players can earn, spend, and manage the game's currency or resources through questing, trading, crafting, or selling items. Players can earn gold by completing quests or defeating enemies.

Generally speaking, this project comprises three game framework modules, online, combat, and homestead systems - with the purpose of unifying scattered functions into an integrated structure. Team Status shows player team information for coordination while Players' Information offers detailed stats. Real-time Synchronism ensures real-time synchronization while Chat Window facilitates player communication. Weapon Effects define combat abilities while Equipment Mode

determines weapon usage while Weapon Attribute defines weapon characteristics; Type of Attack specifies weapon actions while Crops allow cultivation while Architectural Placement facilitates building design - while Store Management facilitates trading while Pecuniary manages in-game economy and finances.

5.3 Game Design

5.3.1 Map Design

The game map aspect uses a variety of viewpoints to switch with each other, allowing players to have different experiences on different maps. Players will be on a large map with a top-down perspective, where they can take on various quests and participate in several other activities. In the subsequent maps, the project adopts a 2D box-like approach to design so that players can feel more fun in the adventure process. Likewise, the project ensured as much coherence in art style and design for each map as possible. Relevant contents design as shown in Figure 5.3.1.

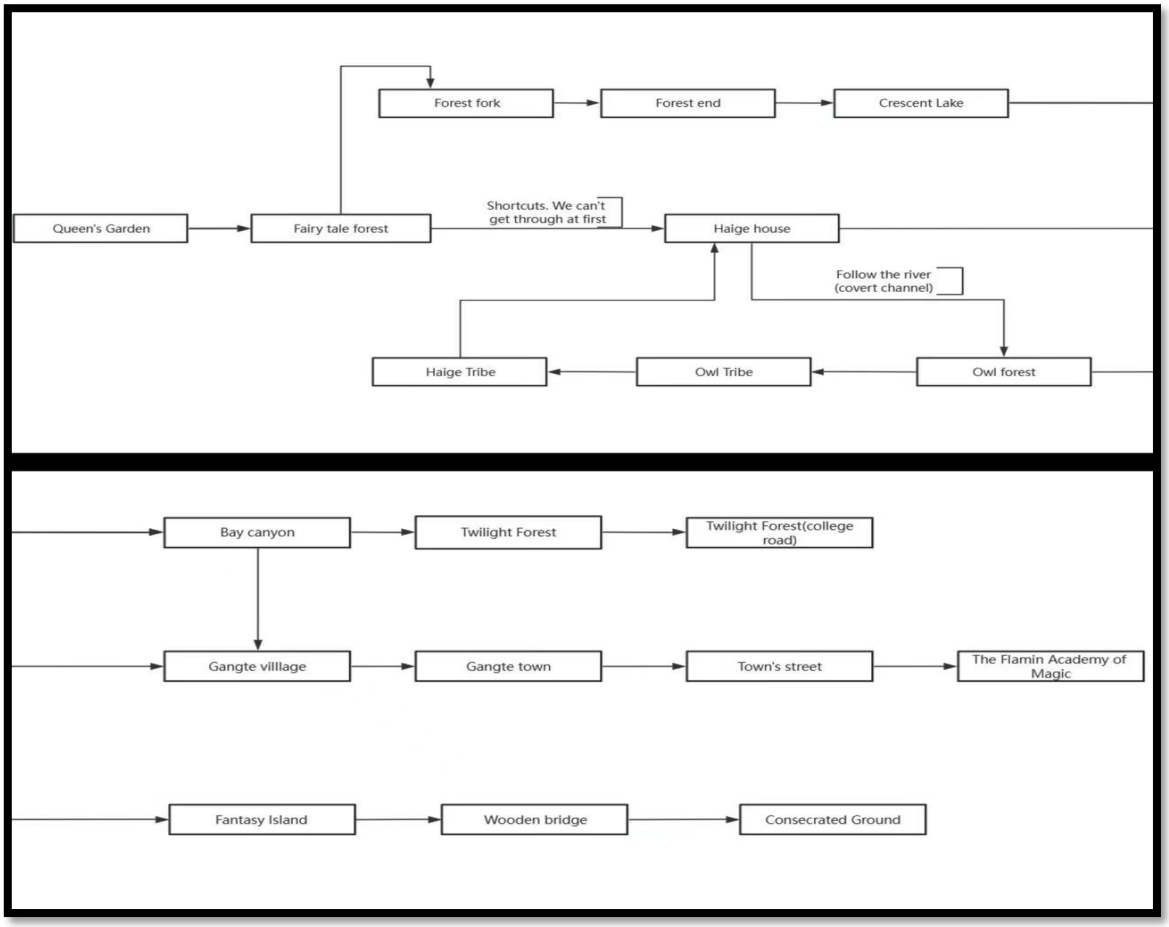

Figure 5.3.1 Map Diagram

5.3.2 Combat System Design

Players can use various means to cooperate in attacking the enemy during the battle. Initially, the game gives the player a basic long-range attack, where the player can fire bullets to deal blows to the enemy, which deal fixed and actual damage. Players will be able to unlock more and more combat tools through later stages of the game, such as melee attacks, a variety of skills and magic, as well as upgrades, enhancements, and buffs to improve themselves. Similarly, players can use a wide range of weapons for each combat tool, and different weapons will have different combat effects and panel gaps. Players cannot use the same type of equipment simultaneously. However, they can use them together to achieve better results. Details on the design of the combat style and detailed values of the

weapon system are shown in Appendix.

5.3.3 Home System Design

In the game, players can build their own homes. This includes, among other things, building houses, placing animals and NPCs, and various decorations. Similarly, the player can grow crops in the system, which can be cultivated for a certain period and in three stages to usher in a good harvest. Players can trade and use all the items they get. Not only that, but the system also includes a complete warehouse system and storage system that allows players to save their progress at any time and not lose any of its contents. Whether it is the entire home system, the warehouse, or some of the buildings, the player can upgrade them, speed up the development of their home and some of the buildings at a specific cost, and even speed up the growth of plants. In short, players can have a lifestyle of their own.

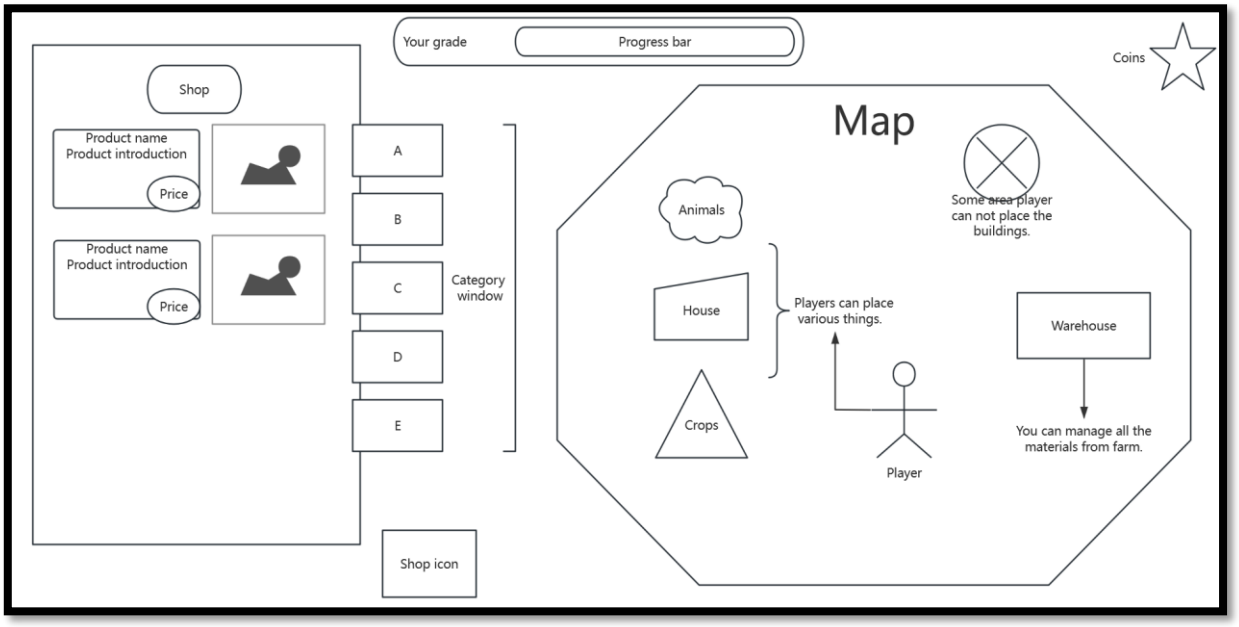

Figure 5.3.3.1 Home System

5.3.4 Task System Design

The game's task system is based on a full dialog system expansion. This project uses the Dialogue System for Unity plugin, which provides powerful dialog templates and an easy-to-use editor page to create character conversations, interactions, and quest prompts. This project modifies the source code and adjusts some of its base settings to enable the game's side and main quests to be inserted at critical points in time. This project also ensures the smoothness of the whole quest system, not only the dialogues but also the behavior of NPCs, the player's choices, and even the hidden content on the map, which will affect the whole quest. The essential props in the quest system will also be stored in the hidden backpack to drive and record the quest's progress. When conducting quest-related dialogue, players can see the name and avatar of the NPC they are currently communicating with players. At the same time, players can read the relevant dialogue content line by line and view it repeatedly to ensure that the information is complete by toggling the mouse.

