HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

GRADUATION THESIS

Funny Zoo - A Casual Game Built With Unity

HOANG MANH HA
ha.hm184250@sis.hust.edu.vn

Major: Information Technology

Supervisor: D.C.Sc. Nguyen Thanh Hung _____

Signature

School: Information and Communication Technology

HA NOI, 06/2022

ACKNOWLEDGMENTS

Words cannot express my gratitude to D.C.Sc. Nguyen Thanh Hung for his detailed guidance, immeasurable patience and feedback. I also could not have undertaken this journey without my defense committee, who generously provided knowledge and expertise.

I am also very grateful to all the lecturers at Hanoi University of Science and Technology, especially the lecturers in the Information and Communications Technology institute because of all the valuable and interesting knowledge and skills as well as hands-on experiences in my five academic years. I found these knowledge and skills so useful and practical in my career and my life as well.

I am also grateful to my classmates for their editing help, late-night feedback sessions, and moral support.

ABSTRACT

Mobile gaming has become a vital part of modern life. People around the world spends hours of their daily life playing mobile games, from a student who plays co-operative games with friends in the break, to a worker who is traveling to work on the train and want something to kill time. As the requirement for gaming increase, more and more games are being published everyday. Though some of the games are well refined with high quality and content, a large number of other games being published on app store are just cloned from the top games of the market, with identical gameplay, poor design, less refinement and full of ads. Those games are a huge annoyance to the players and making them to turn away with mobile game products. In order to gain back the trust of the mobile game player, as well as bringing them relaxing and enjoyable moment, this graduation thesis will aim to make a mobile casual game with carefully designed theme, high quality content and remarkable game play that will satisfy even the strictest players. The product will be developed using a fully featured game engine and a proper workflow so as to maximize the productivity as well as minimize the development time, but still ensure the quality of the game itself. The final products of this graduation thesis will be a complete game with easy-to-catch-up game mechanic, detailed game design and with average game session in order to be suitable for a wide range of mobile game players.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SPECIAL TERMS

Notation	Description
1. AAA games	An informal classification used to categorize games produced and distributed by a mid-sized or major publisher, which typically have higher development and marketing budgets than other tiers of games.
2. boilerplate	Sections of code that are repeated in multiple places with little to no variation.
3. netcode	Netcode is a blanket term most commonly used by gamers/developers relating to networking in online games.
4. rigidbody	A Unity component that put object's motion under the control of physics engine.
5. shader	In computer graphics, a shader is a computer program that calculates the appropriate levels of light, darkness, and color during the graphics rendering.
6. texture	A texture, in general, is some sort of variation from pixel to pixel within a single primitive that can be applied to a surface.

LIST OF ABBREVIATIONS

Notation	Description
7. API	Application programming interface
8. FPS	First person shooter
9. GPU	Graphical processing unit
10. JSON	JavaScript object notation
11. MMORPG	Massively multiplayer online role-playing game
12. MOBA	Multiplayer online battle arena
13. MVC	Model view controller
14. MVP	Model view presenter
15. P2P	Peer to peer
16. RPC	Remote procedure call
17. RTT	Round trip time
18. SDK	Software development kit
19. UDP	User datagram protocol

CHAPTER 1. INTRODUCTION

This chapter will discuss the thesis' overview topic as well as the factors that influenced the decision to focus this graduation study on these issues. Additionally, this section chapter the topic's goal and solution as well as an introduction to the report's structure.

1.1 Motivation

In the modern world, with the rapid evolution of information technology, mobile phone has become an essential electronic equipment in our daily life. The usage of mobile device ranges widely form phone calls, texting, browsing, entertaining, and in recent years, gaming. Gaming on mobile phones has become one of the most viral trending among the youth.

Gaming on mobile phone is entertaining, easy to start, and available almost anywhere when user has a mobile device with them, therefore, the demand for mobile game has grown rapidly. Many studios around the world are publishing games which target different range of users and platforms. With the rise in the sale of tablets and mobiles, game app download has increased immensely, and with the introduction of cloud gaming and real money games, the mobile gaming industry has taken a take-off, and the future of mobile gaming looks very bright indeed. The mobile gaming market has grown so much that it is now ahead of computer gaming. The success of a game depends on many factor, which can include, but not restricted to game theme, market trending, target users and game engine. Behind the development phase of every game is a game engine, a tool which simplified many aspects in game development and support developer code and manage resource more easily and efficiently.

From the growing trend mentioned, in this project, we will use Unity to create a hyper casual mobile game - Funny Zoo.

1.2 Objectives and scope

Nowadays, in the video game industry, there are many different groups of video games which target different types of gamers and platforms. Typically, they're categorized by their size, budget and development time. Some common game categories can be listed as (i) AAA games, (ii) mid-core games and (iii) casual games.

AAA games (pronounced "triple A games") are video games most distinguished by their massive development and marketing budgets. They are meant to be a game development company's best work and provide a high quality video game

experience-comparable to a summer blockbuster. Aside from budget, there are a number of other common features that these games posses. Though there is no limit on genre, unifying qualities among these games are typically photo-realistic graphics, game worlds of massive scale, cross-platform releases, and violent content. There are certain exemptions for each of these qualities, but on the whole, these trends hold. However; money is by far the most important factor in determining whether a game is AAA or not, and equally important to these games' massive budgets is their massive expected revenue and profit.

Mid-core games are more complex than hyper-casual and casual games, requiring players to make time to play, rather than playing opportunistically or sporadically. Mid-core games require skill and strategy to progress and as a result, require players to be more invested than a typical casual game. Mid-core games often feature multi-player experiences, side quests, and resource management, unlike casual games where your goal is to solve a puzzle or complete a repetitive action. Mid-core games often distill an AAA game down to simpler elements in order to have a broader appeal. Think of a hardcore racing game on mobile distilled down to swipe gestures, which are far less punishing and easier to learn than motion controls.

Casual games are video games targeted at a mass market audience, as opposed to a hardcore game, which is targeted at hobbyist gamers. Casual games may exhibit any type of gameplay and genre. They generally involve simpler rules, shorter sessions, and require less learned skill. They don't expect familiarity with a standard set of mechanics, controls, and tropes. Countless casual games have been developed and published, alongside hardcore games, across the history of video games. Most casual games have fun, simple gameplay that is easy to understand. They have simple user interface, operated with a mobile phone tap-and-swipe interface or a one-button mouse interface. Besides, casual game tends to have short sessions, so a game can be played during work breaks, while on public transportation, or while waiting in a queue anywhere. Casual games generally cost less than hardcore games, as part of their strategy to acquire as many players as possible. Any game monetization method can be used, from retail distribution to free-to-play to ad-supported. Also, they are easier to develop and publish.

From the problem mentioned above, along with the researches carried out, we decided to create a casual game. The main reasons are that this type of game has short development times and suitable for a solo developer. It also has a lower standard than other game gernes, so the game assets can be collected from the internet and free asset store. In this project, we also want to do research on and apply new technologies and methods on game development process to produce a mobile game

with high quality, attractive to users and on trend with the market. The game will also be designed with unique gameplay and proper monetization.

1.3 Tentative solution

To achieve the objectives mentioned above, in this project, several researches must be carried out. The first aspect to create a complete game is game design. In game design, game creators must study the trending of the market, as well as game genres and types in order to decided the main theme and play style of the game being created. Additional information about development time, budget and viability of the project should also be taken into account during this phase. From the information mentioned in section 1.2, the target game genre for this thesis project will be hyper casual, which has a short development time, small development cost and can easily be published to google store, but still be competent to other game genres on google store. Next, a game engine must be chosen for the development process. The game engine chosen should be able to balance between the development aspect and the game design aspect, in other words, the game engines should have enough features and can be utilized to speed up the development process and guarantee the quality and content of the game at the same time. Unity Game Engine is by far the most suitable game engine for this project due to its excellent mobile support for game development phase. Detailed explanation for choices of the game engine will be discussed in section 3.1. The final products will be a casual game with average content, interactive gameplay and high engagement time, which mainly target young audience, and expected to have five millions downloads on google store.

1.4 Thesis organization

The rest of this graduation thesis will be structured as follows:

In chapter 2, requirements of the software are first described through use case diagrams. Brief description as well as specification of some important use cases of the system are then presented. These include activities that a player can do in online as well as offline game sessions. Finally, chapter 2 lists some non-functional requirements of the system.

Chapter 3 will mainly focus on the technology stack used for application development. The chapter begins by explaining the need for a game engine. With the objectives of building a game targeting mobile platforms, some of the popular game engines are put on comparison for their advantages and disadvantages. We then give reasons for why Unity is the ultimate choice for this project. The process of choosing a suitable multiplayer framework is showed in a similar fashion.

After a technology stack is determined, chapter 4 dives into the details of appli-

cation development and deployment. Design elements are identified and analyzed to implement the system's functional requirements. These elements include architecture design, package and class design, interface design, application testing and deployment.

Chapter 5 exhibits solutions to a few problems encountered in game development such as object synchronization and reactive UI architecture.

Chapter 6 summarizes achieved results and presents orientation for future development. Remaining limitations of the application are also briefly discussed.

CHAPTER 2. REQUIREMENT SURVEY AND ANALYSIS

This chapter includes information about current top trending game genres on the market, their strength and weakness and general users' opinion toward those games. After that, this thesis will focus on the functional and non-functional requirements for the Funny Zoo game.

2.1 Status survey

For mobile games, Google Play and App Store are the two most popular online stores for mobile app at the moment. With millions of games on these store, one can expect to come across a large numbers of game categories with different themes and play styles. For dedicated gamers who want to show their high skill when competing other players in a realtime battle, MMORPG and MOBA games are at the highest preferences. These games often require strategy, skill and dedication and in return, they provide the player with satisfaction and the feeling of achievement.

Though being high-rated by both players and reviewer, MMORPG and MOBA game are not of the top games ranking due to their high dedication requirements and long game sessions, which are not quite suitable in a modern busy society. Also, the performance of the games, though being highly optimized, is not quite acceptable for the majority of the mobile devices on the market today.

Hyper casual games, in contrast, have a short game session and easy to catch up game play. Their fast and highly interactive style fit almost all sort of player. Kids are attracted by the color full themes, juicy effects and haptic the hyper casual game bring back. Adults can try out some of those interactive and easy-to-understand gameplays when they are on the train or at break time to kill time. Teenagers are also into hyper casual game when they expect something fast and relaxing to entertain during their free time, or they just want to try something new after experiencing other categories of games. From all the reasons above, hyper casual games have become the top trending game category on the online mobile app store. Player can come across a variety of top trending hyper casual genres on store such as runner, shooter, 2d puzzle, simulation and tycoon.

Though being the top trend on the market, hyper casual games also have their own problems. As the competition rise, many hyper casual games are mass produced without detailed designed and refinement. There are also a great amount of games cloned from top trending game on the market, which distract the users and lower the quality of this game genres. And finally, most of the low-rating comment

of hyper casual games are about the disturbing ads during the game session, which significantly affects players' experience and felling about the game.

In order to compete in such a competitive market, the final product this project must be carefully designed in various aspects: concept, game play, main theme and most important, player experience. Funny Zoo, as the name suggest, is a tycoon-like game where player becomes the owner of a small zoo. The player is in charge of feeding the animals, cleaning the cages, collect the fee and some other interest actions. The game is designed with single touch controller, which is easy to catch up even for no-experience player. The main concept of the game is a toony world with cute, low poly characters and bright, colour full environments, which brings player relaxation and chilling time while keeping them busy doing the zoo stuff. The game is also designed to extend the normal engagement time by sub-dividing the game progress into multiple phase, where player can make progression and discover new hidden contents of the game, thus keeping them staying longer. Funny Zoo is also designed with proper monetizing to aid player in progressing the game, which in return, provide some profit to the developers. Structured coding and optimization also need to be applied into the game development in order to ensure that the game can be extended with more feature in the future, while can still run smoothly on even low-end devices.

2.2 Overall description

All use case of the system are generally described by an overall use case diagram as shown in figure 2.1 There are 1 actor in the system that is the player.

2.2.1 Use case diagram

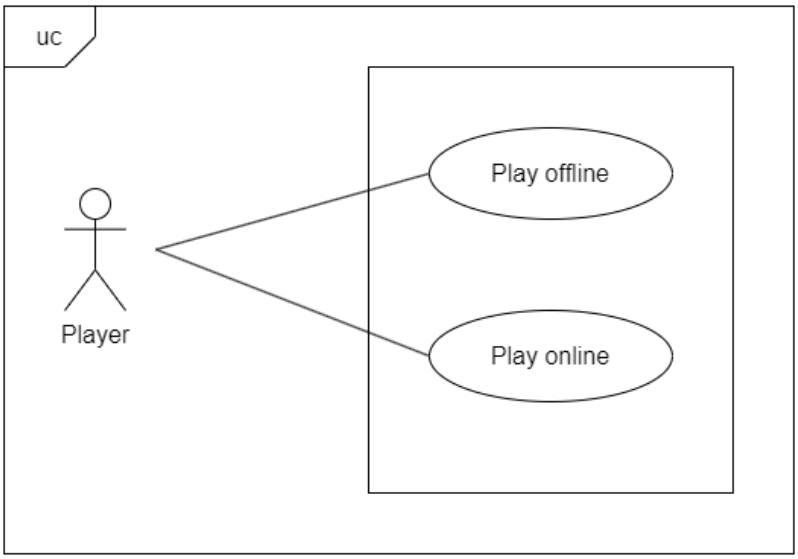


Figure 2.1: Overall use case diagram

Player is an actor that directly plays the game. The people who play the game first need to choose a playing mode, offline or online mode.

2.2.2 Decomposition of use case "Play offline"

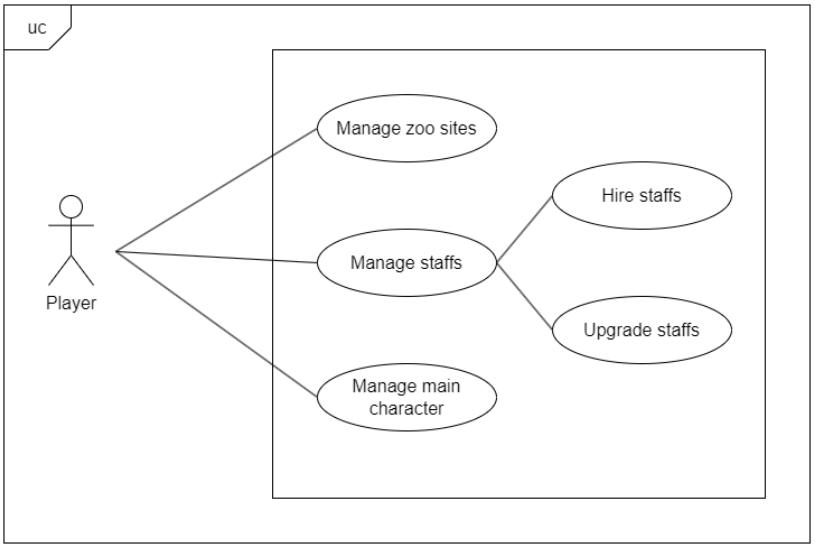


Figure 2.2: Play offline use case diagram

Figure 2.2 describes all functions that a player can do in an offline game session. This is including, but not limited to manage the staffs, manage the zoo sites and control the main character of the game.