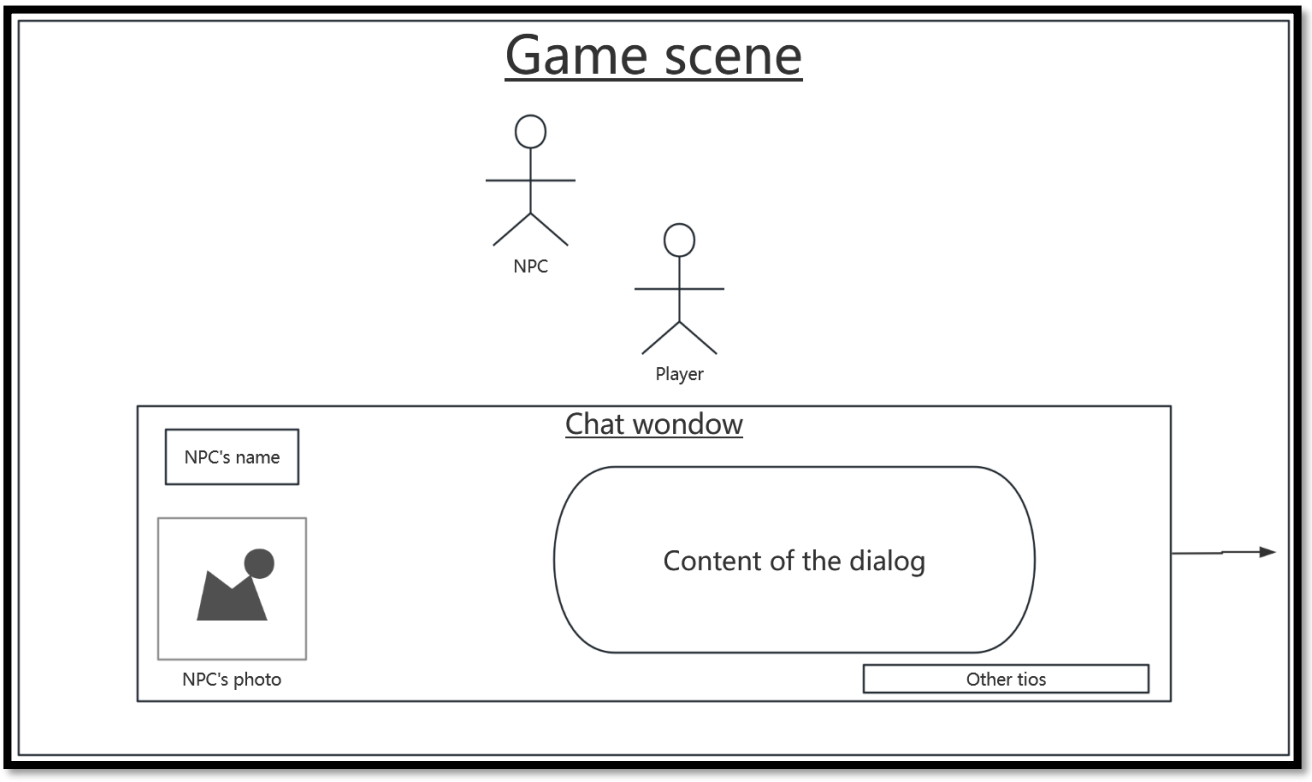

Figure 5.3.4.1 Task System

5.3.5 Team System Design

Players can choose to play solo or in a group at the beginning of the game. The first player to enter the game will automatically create a server and client in a specific IP address. Other players enter the corresponding IP address and can automatically join the server at that IP address as a new client. At this point, the data between the clients will be synchronized with each player's host. Players can edit their names on the lobby page and select a character. After everyone is ready, only the house owner can click to start the game. After entering the game, players can press 'Enter' to open the chat box and communicate.

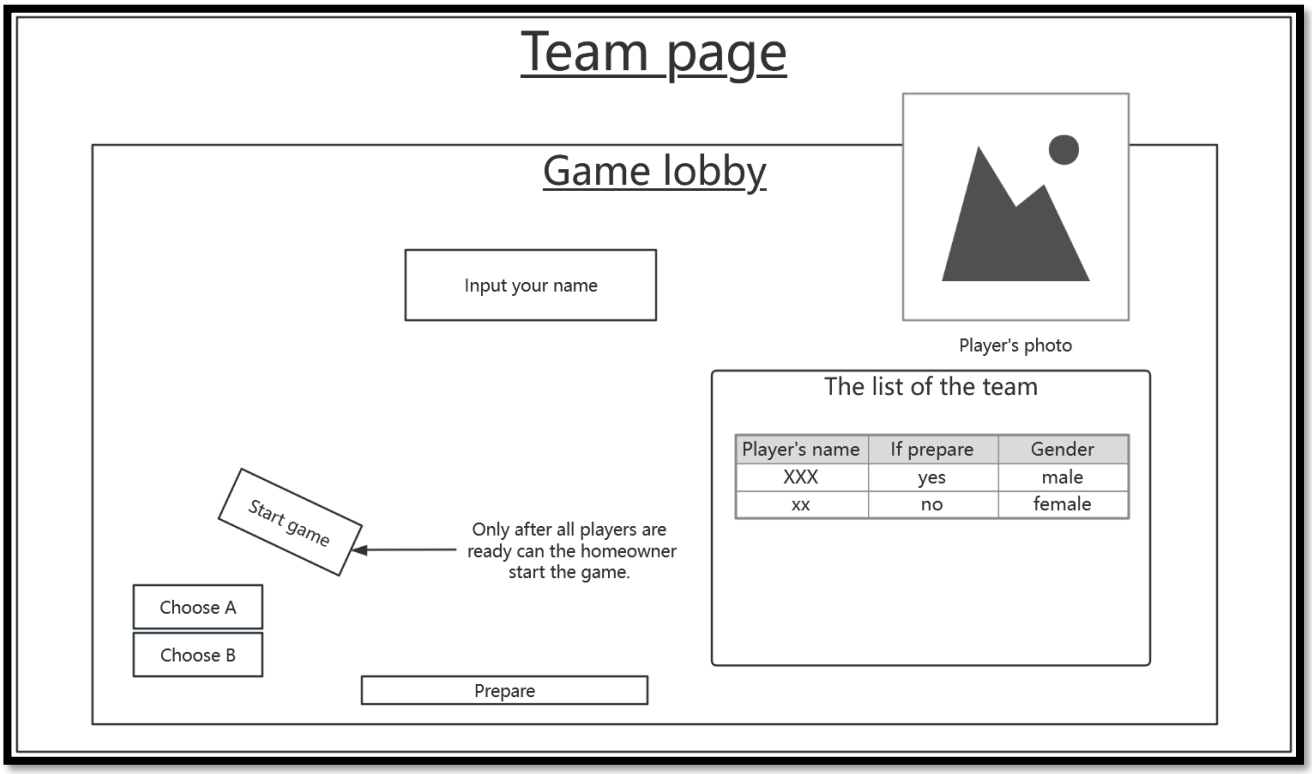

Figure 5.3.5.1 Team System

Chapter 6 SYSTEM IMPLEMENTATION

6.1 The Behavioral Logic of Players, NPCs and Enemies

In this RPG game, the behavioral logic of Players, Non-Player Characters (NPCs), and Enemies is fundamental to creating a dynamic and immersive gameplay experience. Players will have direct control over their characters, allowing for strategic decision-making and exploration of the game world. NPCs and enemies will exhibit scripted behaviors such as quest-giving, trading, and interacting with the player.

6.1.1 Player's Behavioral

The logic of player behavior is relatively complex. The first step is determining how to manipulate the player's movement. In Unity, movement is manipulated by default with the four WASD keys. Use two lines of code to listen for variables in the vertical and horizontal directions and assign the variables to custom variables like the following code. This project also creates rigid body and speed variables when writing player movement. There is also a function called Move, defined here for controlling player movement. It first accepts variables in the vertical and horizontal directions using the previously defined variable of type float. The player is then moved by modifying the rigid body's velocity (accepting two variables, x-direction velocity and y-direction velocity). Adding the Move function to Update to run every frame. At this point, the player can move.

$$movement.x = Input.GetAxisRaw(\text{“}Horizontal\text{”});$$
$$movement.y = Input.GetAxisRaw(\text{“}Vertical\text{”});$$

In the 3D part of the project, in addition to moving in one more direction axis to be considered, the

project also implements a jump function for the player. The first step was to set the affirmative button for jumping to space in the input manager and change the collision detection of the character's rigid body to continuous to prevent the character from getting stuck in the wall when jumping and landing. In the following code determines if the player has entered a jump button that will change the character's speed in the y-direction after it is entered, save, and test. This project adjusts the jumping force to make it jump higher and adjusts the gravity to make its falling speed faster and more realistic. The jumping code showing as Table 6.1.1.

Table 6.1.1.1 Jumping Code

```
if (cc.isGrounded)
{   if (isjumping)
    {
        m += Vector3.up * jumpHeight;
    }
    cc.Move(m * Time.deltaTime);
}
else
{
    velocity.y -= Gravity * Time.deltaTime;
    cc.Move(velocity * Time.deltaTime);
}
```

This project also studied the camera view when the player moves. The plugin ***Cinemachine*** was used to solve the complex math and logic of tracking targets, composing, blending, and cutting between shots. The number of time-consuming manual operations and script revisions during development was significantly reduced. In 3D projects, the distance dragged by the mouse on the screen is converted into an angle by obtaining the player's position information and Euler angles in real-time. We also obtain the coordinates of the camera for each frame, calculate the angle between the two points and the horizontal direction, and update the relative position of the camera in real-time so that we can get the camera's angle of view that rotates freely with the mouse and is centered on the player referring to the following Table 6.1.1.2.

Table 6.1.1.2 Free Perspective

```
FollowedCamera = GameObject.Find("Virtual Camera").transform;
// Getting the camera position
var forward = Vector3.ProjectOnPlane(FollowedCamera.forward, Vector3.up);
var right = Vector3.ProjectOnPlane(FollowedCamera.right, Vector3.up);
var m = (vertical * forward + horizontal * right).normalized * speed;
transform.forward = Vector3.Slerp(transform.forward, m, 0.3f);
```

Compared with the 3D part, this project can quickly realize the function of the camera tracking the player's movement in the 2D part by using the tracking object function with ***Cinemachine***. Adding the component ***Polygon Collider Confiner*** (as shown in Figure 6.1.1.3) to the ***Extension*** property of ***Cinemachine*** to realize the reasonable camera boundaries on different maps so that the player can have a more reasonable and comfortable experience when controlling the character.

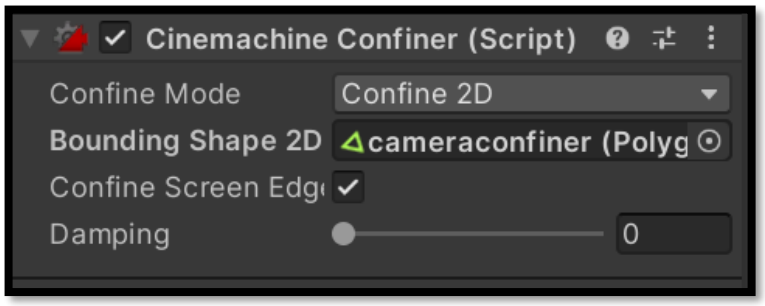

Figure 6.1.1.3 Polygon Collider Confiner

In the game, the player blinks and is invincible for a short period when injured. This is so that the player has enough reaction time to deal with relatively dangerous situations. The project has collision detection to determine if the player is hurt. When a specially labeled object touches a trigger nested on the player's body, the player's life value will be reduced under the ***OnTriggerStay*** method. At the same time, the player will enter the invincible state for 1.5 seconds [15].

Using a ***Boolean*** to judge and when the player is invincible, the life value will no longer be affected, and at the same time, the blinking code will be executed. Get the player's picture information and

control its transparency every short period for 1.5 as shown in following Table 6.1.1.4. The ***numBlinks*** represents the number of flashes per injury, and ***seconds*** represents the amount of time it takes to flash each time. When the player's life value runs out, he will die and return to the fixed resurrection point. Meanwhile, he will recover all his life value. This project mounts the death animation to the player as a UI component. Once the player dies, immediately use the Destroy function to destroy and play the death animation. At the same time, jumping to the corresponding scene, the player as a prefab will be reloaded.