2.2.3 Decomposition of use case "Play online"

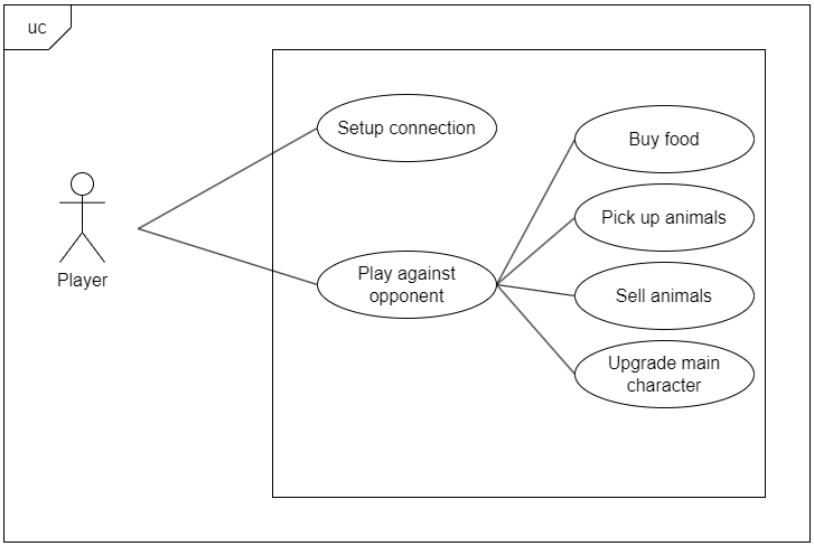


Figure 2.3: Play online use case diagram

Figure 2.3 describes all functions that a player can do in an online game session. Player first needs to set up the connection for the game session and can then later plays against an opponent through the internet.

2.3 Functional description

2.3.1 Specification for use case "Manage zoo sites"

Use case code	**UC001**
Use case name	**Manage zoo sites**
Actor	Player
Description	Player do actions to manage zoo sites
Precondition	Player chooses offline mode
Post condition	No
Activation	No
Main flow of event (success)	1. Player goes to the food containers or food production sites to get the food items for zoo animals. 2. Player feeds the zoo animals. 3. Player cleans up the zoo sites by picking up the poos of the zoo animals and throw them into the trash bin.
Alternative flow of event	No

Table 2.1: Specification for use case "Manage zoo sites"

2.3.2 Specification for use case "Hire staffs"

Use case code	**UC002**
Use case name	**Hire staffs**
Actor	Player
Description	Player do actions to hire more staffs for the zoo
Precondition	Player chooses offline mode
Post condition	No
Activation	No
Main flow of event (success)	1. Player enters staff manage region. 2. The system display the pop up for managing the staffs. 3. Player choose hiring tab. 4. Player choose staff to hire. 5. Player choose to spend cash or watch ads to hire staffs for the zoo. 6. Purchase success, the newly hired staff appears and starts working.
Alternative flow of event	4a. No available staffs to hire. The system notifies that max number of staff has been reached. 5a. Not enough money to spend. The hire-with-money button is grayed out, indicating that there is not enough money to hire the staff.

Table 2.2: Specification for use case "Hire staffs"

2.3.3 Specification for use case "Upgrade staffs"

Use case code	**UC003**
Use case name	**Upgrade staffs**
Actor	Player
Description	Player do actions to upgrade base stats of the staffs
Precondition	Player chooses offline mode
Post condition	No
Activation	No

Main flow of event (success)	1. Player enters staff manage region. 2. The system display the pop up for managing the staffs. 3. Player choose upgrading tab. 4. Player choose staff type to upgrade. 5. Player choose to spend cash or watch ads to upgrade base stats of the staff type. 6. Purchase success. The gloves or boots of the upgraded staff type changed color, indicating the stat has changed.
Alternative flow of event	4a. Max upgrade reached. The system notifies that max upgrade of the staff type has been reached. 5a. Not enough money to spend. The upgrade-with-money button is grayed out, indicating that there is not enough money to upgrade the staff type.

Table 2.3: Specification for use case "Upgrade staffs"

2.3.4 Specification for use case "Manage main character"

Use case code	**UC004**
Use case name	**Manage main character**
Actor	Player
Description	Player do actions to upgrade base stats of the main character
Precondition	Player chooses offline mode
Post condition	No
Activation	No
Main flow of event (success)	1. Player enters the upgrade site. 2. The system display the pop up for upgrading the main character. 3. Player choose stat to upgrade. 5. Player choose to spend cash or watch ads to upgrade base stat of the main character. 6. Purchase success. The gloves or boots of the main character changed color, indicating the stat has changed.

Alternative flow of event	3a. Max upgrade reached. The system notifies that max upgrade of the main character stat has been reached. 4a. Not enough money to spend. The upgrade-with-money button is grayed out, indicating that there is not enough money to upgrade the main character.

Table 2.4: Specification for use case "Manage main character"

2.3.5 Specification for use case "Setup connection"

Use case code	**UC005**
Use case name	**Setup connection**
Actor	Player
Description	Player set up connection to join a previous hosted game
Precondition	Player chooses online mode
Post condition	No
Activation	No
Main flow of event (success)	1. Player enters the IP address of the host and local address. 2. Player click connect button to attempts to connect to room. 3. The system display a waiting popup indicating that the connection is in progress. 4. Connect success.The system display a form to configure the display name and the color of the main character for the game session.
Alternative flow of event	2a. Wrong IP address or no host found. The system hangs at the waiting popup.

Table 2.5: Specification for use case "Setup connection"

Input for use case "Setup Connection" form

No	Field	Description	Required	Validation	Example
1	IP address	The host IP address	Yes	In the form of IPv4 addresses	192.168.1.100

2	Name	Display name of the player	Yes	Maximum 14 characters	KidifyMe

Table 2.6: Input for use case "Play Online" form

2.4 Non-functional requirement

Along with the requirements about business, the system needs to satisfy non functional requirements such as performance and ease of use.

The system needs to be stable and run smoothly on both low-end devices and high-end devices with low latency and energy consumption. The system should also be well designed so that it can be extended later in order to fulfill future user requirements. While data schema changes should be avoided, player's offline session data should be capable of migrating to new version.

The user interface and user experience is also need to be detailed examine for the sake of users' experience. The control of the game and the core game play should be user friendly and easy to catch up even with no experience player. In addition, visual effects, sounds and haptics feedback also need to be carefully designed so as to maximize players' experience during the game session.

CHAPTER 3. METHODOLOGY

Chapter 2 analyzes the basic functional, non-functional requirements and criteria that the game Funny Zoo need to satisfy. In the following chapter, theoretical background of game engine and multiplayer framework that are used to develop the game will be discussed, as well as the reason for which they are used in this project.

3.1 Game engine

A game, at the very least, needs a way to draw stuffs on the screen. The standard solution includes a component for loading and managing textures and 3D models, a component for handling shaders and another one for GPU rendering. Other games also needs to take input from players, do simulate physics, play sounds, do networking, run animations. Doing everything from scratch is costly and prone to errors so developers often reuse systems that have already been developed and battle-tested.

A game engine is a software framework primarily designed for the development of video games, and generally includes relevant libraries and support programs. The core functionality typically provided by a game engine may include a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, and localization support. In many cases, game engines provide a suite of visual development tools that are built on top of reusable software components. These tools are generally provided in an integrated development environment to enable simplified, rapid development of games in a data-driven manner. Game engines arise as a need for all the core functionality needed, right out of the box, to develop a game application while reducing costs, complexities, and time-to-market - all critical factors in the highly competitive video-game industry.

For mobile game development, the list of game engines currently available on the market is vast. Some of the most popular choices are Unity[1], Unreal Engine[2], GameMaker Studio[3], Godot[4]. Each platform has its own set of features that is better suited to different requirements. The merits and downsides of those 4 game engines can be summarized in table 3.1.