Table 6.1.1.4 Invincible after Injury

```
IEnumerator DoBlinks(int numBlinks, float seconds)
for(int i = 0; i < numBlinks; 2; i + +)
{
myRender.enabled = !myRender.enabled;
yield return new WaitForSeconds(seconds);
}
myRender.enabled = true;
```

6.1.2 NPCs' Behavioral

The game is set up with a variety of NPC behavioral logic patterns. The central role of NPCs in the game is to drive through dialogue to provide or offer appropriate clues. Besides, NPCs also have logical behaviors regarding movement and some special functions outside the plot to enrich the game content. Therefore, different particular behaviors of NPCs are represented in the script's processing with the switch function. Changing the current vector's direction and size in the NPC's movement script can change the NPC's movement direction when it touches a wall or a player. In addition, the movement logic between different NPCs is also different; there are five different NPC movement logic modes in the game. Some memorable NPCs have goodwill, teaming, dialogue, and other functions. Change the current state of an NPC by setting potential values to the NPC. When the

player touches the NPC, the program turns off the movement component and pops up a UI selection box for the player to choose options. The player's choice to have a dialogue with the NPC, the dialogue content is extracted and written in the script mounted on the NPC, including doing some actions to increase or decrease the goodwill; all of these results also change the NPC's values, and of course, all of these numerical attributes are recorded in the NPC's script as well. If the player chooses to group, then the NPC will follow the player and move; this can be achieved by obtaining the position of the player and the NPC every frame and delaying for several seconds to change the direction of the NPC's movement vector. As shown in the Figure 6.1.2, it shows the logic of movement of various NPCs in the game.

Figure 6.1.2 NPCs' Movement

6.1.3 Enemies' Behavioral

The logical behavior of enemies is similarly divided into four, and like NPCs, the moving parts of enemies are similarly represented in various ways. Using two ***Boolean*** values as switches, a timer is

also set to change when the movement moves to stop and when it turns. Unlike NPCs, each enemy foe has a trigger hooked up to detect various attacks from the player. The player's behavior changes the state of the enemy's life value. Similarly, enemies will have logical behaviors when attacking the player. The relative position of the player to each enemy is obtained to determine whether or not to attack the player. This project uses sequence frame animation and skeletal animation to express the various states of the enemy's attack and sets the relevant event at a specific animation frame. Let us say the enemy needs to attack the player; then, in those frames when the enemy's weapon falls, the event needs to be triggered with a damage judgment event, which also uses the method of switching triggers so that the player will be hurt if he is in the range of the enemy's attack triggers. As shown in the Figure 6.1.3, it shows variety of enemies.

Figure 6.1.3 Variety of Enemies

6.2 Weather System

This project uses the ***WeatherClock2D*** plugin to create an in-game weather system. ***WeatherClock2D*** has several built-in interfaces that allow the user to manage the time and weather

of a 2D game through a single ***gameObject*** and script. The project uses ***WeatherClock2D*** to significantly save development time and provide environmental behavior to the game. ***weatherClock2D*** contains a scene, a set of particle systems, and prefabs. This project uses the plugin to freely define time and simulate it. In addition, it is possible to customize seasons, durations, environmental phenomena, and the possibilities of these phenomena, and even adjust the light changes caused by each phenomenon. This project also uses the most important features of ***WeatherClock2D*** real time Clock and analog clock to edit the number of seconds per minute, count the seconds, minutes, hours, days, weeks, and months, and edit the number of days per month, and to set the light colors for the different periods. Also included is the ability to edit the start and end month of each season, attach particle systems for phenomena and the probability of starting each phenomenon, cancel seasons and define particle systems for phenomena as well as the probability of unused seasons, and add phenomenon colors to each ***Particle System*** to blend daylight with phenomenon light [16]. In addition, it also can customize the number of months that you want to define along with the name and days per month for each of them. The following Figure 6.2.1 shows an example of this customization by adding the 12 existing months with their names and days.

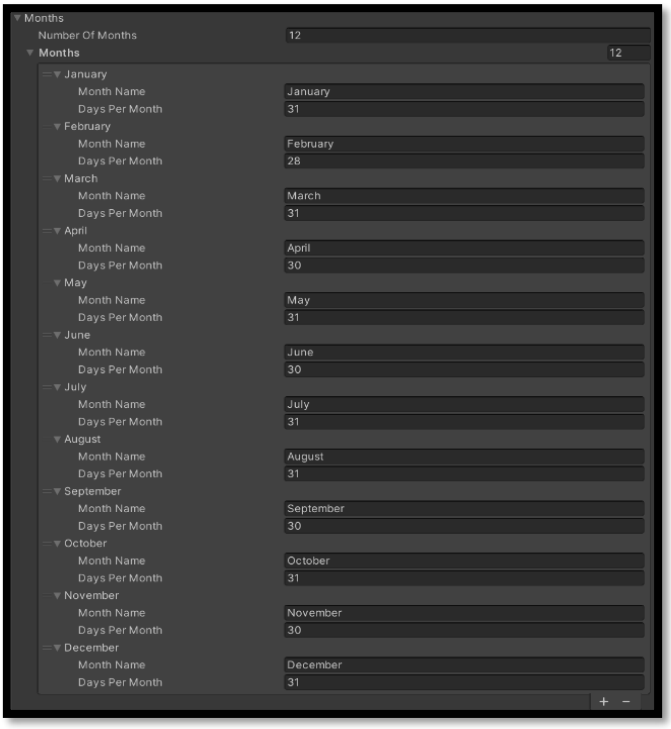

Figure 6.2.1 The Interfaces in ***WeatherClock2D***

After attaching ***Light 2D*** to the script to control the light changes, the project chose to simulate the time with ***WeatherClock2D***, customizing the number of seconds per minute according to the player's needs to ensure that the time loop in the game is correct. Subsequently, using the plugin's versatile interface, it was possible to adjust the morning, evening, and night colors to bring the light to life and reflect the different periods. Additionally, each month's number, name, and day can be customized to suit the game's needs. In the Seasons section, choose whether or not to add seasons and configure them accordingly or even utilize interpolation factors to achieve the effect of dynamically adjusting the light according to the current weather. Next Figure 6.2.2 shows four examples of rainy, windy, sunny, cloudy, and even four seasons of various weather scenarios.

Figure 6.2.2 Weather System

6.3 Dialog System

The primary role of the Dialog System in RPGs is to advance the plot. Most RPGs are filled with

much textual content, so it is crucial to implement a flexible and generic dialogue system during development. This project uses a third-party plugin, ***DialogSystem***, to assist with this function. The plugin integrates many functions and UI interfaces that allow for ready and intuitive adjustment of the various components of the system. The basic functionality of this system is that the ***DialogueManager*** searches for a ***DialogueTrigger*** with a trigger box. If they collide, the ***DialogueManager*** sends itself to be processed by the ***DialogueUl*** component. The ***DialogueUl*** component displays sentences and images on the screen [17]. The task of this component showing as the following Figure 6.3.1 is to trigger the dialog, so this is meant to put on the player. To work this needs a ***Collider*** and a ***Rigidbody*** component or alternatively a ***CharacterControllelcomponent***.

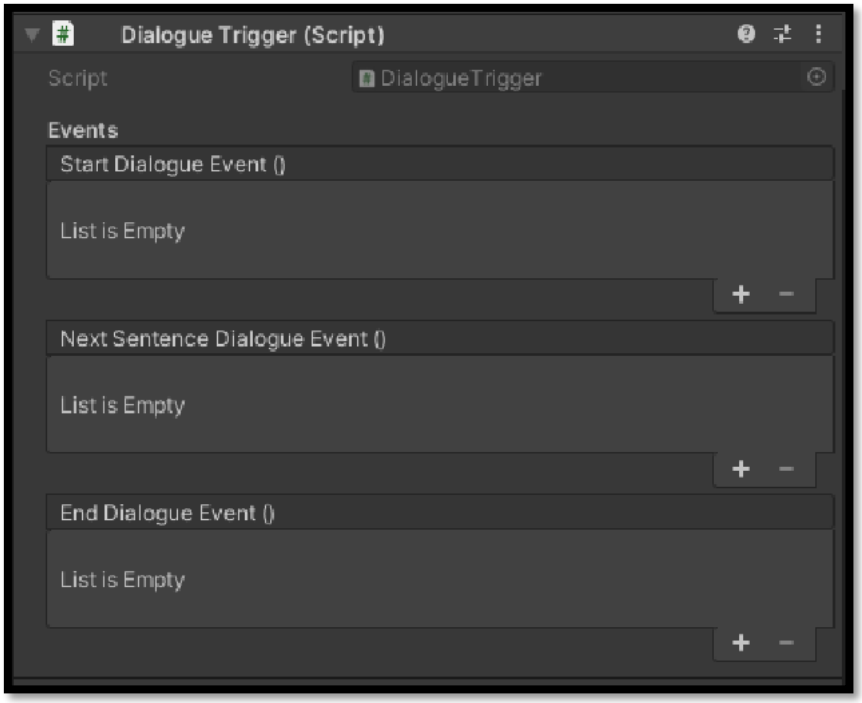

Figure 6.3.1 Dialogue Trigger

Because of the powerful interface integration of the Dialogue System. Each sentence element has ***Dialogue Character***, message, Unity event, and a sound clip. The component ***Dialogue Manager*** is intended to be placed on NPC characters that are supposed to talk to the player. In order to flexibly deal with various unexpected events encountered in the mission and make the game plot more diversified, this project will add corresponding critical event scripts in the following three method

codes to change the behavior of NPCs or players in different states. The behavior under the first method will be executed when the specified target starts the dialogue. The content under the second method will be executed after the previous dialogue. Similarly, for the specified target of the dialogue, the developer also customized the previous dialogue with the availability of the previous dialogue as a variable. The script actions under the third method will be executed after the specified target ends the dialogue.

$$Public\ void\ StartDialogue()$$

$$Public\ void\ NextSentence(out\ bool\ last\ Sentence)$$

$$Public\ void\ StopDialogue()$$

Figure 6.3.2 shows the effect of a player talking to an NPC in a 3D hall scene and the chat window will appear on the bottom of screen.

Figure 6.3.2 Chat with NPC

6.4 Home System

6.4.1 Currency System and Experience System

In order to implement the currency system and experience system in the home system. The project created a script called ***GameEvent*** to be used as an abstract base class for all events and defined five or three types of events: currency change, experience increase, and level change. The game utilizes a third-party plug-in ***Easy Save*** to aid in developing the home system. The project created two scripts, ***CurrencySystem*** and ***ExperienceSystem***, to manage the currency and experience systems. They are linked together in the ***GameManager*** script [18]. Easy Save provides the ***GameManager*** script.

The currency system creates ***Enum*** to store currency types such as gold and crystals. Also, ***Dictionary*** can store all currency values and create an object list for UI initialization. Listen for currency changes and shortages through events under the corresponding methods and subscribe to these events in the ***Start*** method. The experience system defines the current experience value and the level of experience required to get to the next level. Like the currency system, the script uses static ***boolean*** values and ***Dictionary*** to store level experience and rewards. To implement the leveling process, the game initializes the experience and reward Dictionary by reading the ***CSV*** (Comma-Separated Values) file, updating the experience bar, and remaining experience text in the UI. The effect picture of currency system and experience system showing as the Figure 6.4.1.