[1] `https://unity.com`
[2] `https://unrealengine.com`
[3] `https://gamemaker.io/en/gamemaker`
[4] `https://godotengine.org`

	Unity	**Unreal**	**GameMaker**	**Godot**
Supported platforms	Mobile, desktop, web, console, VR	Mobile, desktop, web, console, VR	Mobile, desktop, web, console	Mobile, desktop, web, console
Advantages	Easy to use for beginner developers Many platform supported Great support for mobile development	Open-sourced Has more tools and functionalities Suitable for AAA games	Built-in plug and play system Focused on 2D games	Open-sourced Use node-based interface Cross-platform development environment
Popular developed games	Pokémon Go, Monument Valley, Call of Duty: Mobile, Beat Saber, Cupheah	Fortnite, Werewolf: The Apocalypse – Earthblood, The Matrix Awakens	Samurai Gunn 2, Webbed, Super Hiking League DX	Kingdoms of the Dump, Haiki, Until Then
Pricing	Free for personal use	Free for creators, educators and publishers	Multiple tiers including free	Entirely free
Scripting languages	C#	C++	GML	GDScript, C++ and C#

Table 3.1: Game engines comparison.

With the objectives of building a small-sized game targeting mobile platforms, Unity comes ahead as the first choice for game developers. There are 2.8 billion[5] monthly active users involved with content operated or made by Unity game engine in 2020. In the same year, applications built with Unity generated 5 billion downloads per month. These statistics can be attributed to a collection of features and technologies that Unity gaming solution has to offer.

Perhaps, one of the biggest reasons Unity is preferred amongst game developers worldwide is the ability to build, manage and deploy games cross-platform. Unity provides platform abstraction, allowing the same game to run on various platforms with few, if any, changes made to the game source-code. With this advantage in mind, our mobile game can be built and tested on development environment (Windows/Mac) and then deployed to mobile platforms (Android/IOS) with ease.

Unity exposes a very transparent method for composing game architectures. Games are structured around 2 main concepts: GameObjects and Components. A GameObject can have many Components attached. Each Component defines a particular behavior (e.g. "Renderer" renders graphics on the screen, "Rigidbody" takes care of physics simulation). Developers can also define custom Components in forms of scripts to create complex interactions and logic. The editor windows (scene, game, hierarchy and inspector) provide quick access to all object's properties without the need to dive into the code all the time.

Apart from robust built-in architecture and tooling, Unity also features its large asset store that comes with a variety of paid and free assets ready for any game project. While Unity does develop some of these, many of them are also made by the community, meaning developers have numerous options to choose from.

The last missing piece of technology that our game needs is networking. Multiplayer support has long been a part of Unity feature sets and is comprised of first-party solutions and alternative third-party ones. Detailed exploration and evaluation of some netcode framework is discussed in section 3.2

3.2 Multiplayer solution

When developing a multiplayer game, one has to account and solve for inherent network-related challenges that impact the game experience, such as latency, packet loss and scene management. Criteria for choosing the right solution includes a game's genre, the scale of its players and networked objects, competitiveness and feature breadth. There is hardly a perfect, one-size-fits all solution for all kinds of games and experiences. For example, a MOBA game implemented upon a P2P

[5] `https://create.unity.com/2021-game-report`

topology with deterministic rollback like Heroes Strike[6] will have completely different netcode requirements than a FPS game running on a dedicated game server with server authority for cheat prevention, such as Fortnite[7]. Our choice of network solution was made by identifying the correct netcode type that fits our game and a corresponding network framework implementing that netcode type.

3.2.1 Netcode type

Unlike a local game where the inputs of all players are processed and executed instantly in the same simulation or instance of the game, an online game contains several parallel simulations (one for each player) where the inputs from their respective players are received instantly, while the inputs for the same frame from other players arrive with a certain delay. The goal of video game networking is to create the illusion that two or more players are sharing a single environment [4]. It is the job of network engineers to ensure these simulations stay in sync and create a overall responsive and crisp experience for players under various internet conditions. The extend to which desynchronize resolutions are enforced depends on genres of game. Generally, there are 2 solutions to the netcode problem: delay-based and rollback. Definitions and advantages of each techniques will be briefly discussed below.

Delay-based netcode is the extension of a simpler and classic solution called "Lockstep". In lockstep model, the game progress is paused until all inputs are collected. Turn-based games are generally implemented this way since the strategic nature of them requires players to think for a few moments. A half-second delay in this case is not very noticeable. To successfully apply lockstep netcode to fighting games, however, we need to add what is called input latency. Input latency effectively creates a delay between inputs and their corresponding actions and ensures both players have received the other's input before the simulations proceed. The end result is that all local game versions are in sync. A combination of lockstep model and input latency is commonly known as "Delay-Based Networking"[8] and has been used for decades in video games such as "Age of Empires"[2]. A developer can implement a delay-based online mode with little or no change to the core game loop. Delay-based networking, while simple, comes at the cost of responsiveness and flexibility. If there is a spike or variance in latency (due to a poor connection), the game can be paused for a long time until it receives all the inputs.

Rollback networking approaches the synchronization problem in a much different angle. Rollback removes the need for both player's inputs to continue the simu-

[6] https://wolffungame.com
[7] https://epicgames.com/fortnite/en-US/home

lation. At each loop, the game predicts what the remote player will input and render the game accordingly. When the real input arrives, rollback checks if the prediction it made was correct. An inconsistency between predicted and real input will "roll-back" the game state to a previous committed state, simulate the new inputs, and render the correct game state. Rollback is quite effective at concealing lag spikes or other issues related to latency in the users' connections, as any correction made in successive frames is small and unobtrusive. A downside of rollback model is that its implementation is more complicated than its delay-based counterpart. Tony Cannon, creator of GGPO[8], writes[3] that a game wishing to implement rollback networking must meet three criteria: The game must be *deterministic*, the game must update the game state *independently* of input sampling and visual rendering, and the game must be able to *save and load* the game state on demand.

Funny Zoo, being a real-time fighting game, incorporates ideas borrowed from both networking techniques. To synchronize players' movement under real-time constraints, we build a predictor component similar to what rollback netcode has. Other events and triggers in the game that happen infrequently are processed with delay-based approach. The last step in creating multiplayer solution is to select a framework that satisfies these requirements.

3.2.2 Netcode framework

Unity, since version 5.2, has been offering a first-party netcode solution called UNet[9]. UNet is composed of 2 parts: HLAPI (high-level API) and LLAPI (low-level API). HLAPI lets developer build game with ready-to-use components and provides access to Unity engine and editor integration. On the other hand, LLAPI is a real-time transport layer that is built upon optimized UDP based protocol. LLAPI is for those who need to build network infrastructure or advanced multi-player games. While being deprecated and currently in maintenance mode, UNet has created a strong foundation on which Unity's next first-party netcode solution is built.

MLAPI[10](mid-level API) is a open-source solution being developed (at the time of writing) to become the new Unity netcode foundation. It is customizable and adaptable for the needs of many multiplayer game types[10]. MLAPI offers a great breadth of mid-level features like NetworkedVars, scene management, remote procedure calls (RPCs), messaging, and more. This solution offers an abstraction layer to enable the swapping of different transports depending on the topology and plat-

[8] `https://ggpo.net`
[9] `https://docs.unity3d.com/Manual/UNet.html`
[10] `https://docs-multiplayer.unity3d.com`

forms that the game is shipped to[9]. The solution assumes some form of client-server topology – either a dedicated game server (DGS, where clients connect and interact with the game on a central server) or a listen server (where one client hosts the server for the match).

Photon PUN[11](Photon Unity Networking) is a third-party Unity package for multiplayer games. Its key features include RPCs, serialization, hosted relay and simple matchmaking. Higher-level features like prediction and delta compression is not availble. Photon PUN uses a mesh topology solution (direct P2P), in which each client synchronizes everyone else's data. PUN is suitable for developers making a 2–4 player game that is cooperative or very casual. While having simple interfaces, it struggles with scale in any implementation[9] and may not be viable for very fast-paced games.

DarkRift 2[12] is a fast and highly performant low-level networking solution. Its feature set contains bi-channel TCP and UDP, serialization, and customizable logging. DarkRift requires a dedicated server topology and the entire solution is multithreaded by default. While it does not offer any of the mid- or high-level features, the solution was highly rated for performance. DarkRift targets developers who are in need of full control over networking and are comfortable building additional netcode.

After careful consideration, MLAPI was selected as our primary framework to handle the multiplayer part of Funny Zoo. MLAPI's tight integration with Unity leads to low learning curve and fast iterations. MLAPI being fully free and open-sourced is another key attribute leading to the final decision.

[11] `https://photonengine.com/en-US/PUN`
[12] `https://darkriftnetworking.com`

CHAPTER 4. EXPERIMENT AND EVALUATION

Chapter 3 provides an insight over the game engine and the multiplayer framework used in the project. This chapter will go into details of the architecture design for the system as well as some class design and deployment characteristics.

4.1 Architecture design

4.1.1 Software architecture selection

Game development is different from traditional software engineering in that there are no real functional requirements and the customers buy and use the software only because it is engaging and fun[11]. One of software engineering challenges in game development has been identified[6] as *Diverse Assets* - game development is not simply a process of producing source code, but involves assets such as 3D modes, textures, animations, sound, music, dialog and video. In Funny Zoo, software elements are grouped into 3 major components in which common software architectures are applied.

a, Game architecture

A common technique that is typically shown in beginner-level game development is to organize game objects in a hierarchical manner. While this approach works for small game, the practice of organizing game objects in a hierarchy introduce inflexibility and maintenance problem. A good game object system must be scalable such that game object types can be added easily to the system regardless of the number of types already defined[7]. Unity engine, amongst numerous other game engines, provides developers with component-based game object system which uses composition, rather than inheritance, to define game object types and their behaviors. Game objects are the fundamental objects in Unity that represent the actors of the game world (characters, props and scenery). A game object defines their functionalities and behaviors through a list of components. For example, a solid cube object has a *MeshFilter* and *MeshRenderer* component, to draw the surface of the cube, and a *BoxCollider* component to represent the object's solid volume in terms of physics.

In figure 4.1, a player is represented as a game object accompanied by a list of components. The *Transform* component is used to store a GameObject's position, rotation, scale and parenting state and is thus very important. *Rigidbody* enables physics-based behaviour such as movement, gravity, and collision. *CharacterController* component gives the character a simple, capsule-shaped collider and pro-

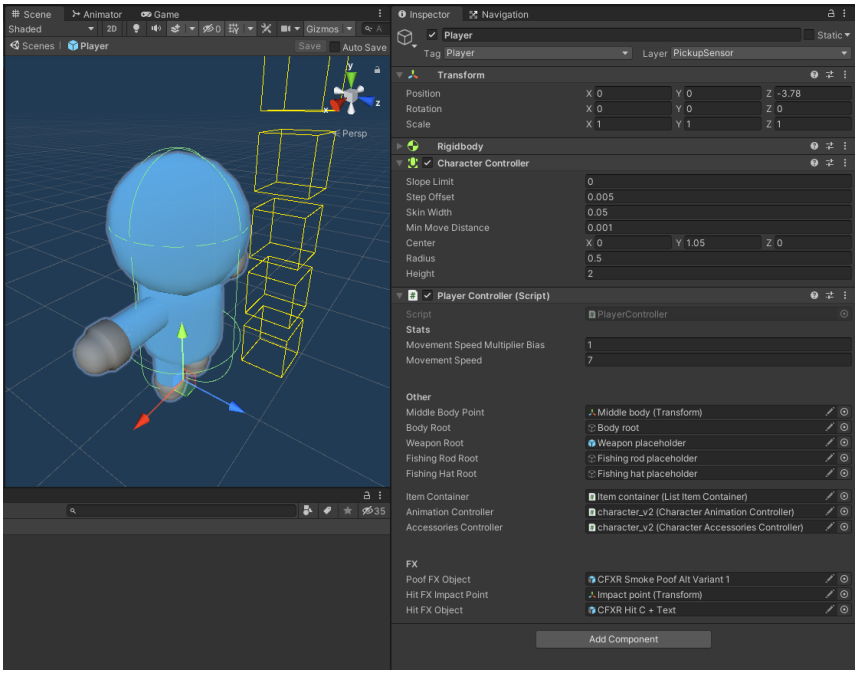


Figure 4.1: Player object in component-based system

vides special functions to set the object's speed and direction. Input handling and extra behaviors are handled inside *PlayerController*, which is a custom script written in C#. Since the behaviors are decoupled from game objects, the game object system not only becomes much more maintainable, but also very flexible[7].

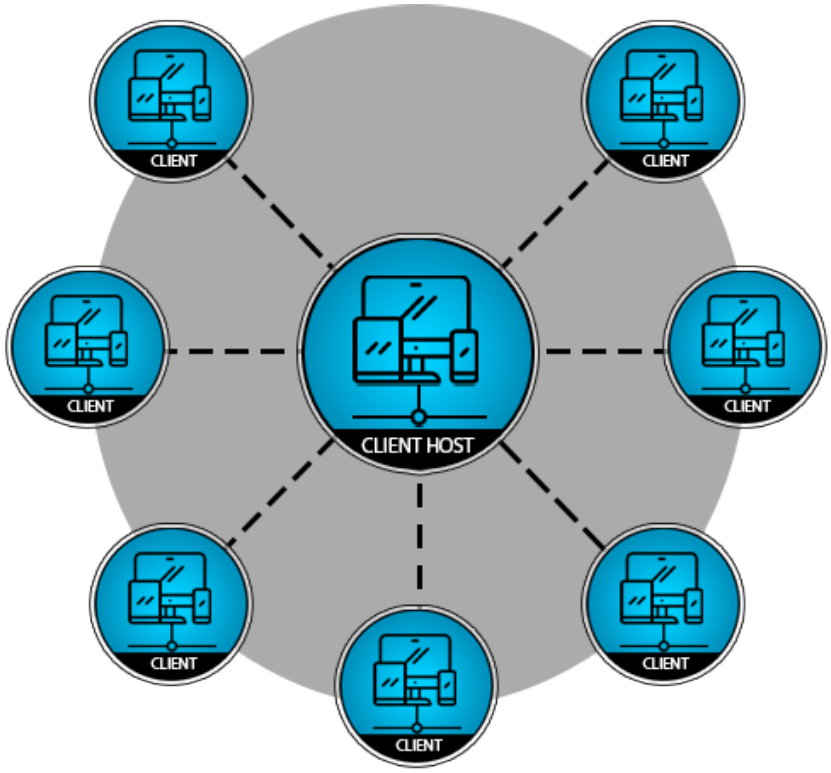
b, Network architecture

Client hosted (Listen server) is selected as the primary network topology for multiplayer experience in Funny Zoo. In this architecture, a player will host the game server (own the game world) and other players will connect to it (see figure 4.2). Client hosted model does not require any special infrastructure or forward planning to set up, which makes it common at LAN parties where latency and bandwidth issues are not a concern. MLAPI framework (mentioned in 3.2.2) also supports other setup such as *Relay Server* and *NAT Punchthrough* though its modular transport system.

c, User interface architecture

For user interfaces, MVP pattern is used to structure all screens and popups in our game. Model–view–presenter (MVP) is a derivation of the model–view–controller (MVC) architectural pattern. The model is an abstraction defining the data to be displayed. The view is a passive interface that displays data (the model) and routes invocations (from user) directly to the presenter. The job of the presenter is to retrieve data from repositories (the model) and format it for display in the view.