Figure 6.4.1 Currency System and Experience System

6.4.2 Building System

This section implements the player dragging objects from the shop onto the map, and after a certain point, the shop closes and instantiates the object. After that, the player can drag the object on the map and release it at the right location to place a building in the current scene. This project also implements a grid adsorption feature. After placing it, the player can edit the building, long-press the object to enter building mode and move it. The system uses a ***tilemap*** to keep track of available areas. If a tile on the ***tilemap*** is colored, the area is occupied; if the tile is empty, the area is available for placing objects [19]. In this system, prefabs are frequently used and saved in local files, and each building has a pre-defined range size. For example, building A occupies 2x3 tiles, but building B occupies 3x4 tiles. When placing an object, the system checks to see if the area underneath is clear, and if so, it can be placed. After the player has placed the object, the system will color in the ***tilemap*** to indicate that the area is occupied. The effect as shown in Figure 6.4.2.

Figure 6.4.2 Placing Building

In order to implement the above functionality, the project defines a public variable, ***building prefab***, under the ***Building System*** script, which stores the prefabricated parts of the building that the player wishes to place. When the player clicks the left mouse button, the script detects this event and

instantiates a new building object at the mouse position. At this point, the object is movable, and the player can change its position in the scene by moving the mouse. Next, the ***Update*** method is used to write the logic script that is responsible for listening to the player's input. If a right mouse click is detected, the script calls the ***PlaceBuilding*** method to place the building. In this method, additional logic can be added to check if the selected location is suitable for placing the building, such as prioritizing whether the terrain or other buildings in the game will prevent the placement before running the method. Finally, once the building has been placed, the script clears references to the current building, allowing the player to select and place the next building. This process can be repeated to create complex building layouts in the game. Showing as the following sample code Table 6.4.2.

Table 6.4.2 Placing Buildings

```
public class BuildingSystem: MonoBehaviour
{
  public GameObject buildingPrefab; // Building prefab
  private GameObject currentBuilding; // Buildings which currently being placed

  void Update()
  {
    if (Input.GetMouseButtonDown(0)) // Detecting left mouse clicks
    {
      if (currentBuilding == null)
      {
        Vector3 mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        mousePos.z = 0; // Ensure that the building is on a 2D plane
        currentBuilding = Instantiate(buildingPrefab,mousePos,Quaternion.identity);
      }
    }

    if (currentBuilding != null)
    {
      Vector3 mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
      mousePos.z = 0; // Updated building locations
      currentBuilding.transform.position = mousePos;

      if (Input.GetMouseButtonDown(1)) // Detect right mouse clicks
```

```
        {
            PlaceBuilding(); // Place the building
        }
    }
}

void PlaceBuilding()
{

// This project adds a logic script to this method that checks if the building can be placed in its current position.
// If it can be placed, the scene data will be updated, and the building position will be fixed.
    currentBuilding = null; // Clears the current building
}
}
```

6.4.3 Planting System

The crop planting feature is an essential part of the homestead system, which simulates the process of sowing to harvesting crops in the real world. The project starts by defining areas where crops can be planted in the game scene and identifying plots where players can interact with them. Next, a planting system is created to manage the planting process. This system must detect player input, such as click or drag actions, and generate crops on the appropriate plots. This part is similar to the principle of placing buildings earlier. The most important part of this feature is the crop growth logic. This involves tracking time and the state of crop maturity and updating the crop's visual representation as it matures. In order to allow the player to interact with the growing system, the project created a user interface through which the player can select the type of crop to be grown and then drag and drop it onto a plot in the scene [20]. Also, the gas pedal mentioned above appears with the UI after the crops have been planted and is implemented by tracking the time difference between each frame using Unity's ***Time.deltaTime***. Accelerating the crop harvesting process can be achieved. Crops usually have multiple growth stages, such as germination, growth, and maturation. Each stage will have a different visual representation, such as a different texture or model. This project does not

use a ***State Machine*** to manage these growth stages. Instead, we prepare the image materials for each stage in advance and switch the image properties of the GameObject, the crop, at the appropriate time. The implementation code in Table 6.4.3 is as follows.

Table 6.4.3 Crops System

```
public class CropGrowth : MonoBehaviour
{
  public GameObject[] growthStages; // Modeling of different growth stages of crops
  public float growthDuration = 30.0f; // Total time from sowing to maturity of the crops
  private float growthTimer; // Timer to track crops growth
  private int currentStage; // Current growth stage

  void Start()
  {
    growthTimer = 0.0f;
    currentStage = 0;
    UpdateCropModel();
  }

  void Update()
  {
    // Simulation of crops growth
    if (currentStage < growthStages.Length - 1)
    {
      growthTimer += Time.deltaTime;

      if (growthTimer >= growthDuration / growthStages.Length * (currentStage + 1))
      {
        currentStage++;
        UpdateCropModel();
      }
    }
  }

  void UpdateCropModel()
```

```
    {
        // Updating crops models to reflect current growth stages
        for (int i = 0; i < growthStages.Length; i++)
        {
            if (growthStages[i] != null)
            {
                growthStages[i].SetActive(i == currentStage);
            }
        }
    }
}
```

In this script, the ***growthStages*** array contains models of the different growth stages of the crop. ***growthDuration*** variable defines the total time required for the crop to mature from sowing to maturity. growthTimer is used to keep track of the growth time of the crop and the currentStage variable indicates the current growth stage. In the Update method, the script updates the ***growthTimer*** with the time and the crop model by calling the ***UpdateCropModel*** method when the time required for the next growth stage is reached. The effect as shown in Figure 6.4.3.

Figure 6.4.3 Planting Crops

6.4.4 Storage System

A storage manager mainly unifies the storage system in the home system. It will keep track of all collectibles within the game, which are all abstract ***ScriptableObjects***. All in-game item types in this system are based on an abstract ***ScriptableObject*** called ***Collectible Item***, which contains standard fields for the item, such as name, description, and unlock level. Regardless of their type, the ***StorageManager*** communicates with Barns and Silos. ***StorageBuildings*** in the system inherit from ***PlaceableObjects*** and will have UI windows to display items. The following Figure 6.4.4 shows the specific class structure diagram of the system.

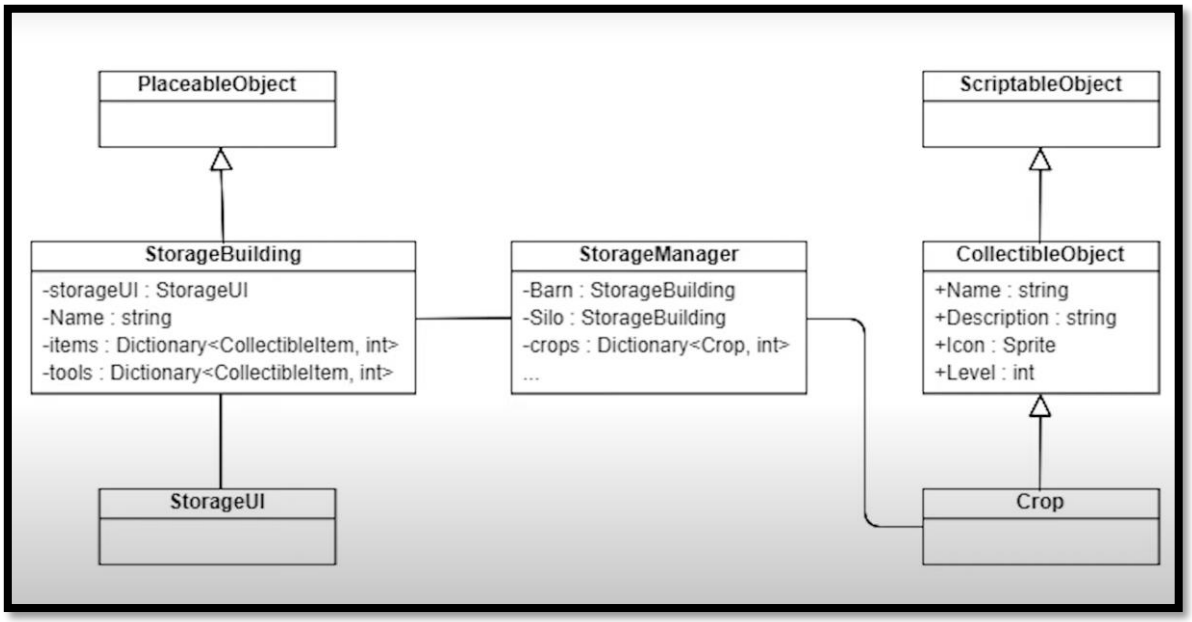

Figure 6.4.4.1 Storage System Diagram

This project uses scripts to implement the logic of item storage and the UI interface to manage the storage building. The specific logic of storing items is similar to the backpack system in the game. The following sample code Table 6.4.3 shows the process of managing the storage system for this project. The ***StorageBuilding*** in the code inherits from ***PlaceableObject***, indicating that it can be placed in the game. The ***[SerializeField]*** property is used to expose private fields in the Unity so that

the ***storageUIPrefab*** and ***buildingName*** can be set manually, whereas the ***name*** property is used to store the name of the building, which has a private setter so that it can only be assigned from within the class. The ***Initialize*** method accepts a string parameter ***name***, which is used to initialize the building name and the UI window. It creates an instance of ***storageUIPrefab*** and sets it inactive. This allows the player to switch the storage UI through specific actions. Finally, the ***OnMouseUpAsButton*** method is a built-in Unity method used when the user clicks on the object. In this method, if the ***storageUIInstance*** is not empty, it is activated, and the ***Initialize*** method is called to initialize the UI.

Table 6.4.4 Storage System

```
public class StorageBuilding : PlaceableObject
{
    [SerializeField] private StorageUI storageUIPrefab;
    [SerializeField] private string buildingName;
    private StorageUI storageUIInstance;

    public string Name { get; private set; }

    private void Initialize(string name)
    {
        Name = name;
        GameObject window = Instantiate(storageUIPrefab.gameObject, FindObjectOfType().transform);
        window.SetActive(false);
        storageUIInstance = window.GetComponent < StorageUI > ();
    }

    private void OnMouseUpAsButton()
    {
        if (storageUIInstance != null)
        {
            storageUIInstance.gameObject.SetActive(true);
            storageUIInstance.Initialize(Name);
        }
    }
}
```

The UI interface of the warehouse is shown in the Figure 6.4.4.2, which shows several crops that already exist for the player. Players can exit the warehouse interface via the button in the upper right corners, or they can click the white button below to perform subsequent processing operations on the already existing stocks.

Figure 6.4.4.2 Storage

6.5 Combat System

The combat system in this program is well-developed and complex. Players can use various weapons, props, jewelry, gauntlets, and so on. in the game. Each object class will have a separate script mounted to control them. Take weapons as an example. In order to make different weapons show different effects, each weapon will be mounted on the corresponding script, and the corresponding logic and implementation will also be different. The various weapons in this project are permanently mounted on the player as GameObject, and the ***bool*** value is changed to change the state of whether the player is equipped with a particular weapon. Therefore, the player can obtain or equip a weapon only when certain conditions are reached, and the ***SetActive*** attribute of the weapon will become true. There are corresponding scripts to control the performance of different weapons, as shown in the

following example code Table 6.5 for the sword.

Table 6.5 Sword Code

```
private bool cooling = true;
[Header("攻击持续时间")]
public float time = 1f;
[Header("攻击间隔时间")]
public float AttackTime = 5f;
[Header("攻击力")]
public float Attackharm;
private float angle;
private Camera cam;
private Vector2 Direction;
private Vector3 mousePos;
private float attacktime;
void Start()
{
    cam = GameObject.FindGameObjectWithTag("MainCamera").GetComponent<Camera>();
    MovingT0 pc = Player.GetComponent<MovingT0>();
    attacktime = Mathf.Clamp(pc.attackTime, 0, 50f);
}
void Update()
{
    mousePos = cam.ScreenToWorldPoint(Mouse.current.position.ReadValue());
    Direction = (mousePos - transform.position).normalized;
    angle = Mathf.Atan2(Direction.y, Direction.x) * Mathf.Rad2Deg;
    gameObject.transform.GetChild(0).gameObject.transform.eulerAngles = new Vector3(0, 0, angle);
    if (Input.GetKeyDown(KeyCode.Mouse0)&&cooling == true)
    {
        cooling = false;
        Attack();
    }
}
private void attackstop()
{
    gameObject.transform.GetChild(0).gameObject.SetActive(false);
    Invoke("changeCooling", AttackTime * (1 - attacktime * 0.01f));
}

private void Attack()
{
```

```
    gameObject.transform.GetChild(0).gameObject.SetActive(true);
    Invoke("attackstop", time);
  }
  private void changeCooling()
  {
    cooling = true;
  }
```

Code for the definition of the various attributes of the weapon will be introduced in detail in the catalog at the end of the article. During the player's battle, the program obtains the mouse machine's position coordinates and Euler angles in real-time. The significance of doing this is that the attack mode of the sword used by the player changes according to the mouse position, and the weapon is mounted on the player's body. By constantly updating the player's position information relative to the screen and the mouse's position information and adjusting the angle of the player's launching weapon by a particular ratio to realize the purpose of the sword attacking the mouse way. When the weapon makes an attack, a timer that calculates the duration and cooldown time of the weapon attack starts working and is zeroed out for the next weapon attack.

The images in Figure 6.5 below shows the effects of the various weapons and equipment used by the player in battle. The player in the upper left corner used an axe to fight; the player in the upper right corner used a battle technique to try to defeat the enemy; the player in the lower left corner used a magical ranged attack; and the player in the lower left corner raised a shield to be able to fend off the enemy's attack. The game is completed with a particle system for all weapon effects.

Figure 6.5 Weapon Effects

6.6 Shops and Bags

The Backpack System features a sophisticated UI interface to help players manage their inventory. When players open their backpacks, their items can be automatically categorized and sorted according to various criteria. Players can easily access and use them during gameplay. The backpack system has a capacity limit, and players must think about which items to consume and discard during gameplay.

6.6.1 Data Save

This project uses ScriptableObject to create an in-game backpack and store system. ***ScriptableObject*** [21] is a data container to store information about items in the backpack. ***ScriptableObject*** is a class used in Unity to save data, which can save a large amount of data without attaching it to a

GameObject. This project uses ***ScriptableObject*** as a data container to store the information of the items in the backpack and realize the separation of data and logic, which makes it easy to manage the whole system and expand the new needed items.

ScriptableObject as immutable data corresponds mainly to configuration data, such as the name and description of the items in a game. Also, variable data is needed in the project to control the number of items the player owns. The project creates a script code based on ***ScriptableObject*** to store all the attribute information of the item. The new header file method ***CreateAssetMenu*** creates a new script file for each backpack item individually. Adding the ***CreateAssetMenu*** property to the ***ScriptableObject*** class can create the menu path and icon displayed when a new instance of ***ScriptableObject*** is created. This makes it easy to quickly create new instances of backpack items in the Unity editor. This project defines different attribute information under each type of item script. It also creates a particular ***ScriptableObject*** script for the backpack and gives it a list of properties. After the preparatory work, the project will create a regular standalone script to manage the backpack system. It ensures that the properties of the items are accurately associated with the list of backpacks.

6.6.2 Putting Items into the Bag

Implementing this functionality requires creating a separate script on each item to detect if the player has picked up the item. As shown in the code Table 6.6.2.1 below, the ***OnTriggerEnter2D*** method detects if contact has occurred between the player and the item, provided that the player and the item have triggers mounted on them. When the player-tagged game item is detected as touching the item, the ***AddnewItem()*** method is executed, which works with the backpack system's management script to transfer the item's attribute information to the backpack's corresponding grid. In this project, a separate ***Slot*** script is created for the backpack and mounted on the backpack's grid so that whenever a new item is added, the grid will be cloned to the corresponding position according to a specific

pattern. This part is implemented using ***Grid Layout***, a component that comes with Unity. As long as the corresponding positions are set in advance, the images added later will be arranged in the order and position set in advance.

Table 6.6.2.1 Picking Items

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

public class objectonworld : MonoBehaviour
{
  public Item thisItem;
  public List ObjectList;

  private void OnTriggerEnter2D(Collider2D other)
  {
    if (other.gameObject.CompareTag("Player"))
    {
      AddnewItem();
      Destroy(gameObject);
    }
  }

  public void AddnewItem()
  {
    if(!ObjectList.list.Contains(thisItem))
    {
      ObjectList.list.Add(thisItem);
    }
    else{
      thisItem.number += 1;
    }
    if(thisItem.style == ""){
      Manager.instance.RefreshItem();
    }

  }
}
```

As in Table 6.6.2.2, go back to the backpack management script and synchronize the previously mentioned item information to be added to the backpack's backend list. There is also an initialization of the items in the backpack's ***ScriptableObject*** script to ensure that all of the item information in the list appears in the backpack's UI grid interface. Before adding an item, it determines whether it is in the backpack; if not, it will be added directly. Otherwise, it will be necessary to change the number of items.

Table 6.6.2.2 Putting Items

```
public void AddItem(Item item)
{
  if (!Mybag.list.Contains(item))
  {
    Mybag.list.Add(item);
  }
  else
  {
    item.number += 1;
  }
  RefreshItem();

}
```

Once all the work is done, the items the player has acquired in the game can be added to the backpack without any problem. Items in the backpack do not disappear with scene updates or game closures but are permanently stored as game local files in the corresponding folders. The effect of the backpack system is shown in Figure 6.6.2. The image on the left shows the player about to pick up a weapon on the ground, and the image on the right indicates that the weapon appeared in the bag after the player picked it up. At the bottom is the UI page for the bag.

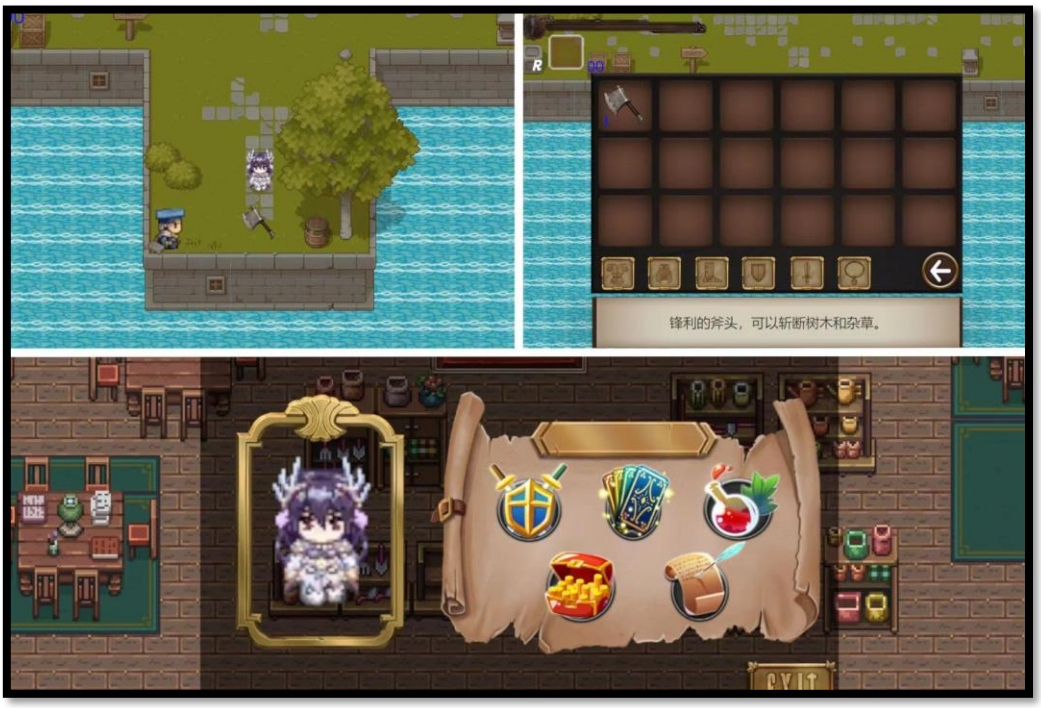

Figure 6.6.2 Bag System

6.6.3 Shop System

The store system is the center for players to purchase essential items and acquire defenses and consumables. Players will need to purchase powerful equipment to prepare themselves for the upcoming challenges in the game. The principle of the store is similar to the backpack system; both are made with ***ScriptableObject***. Unlike backpacks, items in the store are initially mounted in the store's list, which is stored locally in the player's files, as with backpacks. In addition, the process of the player purchasing an item from the store and the item going into the player's backpack is similar to the process of the player picking up an item on the ground and putting it into the backpack. In the store system, the slots that store information about the items are transferred from the store grid to the slots in the backpack. The Shop system is closely linked to the Store system to ensure that game items bought by the player will appear correctly in the corresponding player's backpack's sandwich. The realization of the store system is shown in the following Figure 6.6.3.

Figure 6.6.3 Shop UI

6.7 Multiplayer Online

The utilization of ***Netcode*** enables the seamless implementation of features in online multiplayer, thereby streamlining the process of constructing network frameworks. Within the realm of ***Netcode*** for ***GameObject***, three crucial roles exist: Host, Server, and Client. Upon the establishment of the Host, the server is concurrently instantiated. Subsequently, a client is automatically generated within the server. In contrast, the Server and Client roles solely create a server and a client, respectively, without additional functionalities. Clients serve the purpose of connecting to both servers and hosts. Within the ***NGO*** (Network ***GameObject***), pivotal components are present, among which is the ***NetworkManager***. The ***NetworkManager*** defines connection properties, including IP addresses and ports. Furthermore, the ***NetworkManager*** can be subdivided into various managers based on different criteria, establishing itself as the primary manager within the entire network system. In the context of online multiplayer, any GameObject necessitating transmission to the server must possess a network object identifier. Examples of such objects include players, enemies, and significant public

assets within the game [22].

6.7.1 Network Synchronization

Network synchronization is also an essential part of online multiplayer games. This project uses ***NetworkTransform*** to synchronize various GameObject transforms. The standard information in transform is position, rotation, and scale. ***NetworkAnimator*** synchronizes the state of players and enemies and varies in the game. Sever ***RPCs*** (Remote Procedure Call) are used in ***NGO*** to realize data synchronism. The server will invoke the Client ***RPCs*** [23], which will be executed later on the server as shown in Figure 6.7.1.1.

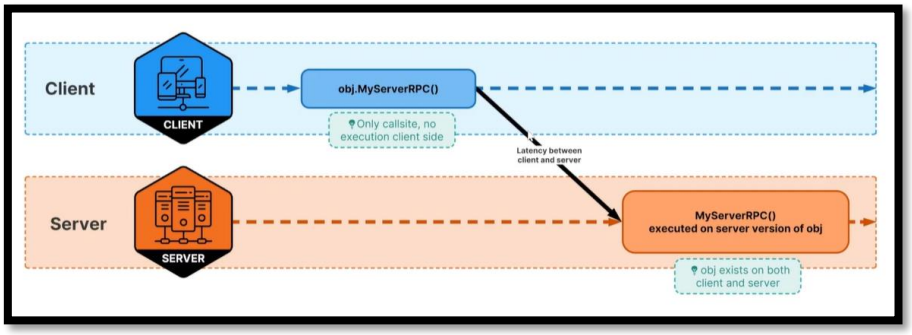

Figure 6.7.1.1 Client invokes a server RPC.

Inversely, the server invokes Client the ***RPCs*** but will be executed in every client. There is always one server in this project, but maybe more than one client. Sever ***RPCs*** and Client ***RPCs*** are both sent and executed in the Host. As shown in Figure 6.7.1.2.

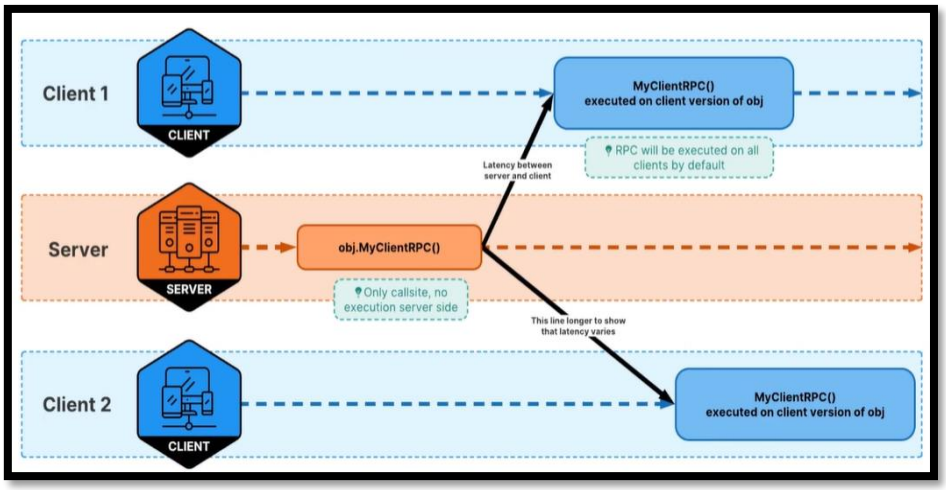

Figure 6.7.1.2 Server invoke a client RPC.

6.7.2 Data Transmission

The project employs the specified method to facilitate data transmission among distinct clients. Within the Unity ***Netcode*** framework, the Netcode for GameObject ***NGO*** module incorporates predefined serialization functionalities tailored for C# and Unity's fundamental data types. Notable among the supported C# types are bool, char, byte, ushort, short, int, uint, long, ulong, float, double, alongside numerous others like Unity-specific type. Enumerations are similarly subject to automatic serialization. The endeavor introduces an array of custom data structures that implement the ***INetworkSerializable*** serialization interface, thereby streamlining data transmission processes.

In the following Table 6.7.2, the ***PlayerInfo*** class implements the ***INetworkSerializable*** interface by defining the ***NetworkSerialize*** method to serialize the ***id*** and ***name*** fields. In a real-world scenario, it would include all the necessary data fields of custom data structure in the ***NetworkSerialize*** method to ensure they are properly transmitted over the network.

Table 6.7.2 Serialize the ***id*** and ***name***.