In Funny Zoo, the *view* part of MVP is implemented with the help of uGUI



Source: Unity multiplayer docs

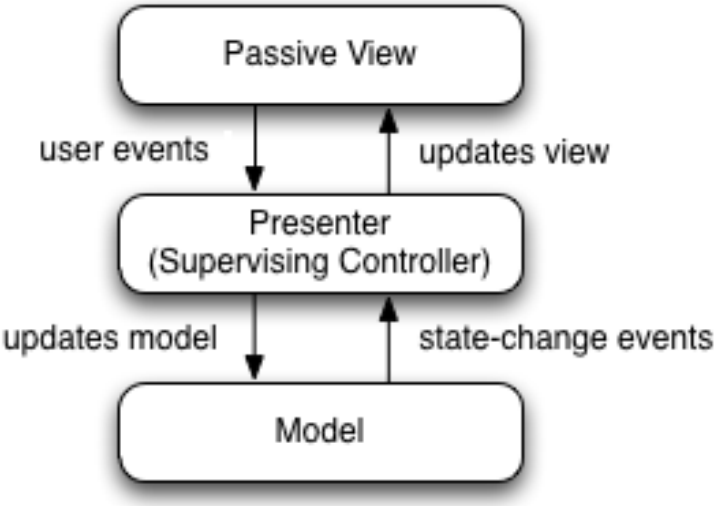
Figure 4.2: Player object in component-based system

package. uGUI (Unity UI) is a set of tools for developing user interfaces for games and applications. It is a GameObject-based (see a) UI system that uses Components and the GameObjects to arrange, position and style user interfaces. A *presenter* is a custom defined component talking to the *view* and the *model*. Visual components in uGUI exposes setter properties that the *presenter* uses to set the data. The *model* is a global state container enhanced with reactive data streams that *presenters* can subscribe to. A detailed explanation of this architecture will be presented in section 5.2.

4.1.2 Overall design

One advantage of using Unity engine as the primary developing environment is that the entire code base is generally enclosed in a single project. The overall code structure is depicted in package diagram 4.4.

Config and *data* are the foundational packages of the project. *Config* package contains game related information such as properties of game objects, level layouts and game design parameters. These configs are typically implemented as ScriptableObjects - Unity data containers that support creating, editing and saving inside Unity editor. The *data* package provides models that the game depends on to save



Source: `https://github.com/neuecc/UniRx`

Figure 4.3: MVP pattern

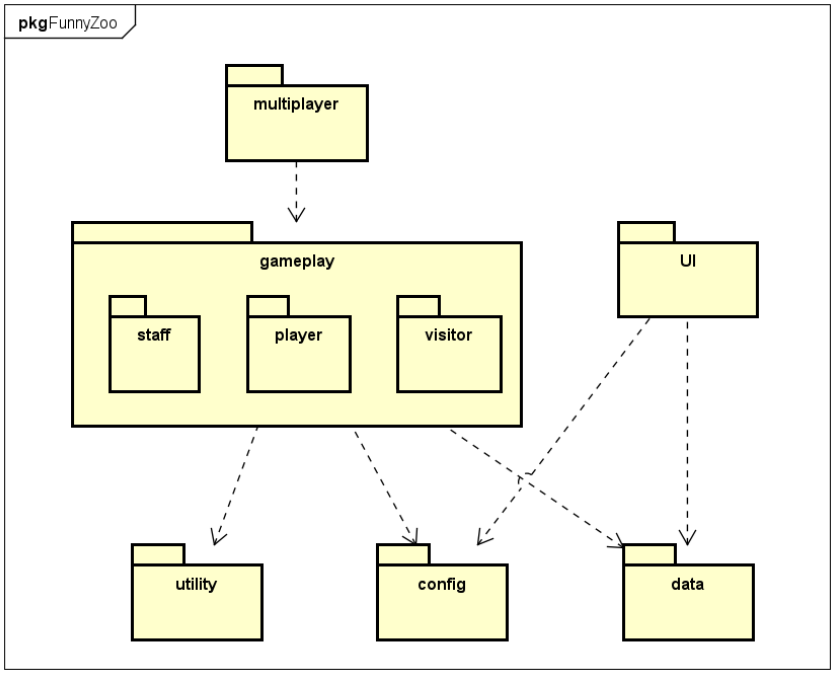


Figure 4.4: General package design

and load data. Game data is stored as a single JSON blob on PlayerPrefs. PlayerPrefs is an Unity interface for storing preferences between game sessions. This interface is handled differently based on which platform the application runs on. An instance of game data includes game progress, player and in-game characters' upgrade state and various user tracking information.

Gameplay package is the core package defining behavioral components for all elements in the game. Some of these components are player and character con-

trollers which are grouped and defined inside sub-packages such as *staff*, *player* and *visitor*. *Gameplay* package also involves game manager and controllers for other interactable entities.

Multiplayer package extends some behaviors described in *gameplay* package to handle multiplayer scenario. It is composed of a multiplayer game manager and custom controllers for player that interact with the network system.

UI package is dedicated to building all user interfaces in the game. As a mobile game, Funny Zoo's UI mainly consists of a main screen screen and several popups. *UI* package contains presenter components (in MVP pattern) that bind to corresponding uGUI objects (see previous section c).

Last but not least, *utility* package provides helpful routines and methods that make game development easier. Built on top of Unity API, *utility* contains helper functions for math (vector) calculation as well as game object and components manipulation.

4.1.3 Detailed package design

A substantial portion of the code base is related to the player. Figure 4.5 shows a list of main classes and packages that handle player behavior as well as their relationship to each other. `PlayerControllerBase` provides common functionalities such as handling user's input and controlling player game object. `PlayerControllerBase` refers to `PlayerConfig` for predefined paramters and `PlayerData` for run-time information. `PlayerController` and `NetworkPlayerController` can then define additional or custom behaviors on top of the base implementation for offline and online cases respectively.

Regarding user interfaces, `UIPanel` and `GUIManager` are the 2 main classes that make up the UI system. `UIPanel` provides common functionalities for UI elements in the game: transition animations and life cycle hooks. `GUIManager` is responsible for arranging these `UIPanel`s and display them on the screen. Generally, screens inherit directly from `UIPanel` while popups extends `UIPopupPanel` - a subclass of `UIPanel` that has extra convenient methods for creating and managing popups. An overview of the UI system is presented in figure 4.6