```
public class PlayerInfo : INetworkSerializable
{
    public int id;
    public string name;

    public void NetworkSerialize(NetworkSerializer serializer)
    {
        serializer.Serialize(ref id);
        serializer.Serialize(ref name);
    }

    public void NetworkDeserialize(NetworkSerializer serializer)
    {
        serializer.Deserialize(ref id);
        serializer.Deserialize(ref name);
    }
}
```

6.7.3 Establishing Network Connections

The subsequent phase pertains to establishing network connections. Following the development of the multiplayer game interface utilizing UI resources, the connection initiation between clients and the server ensues. The ***StartHost*** function is employed to instantiate client and server entities on the host computer. The acquisition of player-entered IP addresses from the game UI interface is facilitated through the ***GetComponent*** method. Subsequently, the IP addresses are converted to the ***UnityTransport*** format before invoking the ***StartClient*** function from the NetworkManager to spawn a new client, which is then linked to the server corresponding to the designated IP address of the host. It is imperative to note that invoking functions within the ***NetworkManager*** script necessitates adherence to the ***Singleton*** pattern. To enable additional players to join the room, the ***LoadScene*** function within the ***GameManager*** script, situated in the initial scene, is invoked to ensure networked scene loading. As shown in Figure 6.7.3. The scene name and load mode are specified

while declaring a static reference within the script, subsequently enabling the successful invocation of the ***LoadScene*** function.

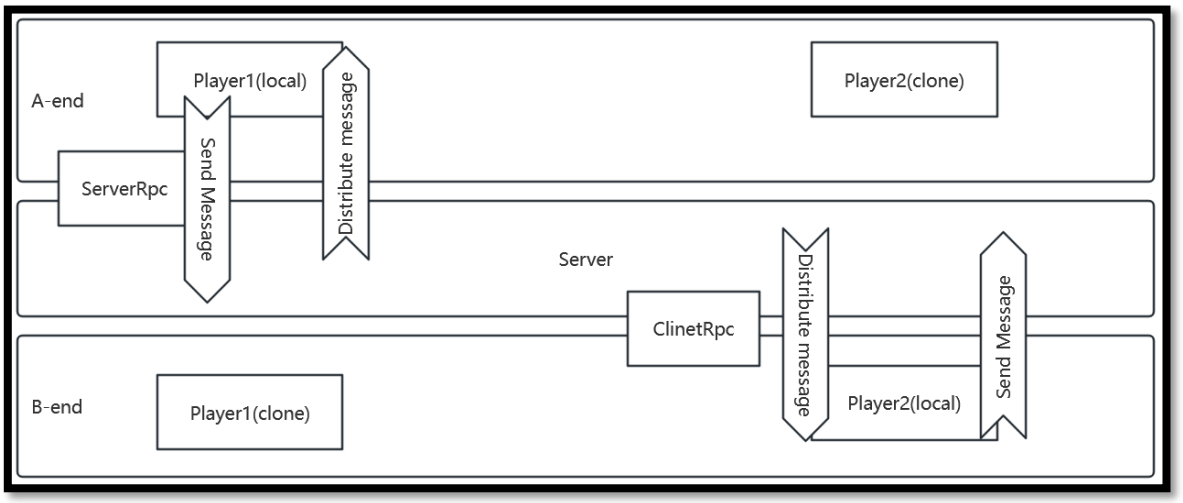

Figure 6.7.3.1 Player information is synchronized on the server.

Prior to integrating additional players, this project establishes a struct named ***PlayerInfo*** to ensure complete duplication of parameters. Subsequently, the struct is designed to inherit the ***INetworkSerializable*** interface to facilitate its transmission across the network [24]. Within the ***BufferSERIALIZER<T>*** function, the struct is configured as a custom serializable type, thereby finalizing the encapsulation of parameters within the struct for widespread utilization in subsequent code segments. To incorporate additional players, the project leverages the ***OnClientConnectedCallback*** interface from the ***NetworkManager*** component to implement event listening. Upon each new client's connection to the server, the associated function is triggered, allowing for the definition of parameters using the ***PlayerInfo*** struct. Similar to the process of adding a local player, details of new players are stored in a list. Iterating through this list, the project retrieves information pertaining to all players and employs a remote procedure call ***ClientRpc*** to notify all clients of the addition of new players [25]. During this stage, the project employs conditional statements to ascertain the existence of players on the current client. If a player does not exist, the ***AddPlayer*** function is executed. In figure 6.7.3.2, the variables for different clients are successfully synchronized. Notably, it is imperative to recognize that a Host encompasses both a

server and a client, necessitating an additional conditional statement to evaluate its server status using the ***isServer*** parameter. This ensures the seamless addition of all players to the game.

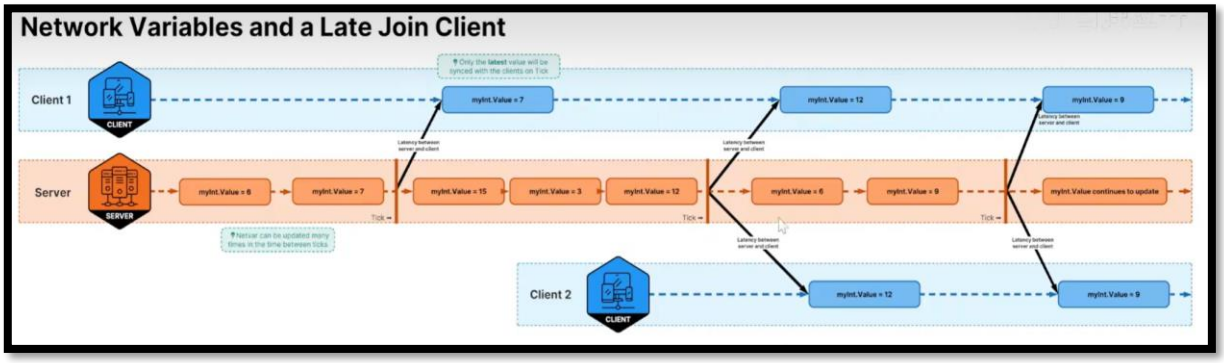

Figure 6.7.3.2 Network Variables and a Late Join Client

6.7.4 Updating Player Attributes for Network Transmission

In order to incorporate additional attributes requiring transmission over the network, such as readiness, gender, and name, new variables are to be added to the ***PlayerInfo*** structure. The project offers two distinct character options for players, each corresponding to different genders. To facilitate the transfer of the aforementioned attributes, three non-public functions, namely ***OnReadyToggle***, ***SwitchGender***, and ***OnEndEdit***, as shown in the following three code examples, have been defined to respectively manage the player's readiness status, gender, and name. Initially, ***PlayerListCell*** and PlayerInfo are initialized with corresponding values through the nested implementation of a dictionary utilizing their respective key-value pairs. These entities are denoted as ***_cellDictionary*** and ***_allPlayerInfos*** within the project, serving to govern, regulate, and alter player attributes prior to game commencement.

This code snippet in Table 6.7.4.1 is a method that handles changes in player readiness status. It updates the player information stored in a dictionary with the local client's ID and the new readiness status. If the code runs on the server, it is called the ***UpdateAllPlayerInfos ()*** method, which updates

information for all players. However, if the code runs on a client, it calls a method ***UpdateAllPlayerInfosServerRpc()*** with the local player's information as a parameter. This method synchronizes the player's information with the server and other clients. The project uses this code to manage player readiness changes, update player information locally, and synchronize this information with other clients or servers based on the current execution context.

Table 6.7.4.1 Readiness Status

```
private void OnReadyToggle(bool arg0)
{
    _cellDictionary[NetworkManager.LocalClientId].SetReady(arg0);
    UpdtePlayerInfo(NetworkManager.LocalClientId, arg0);

    if (IsServer)
    {
        UpdateAllPlayerInfos();
    }
    else
    {
        UpdateAllPlayerInfosServerRpc(_allPlayerInfos[NetworkManager.LocalClientId]);
    }

}
```

In the context of handling player readiness state changes, the code logic involves updating the local player information based on the changes made. This includes ensuring that any modifications to the player's readiness status are accurately reflected both locally and then synchronized with the server or other clients. The approach taken may vary depending on the environment, such as ensuring that the player data is consistently updated and synchronized in real-time to maintain the integrity and accuracy of the player information across all connected clients. Showing as the Table 6.7.4.2.

Table 6.7.4.2 Update Player's Status

```
private void OnEndEdit(string arg0)
{
if (string.IsNullOrEmpty(arg0))
{
return;
}
PlayerInfo playerInfo = _allPlayerInfos[NetworkManager.LocalClientId];
playerInfo.name = arg0;
_allPlayerInfos[NetworkManager.LocalClientId] = playerInfo;
_cellDictionary[NetworkManager.LocalClientId].UpdateInfo(playerInfo);
if (IsServer)
{
UpdateAllPlayerInfos();
}
else
{
UpdateAllPlayerInfosServerRpc(playerInfo);
}
}
```

The following code snippet enables the switching of gender by activating one gender option while deactivating the other, based on the integer parameter passed to the method.

$$transform.GetChild(gender).gameObject.SetActive(true);$$
$$transform.GetChild(1-gender).gameObject.SetActive(false);$$

6.7.5 Synchronizing Player Information Across Clients

The primary challenge lies in synchronizing players across multiple clients with the host's state upon player additions. While the project has thus far modified the ***_cellDictionary*** class, the synchronization of variables necessitates the synchronization of ***PlayerInfo***. This calls for the

encapsulation of an additional function. For instance, the project redefines and assigns values to ***PlayerInfo*** properties within the ***UpdatePlayerInfo*** class dictionary in player readiness. By establishing an equivalence between the information in ***PlayerInfo*** and the id in ***_allPlayerInfos*** and reassigning the id from ***_allPlayerInfos***, the objective of updating ***PlayerInfo*** is achieved. Subsequently, invoking the custom ***UpdatePlayerInfo*** within the ***OnReadyToggle*** function facilitates the updating of local information.

After local updates, the revised information must be transmitted. Analogous to player additions, conditional statements are implemented in the ***OnReadyToggle*** function to ascertain the server status. If the entity is a server, direct notification of the client can be issued; conversely, for clients, ***ServerRpc*** must be employed to apprise the server, enabling the invocation of the respective function. In the client scenario, the server lacks awareness of data alterations, necessitating the invocation of a custom ***Rpc*** function within the ***OnReadyToggle*** class to convey data from ***_allPlayerInfos*** to the ***UpdatePlayerInfo*** function. As previously delineated, ***_allPlayerInfos*** is a repository for diverse player information. Furthermore, ***PlayerInfo*** is passed as a parameter to this method, facilitating comprehensive data communication. Similarly, other player attributes, such as gender and name, can be synchronized across all clients utilizing a comparable procedure. For example, this project synchronizes the player's position information for each frame, as shown in Figure 6.7.5.1.

Figure 6.7.5.1 Player Position Synchronization

This project implements the player's chat window function based on ***NetCode***, as shown in Figure 6.7.5.2. In this window, the player is free to send messages to other players and receive messages from other players.

Figure 6.7.5.2 Chat Window

Chapter 7 Game Testing

7.1 Test Objectives and Methods

As part of the development process, the project tested various aspects of the game mechanics to ensure they function as expected. This includes identifying bugs, making necessary fixes, balancing game elements, and providing an enjoyable player experience. In the game testing phase, testing typically involves various goals such as functional testing, compatibility testing, performance testing, regression testing, usability testing, and localization testing. Functional testing ensures the game functions properly, while compatibility testing ensures the game runs well on different platforms and devices. Performance testing evaluates the game's performance under different conditions, such as frame rates or loading times. Regression testing ensures that modified code does not introduce new issues and maintains game stability. In contrast, usability testing optimizes player interactions and usability to create an enjoyable experience [26].

The project focuses on various aspects of game creation, from game mechanics and user interface, balance, sound effects, and narrative flow to the smooth operation of various functions (such as character movement, combat systems, item acquisition, and quest completion) to ensure their proper functioning. The critical features tested in this project are the save system, storage system, and skill system because they interact with each other and significantly impact the game. Additionally, any map designs, quest settings, and character configurations that constitute the gameplay and narrative requirements of the game were carefully examined to ensure no bugs or unreasonable values that could affect player experience appear in the game.

In addition to the game's content, this project pays particular attention to its performance, including frame rates, loading times, and memory usage, to ensure optimal gameplay under all circumstances.

7.2 The Function Test Case

Testing was completed of RPG game's essential functions, such as Collision Detection, Damage Judgment, Scene Change, and Data Archiving. Collision Detection was checked for its ability to accurately detect object interactions that led to successful combat scenarios; Damage Judgment focused on precise calculations of damage dealt and received for balanced gameplay. Scene Change testing involved smooth transitions between different game environments, enhancing player immersion. Data Archiving testing verified the secure storage and retrieval of player progress, maintaining continuity in the gaming experience.

Table 7.2.1Game Functions Test

Function Description: game functions Purpose of function test: Ensure that the game functions correctly Prerequisite: Game function scripts do not report errors and the game runs normally		
Functions	Output or Corresponding	The Actual Situation
Collision Detection	The collision detection function should accurately detect when two objects in the game world come into contact, triggering appropriate responses or events.	During testing, collision detection successfully identified when the player character collided with obstacles or enemies, leading to actions such as taking damage or triggering animations.
Damage Judgment	The damage judgment function should correctly calculate and apply damage to characters or objects based on various factors such as weapon strength, enemy defenses,	Testing revealed that the damage judgment system accurately calculated damage dealt by player attacks and enemy

	and player stats.	abilities, taking into account relevant game mechanics and variables.
Scene Change	The scene change function should smoothly transition the player between different game environments or levels without interruptions or errors.	Through testing, the scene change feature successfully transitioned the player character between distinct game scenes, maintaining visual continuity and ensuring a seamless gameplay experience.
Data Archiving	The data archiving function should reliably save and load player progress, including game settings, character stats, and completed objectives.	Testing confirmed that the data archiving system effectively stored and retrieved player data, allowing for the preservation of game progress and customization choices across gaming sessions.

Chapter 8 CONCLUSION AND EXPECTATION

8.1 Conclusion

This project divides the game into the big game world and many mini-games. The big world will contain three big maps; players can carry out all kinds of interactions in the big world (the mission system, backpack system, character control system, variety of enemies, and the corresponding UI systems). Players will be the characters in the main world and can finish the tasks, upgrade, fight with monsters, and get equipment and props in the main world. Players can also find a way to unlock other mini-games by acquiring unique props for main tasks in the world. They will obtain more advanced equipment and skills by completing various mini-games successfully.

Mini-games are the most significant characteristic of this project. Different mini-games can be unlocked in different ways, including defeating enemies, completing hidden quests and finding key props. In the big world, the mini-games are of different types, but the only thing we can be sure of is that the style of the mini-games will have a manageable gap between the big world and the mini-games. The mini-game data will be bound with the player's personal information in the big world. If the player obtains props in the mini-game, it will affect the character's panel data or the critical prop used in the following game plot.

In this project development, many tools will be used besides Unity. Using the case method, for example, the RPG game "Eastward," which has art material with a unified style to build terrain and levels. This project will use Photoshop to optimize the game material in the 2D part. They are building the necessary simple 3D model by 3Dmax to add the essential material content to the game. Coordinate the material style through Blender rendering or material spheres in Unity. In the logic of interaction, scripting is usually done with sharp language. Also, the Unity plugin can match the script and create interesting systems to enrich the game's playability.

This project adopts the research method of qualitative analysis to get the feelings and experiences of different game developers with their development process. For example, they use the Doozy UI plugin to realize some cool effects quickly. The Doozy UI plugin can also conveniently manage UI systems and event delivery mechanisms. Compared with using Unity's canvas tool to create UI directly, managing UI in a group can be more convenient, including mutual exclusion and switching multiple UIs.

By comparing skeletal and sequence frame animation, skeletal animation is primarily used in 3D games with more flexible functions and complexity. In 2D games with sufficient art resources, sequence frame animation can be a faster and more convenient way to solve the game's needs for character animation. This project will repeatedly experiment with the implemented features to find out the problems and solve the bugs to ensure the integrity of the game.

This paper uses various research methods, combining unity tools and scripts to flexibly develop various game functions and complete an independent game with a novel style according to actual conditions.

8.2 Expectation

Moving forward, it is crucial to continue refining these essential functions to enhance the overall gaming experience. Improving Collision Detection can optimize combat mechanics, making battles more dynamic and engaging. Enhancing Damage Judgment calculations can add depth to gameplay strategies, offering players more diverse options in combat scenarios. Streamlining Scene Changes will provide a seamless gaming experience, minimizing disruptions and maximizing player immersion. Strengthening Data Archiving processes will ensure reliable storage and retrieval of player data, safeguarding progress and achievements. By prioritizing these aspects in future