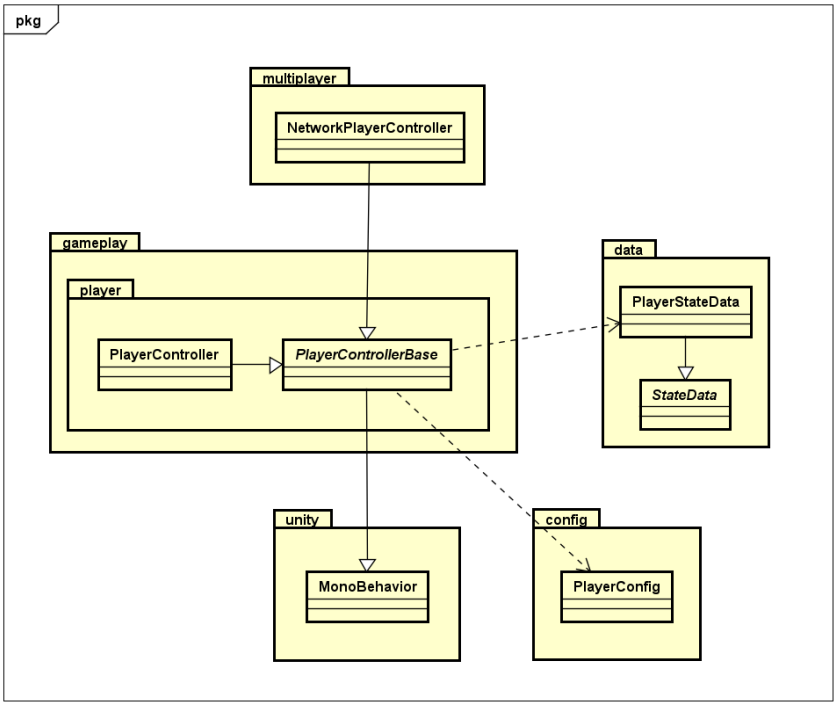


Figure 4.5: Detailed package design for player behavior

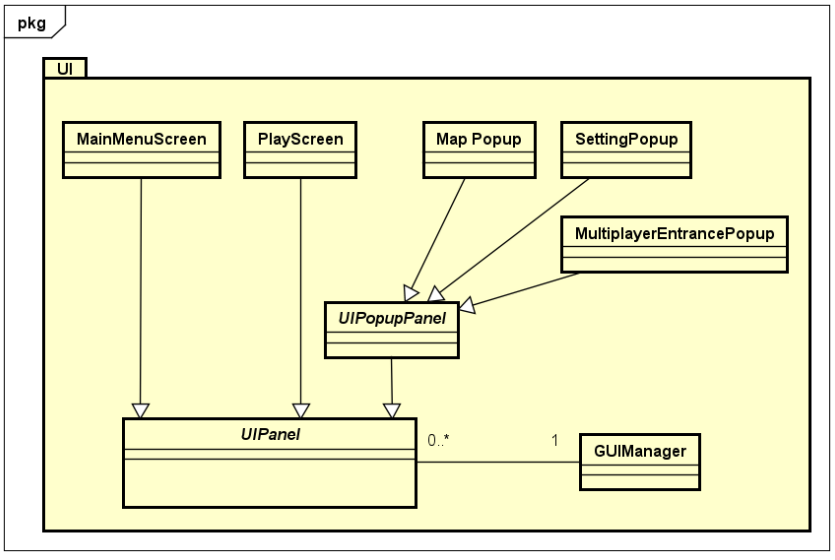


Figure 4.6: Detailed package design for UI

4.2 Detailed design

4.2.1 User interface design

a, Screen configuration standardization

Display

Screen resolution:	720 x 1280px
Number of colors supported:	16,177,216 colors

Screen

Size:	720 x 1280px
Location of buttons:	Located on the sides of the screen
Main color theme:	#72FFFF, #FFE59D, #3CCF4E
Location of messages:	Located on the bottom of the screen
Text:	Lilita One, size at most 72px

b, Screen specifications for Play Screen

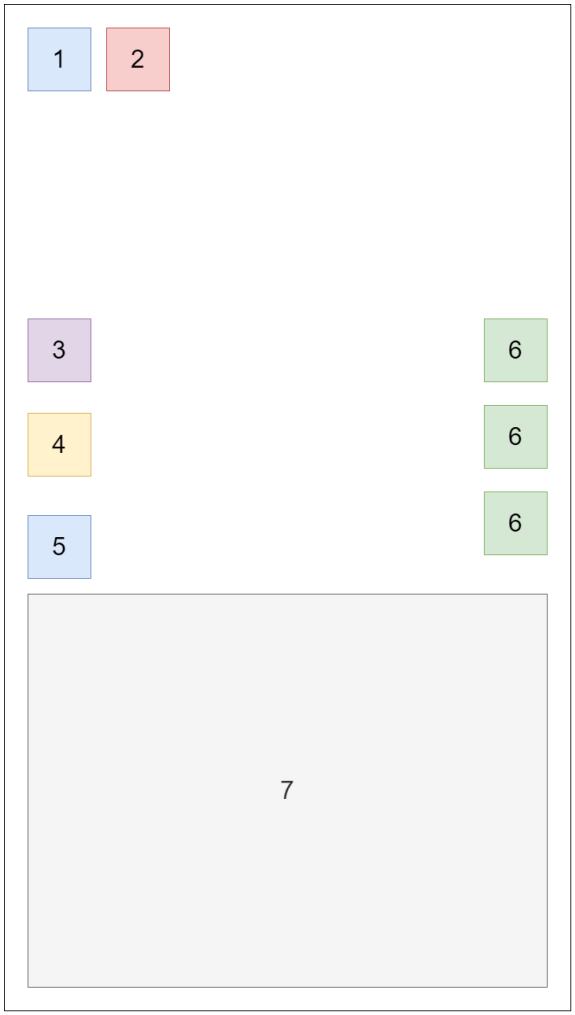


Figure 4.7: Play screen template

No.	Control	Operation	Function
1	Setting button	Click	Show setting popup
2	No ads button	Click	Show no ads popup
3	Map button	Click	Show map popup
4	Sites button	Click	Show sites popup
5	Beginner gift	Click	Show beginner gift popup
6	Buffs button	Click	Apply a buff to the game
7	Movement zone	Drag	Control main character movement

Table 4.1: Play screen specifications

c, Screen specifications for Manage Staffs Screen

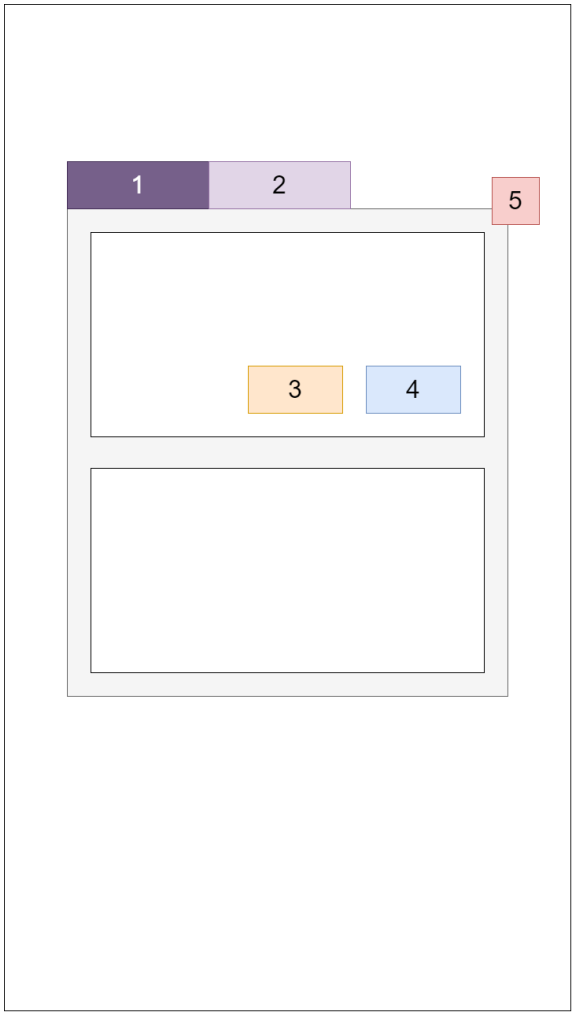


Figure 4.8: Manage staffs screen template

No.	Control	Operation	Function
1	Hire staffs tab button	Click	Show hire staffs tab
2	Upgrade staffs tab button	Click	Show upgrade staffs tab
3	Money upgrade button	Click	Upgrade staffs using money
4	Ads upgrade button	Click	Upgrade staffs by watching ads
5	Close button	Click	Close manage staffs popup