development cycles, the RPG game can continue to evolve and provide players with a captivating and immersive gaming experience.

REFERENCES

[1] Concept Art Empire, *What is Unity 3D & What is it Used For?*, 2024. [Online]. Available: https://conceptartempire.com/what-is-unity/ [Accessed Jul. 7, 2023].

[2] GreeksforGreeks, *Introduction to C#*, 2023. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-c-sharp/ [Accessed Jul. 8, 2023].

[3] Fireship, *VS Code in 100 Seconds*, 100 Seconds of Code, Nov.26, 2021. [Video file]. Available: https://www.youtube.com/watch?v=KMxo3T_MTvY [Accessed Jul. 10, 2023].

[4] Unity Technologies, *About Netcode for GameObjects*, 2024. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/about/ [Accessed Jul. 11, 2023].

[5] T. Norton, *Learning C# by Developing Games with Unity 3D Beginner's Guide*. Packt Publishing Ltd, 2013, ch. 7, pp. 112-114. [ebook]. Available: https://www.computo-visual.org/Videojuegos/Libros/Learning%20C%23%20by%20Developing%20Games%20with%20Unity%203D.pdf [Accessed July. 18, 2023].

[6] Y. Wu, *Unity you xi kai fa ji su xiang jie yu dian xing an li* [Unity game development technology with the example and introduction]. Posts and Telecommunications Press, 2018, ch. 19, pp. 689-692.

[7] K. Howland, J. Good, and J. Robertson, "Script Cards: A Visual Programming Language for Games Authoring by Young People," *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pp. 181-186, 2006. [Online]. doi: 10.1109/vlhcc.2006.42 [Accessed July. 20, 2023].

[8] J. Zagal, S. Deterding, *Role-Playing Game Studies: Transmedia Foundations*. Routledge, 2018. p. 9.

[9] G. Gygax, D. Arneson (1974). *Dungeons & Dragons*, Total Shareholder return. 1974. Available: https://dnd.wizards.com/ [Accessed July. 22, 2023]

[10] Interplay Inc (2009). *Fallout*, Bethesda Softworks. 2009. Available: https://fallout.bethesda.net/ [Accessed July. 24, 2023]

[11] M. Morhaime, MarkCohen (2004). *World of Warcraft*, Blizzard Entertainment. 2004. Available: https://worldofwarcraft.blizzard.com/zh-tw/ [Accessed July. 25, 2023]

[12] Kong Studios (2021). *Guardian Tales*, Kakao Games. 2021. Available: https://guardiantales.com/ [Accessed July. 27, 2023]

[13] W. Pellen (2017). *Hollow Knight*, Team Cherry. 2017. Available: https://www.hollowknight.com/ [Accessed July. 29, 2023]

[14] G. Orsmond, E. Cohn, *The Distinctive Features of a Feasibility Study*. OTJR: Occupation, Participation and Health, 2015. [ebook]. Available: https://doi.org/10.1177/1539449215578649 [Accessed July. 30, 2023]

[15] Alexander Zotov, *Unity 2D Tutorial About How To Make Temporary Invulnerability Feature For Game Character*, How To Create Simple 2D Platformer In Unity, Mar.30, 2018. [Video file]. Available: https://www.youtube.com/watch?v=S61J3kDQ5Mk [Accessed Jan. 1, 2024].

[16] Harrynesbitt, "2d day/night cycle and weather system à la 'Alto's Adventure'", Jan. 8, 2016. [Blog]. Available: https://forum.unity.com/threads/2d-day-night-cycle-and-weather-system-a-la-altos-adventure.377959/ [Accessed Jan. 2, 2024]

[17] Stack Overflow, *Unity dialogue system for multiple dialogues in the scene*, 2024. [Online]. Available: https://stackoverflow.com/questions/77964808/unity-dialogue-system-for-multiple-dialogues-in-the-scene [Accessed Jan. 3, 2024]

[18] F. Hugo, *Looking at In-Game Currencies*. GameSpark, 2014. [ebook]. Available: https://www.gamesbrief.com/2014/12/looking-at-in-game-currencies/ [Accessed Jan. 4, 2024]

[19] Tamara Makes Games, *Grid Building System Tutorial in Unity | Making HayDay*, Making HayDay - Tutorial series, Nov.7, 2021. [Video file]. Available: https://www.youtube.com/watch?v=3dmyKZw6i5k&list=PL0FALXGqi7wIbQRN1u-D-gu0CjIU4xVmj&index=3 [Accessed Jan. 5, 2024].

[20] Tamara Makes Games, *Farming Tutorial Unity | Making HayDay - Planting & Harvesting*, Making HayDay - Tutorial series, Apr.16, 2022. [Video file]. Available: https://www.youtube.com/watch?v=Xkocz0BMPnU&list=PL0FALXGqi7wIbQRN1u-D-gu0CjIU4xVmj&index=7 [Accessed Jan. 6, 2024].

[21] Marco Mignano, "Mastering Scriptable Objects in Unity: A Complete Guide, 2019. [Online]. Available: https://marcomignano.com/posts/mastering-scriptable-objects-in-unity-a-complete-guide [Accessed Jan. 8, 2024].

[22] P. Lorenz, "Network Transform – A Tool Based Approach," *International Journal of Future Computer and Communication*, vol. 2, no. 2, pp. 96-100, Apr. 2013. [Online]. Available:

https://www.ijfcc.org/index.php?m=content&c=index&a=show&catid=38&id=361 [Accessed Jan. 9, 2024].

[23] Unity, *Sending events with RPCs*, 2024. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/messaging-system/#docusaurus_skipToContent_fallback [Accessed Jan. 10, 2024]

[24] Unity, *INetworkSerializable - Unity Multiplayer Networking*, 2023. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/1.4.0/advanced-topics/serialization/inetworkserializable/ [Accessed Jan. 11, 2024]

[25] WinCC OA, *Synchronization Client-Server*, 2024. [Online]. Available: https://www.winccoa.com/documentation/WinCCOA/3.18/en_US/Synchronisation/Synchronisation-01.html [Accessed Jan. 12, 2024]

[26] KEVURU GAMES, *Main Stages of Video Game Testing: From Concept to Perfection*, 2023. [Online]. Available: https://kevurugames.com/blog/main-stages-of-video-game-testing-from-concept-to-perfection/ [Accessed Jan. 15, 2024].

APPENDIX

Players can use various weapons to fight with the enemy in the game. Different types of weapons have different attack templates and modes, and different weapons in the same category have unique attributes and corresponding attack effects. Therefore, players need to choose the right weapon when facing different enemies. The table lists the attribute panels of different weapons in detail, and in the category of melee weapons, the game has three types of weapons for players to choose from swords, axes, and hammers. Their attributes represent similar meanings.

Attack distance is the furthest distance the player can damage an enemy, but this does not mean any position within that distance will have an attack judgment. Whether or not damage can be dealt to an enemy is determined by the shape of the trigger used to determine damage generated when the weapon is swung in the game. Therefore, the player needs to keep checking the combat feel of different weapons, as different weapons judge damage differently. A weapon's attack power is the initial value of the weapon, but this does not mean that it can directly damage the enemy with this value. The game has its damage calculation formula, and all melee weapons can only deal average physical damage. The in-game damage formula is as follows.

$$H = P * 40\% + S * 60\% - 50\% * def$$

Where H is the final damage dealt by the player to the enemy, P represents the player's physical attack, S represents the weapon's attack power, and def represents the enemy's physical defense. In addition, some weapons can also cause special effects, which will be explained to the player in the game. If a weapon's attack power has an additional multiplier by a completely new value, the weapon can deal multiple damage on each attack. The attack interval of a melee weapon represents that the player can only use that weapon a maximum of once within the stipulated time. The design of the relevant values is shown in the following tables.

Appendix 1 Close-in Weapons

Appendix 1.1 Sword

Table Appendix 1.1 Sword Information

武器	攻击力	攻击间隔	攻击范围
铁剑	5	1.5	8
毒剑	4	1.5	8
刺刀	3.5	0.8	6
守卫者的长剑	7.5	1.9	9
骑士大剑	8.5	2.3	9
雷光刃	6	1.65	7
二叉戟刃	4.5*2	1.8	7.5
木剑	1	1.5	8

Appendix 1.2 Axe

Table Appendix 1.2 Axe Information

武器	攻击力	攻击间隔	攻击范围
粗制石斧	6	1.5	7.2
双头斧	8	1.85	6.5 + 3
细柄直斧	5.3	1.32	7
黑骑士大斧	17	2	7

Appendix 1.3 Hammer

Table Appendix 1.3 Hammer Information

武器	攻击力	攻击间隔	攻击范围
粗制铜锤	5.5	2	9
工程锤	6.1	1.8	8
白银之锤	14	2.4	9
神圣回声	20	2.76	8.5

Appendix 2 Armor

In the game, players can use defense equipment to resist the enemy's attack, and the shield is the primary defense in the game. Different shields have the same defense templates and mechanics, but a large panel gap exists between shields. Therefore, players must choose the proper defense when facing different enemies. The table lists the attribute panels of different defenses in detail. The shield's immunity rate represents the damage a player receives when using a shield to successfully defend against an enemy attack, minus the damage the player should have received multiplied by the shield's immunity efficiency. The game has a set of physical formulas for calculating the damage the player takes. The formula for calculating the damage taken by a player attacked by an enemy without a shield raised is as follows.

$$H1 = Y - (X / 100) * 70\% * Y$$

Where H1 is the value of the damage received by the player, Y represents the enemy's attack power, and Y contains both the enemy's physical and magic attacks. x represents the player's defense power. Like the Y, it contains the player's physical and magic defenses. If the player blocks a single enemy attack with his shield, the damage taken is as follows.

$$H2 = H1 * A\%$$

A represents the shield's immunity rate, and H2 represents the player's ability to block a single enemy attack with the shield. The design of the relevant values is shown in the following tables.

Table Appendix 2 Shield Information

盾牌	免伤率	抵挡次数	损坏后冷却时间
牢固的木板	50	3	15s
小圆盾	30	4	9s
亚诺尔隆德士兵盾	68	3	23s
魔法师护身盾牌	37	8	31s
红狮子战士盾牌	75	2	18s
国王近卫军大盾	84	2	26s

Appendix 3 Accessories

The game accessories will also significantly impact the battle; there are three different kinds of accessories in the game. Players wear necklaces to access appropriate magic, badges to access appropriate battle skills, and rings to gain specific attribute bonuses. The three types of trinkets can be worn simultaneously, but each type can only be worn by a maximum of one.

The magic used by the player wearing the necklace can cause magic damage to the enemy, at which time the corresponding vector is deducted based on the enemy's magic defense. Similarly, when the player wears a badge and hits an enemy with a battle technique, the enemy deducts the corresponding amount of blood based on their physical defense. The in-game formula for this is as follows.

$$Necklace: H = P * 35\% + S * 70\% - 30\% * def2$$

$$Badge: H = P * 35\% + S * 70\% - 30\% * def1$$

Where H is the value of damage the enemy takes, and S represents the player's attack power. def1 and def2 represent the enemy's physical and magical defenses, respectively. The design of the relevant values is shown in the following tables.

Appendix 3.1 Necklace

Table Appendix 1.3 Necklace Information

项链	攻击力	技能冷却	持续时间
草纹项链	20	5	2
熔岩项链	24	5.4	1.8
飓风项链	15	4.6	3

Appendix 3.2 Ring

Table Appendix 3.2 Ring Information

戒指	效果
幽灵戒指	增加移动速度和冷却
大地戒指	提高物理魔法防御力
贵族戒指	增加物理魔法攻击力

Appendix 3.3 Badge

Table Appendix 3.3 Badge Information

徽章	攻击力	技能冷却	持续时间
红色骑兵徽章	25	4	1
蓝色武士徽章	22	5	1
堕落魔女徽章	36	8	1

Appendix 4 Drug

Medicines in the game can also significantly impact combat, and there are five different types of medicines in the game. Players can wear them in the shortcut bar and use the shortcut keys to take them. There is a limit to the number of medicines that can be taken, and different medicines have different effects and can increase one of the player's attributes quickly. Players can take unlimited medicines, and the effects of different medicines can be stacked, but the same medicines cannot.

Table Appendix 4 Shield Information

盾牌	效果
攻击药	提高 15%攻击力 30s
回血药	回复最大生命值的 30%
防御药	提高 15%防御力 30s
加速药	提高移动速度 20% 1 分钟
净化药	清除所有负面状态