Table 4.2: Manage staffs screen specifications

d, Screen specifications for Setup Connection Screen

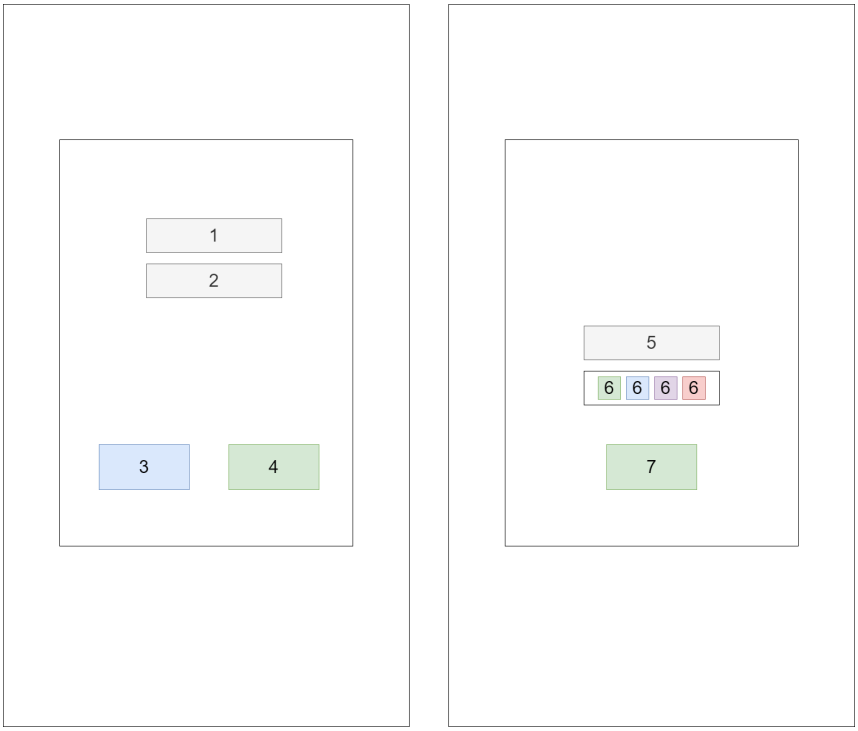


Figure 4.9: Setup connection screen template

No.	Control	Operation	Function
1	Local IP address input field	Input	Contain local IP address
2	Host IP address input field	Input	Contain host IP address
3	Host button	Click	Host a game
4	Join button	Click	Join a game
5	Player name input field	Input	Contain player display name
6	Player color button	Click	Choose player display color
7	Ready button	Click	Mark state as ready

Table 4.3: Setup connection screen specifications

4.2.2 Class design

As previously stated in section 4.1.2, the `PlayerController` class and its accompanying components play an essential part in interfacing between player and the game. A detail design is demonstrated in figure 4.10. Adhering to "component over hierarchy" architecture previously outlined in section "Game architecture", the player game object is comprised of several components: *ItemContainer*, *CharacterController*, *CharacterAnimationController* and *CharacterAccessoriesController*. All of these components are specially designed to ensure high reusability and low coupling - they can be added to other character-like entities (e.g. staffs and visitors) without any modification. A brief explanation of each component is given in table 4.4.

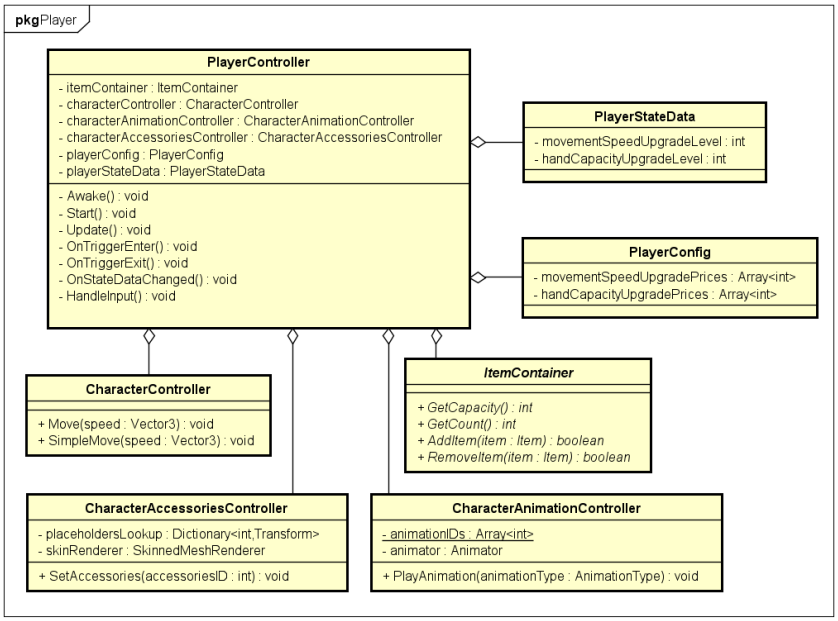


Figure 4.10: Detailed class design for Player

Component	Purpose
ItemContainer	Provide ability to store, take and give items
CharacterController	Do movement constrained by collisions without having to deal with a rigidbody
CharacterAnimation-Controller	Play a set of predefined animations on player model
CharacterAccessori-esController	Add/remove accessories (e.g. bags, hats and glasses) to/from player object

Table 4.4: List of components attached to player object

Regarding methods defined in `PlayerController` class, some of which are special functions handled by Unity Message System. *Awake*, *Start* and *Update* are

called by Unity engine on special life-cycle events - when the script is being loaded, being enabled and being updated every frame. Inside `PlayerController` initial setup is done on *Awake* and *Start* while input from player is handled on *Update*. *OnTriggerEnter* and *OnTriggerExit* are also Unity message methods invoked when a collision with another object start and stop. Figure 4.11 illustrates how `PlayerController` uses these 2 messages to detect if player enter an interactable region.

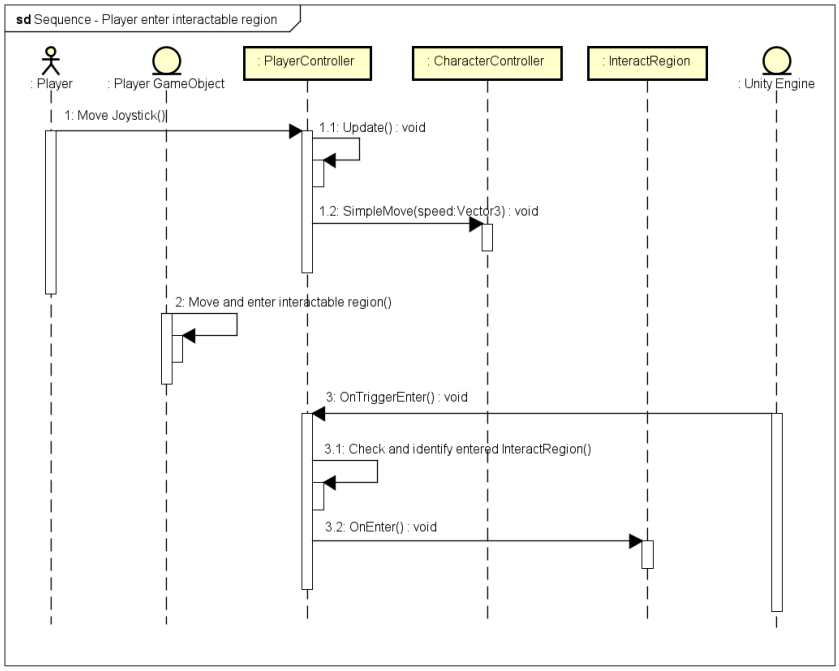


Figure 4.11: Sequence diagram for "player enters interactable region" case

Figure 4.12 elaborates on detailed implementation of `PopupPlayerManage` as an example of the UI system. `PopupPlayerManage` holds a list of upgrade items and presents them to the player. Each displayed item in the list is of concrete type `PlayerUpgradeItem` and can bind to `PlayerStateData` to retrieve and update player stats. After entering the entry point - the *Show* method, the popup subscribes to child components' events in `RegisterEvents` and then refreshes items' state. A complete sequence of interactions between the player and this popup is described in figure 4.13

Class design for multiplayer system in Funny Zoo is a bit unusual. A normal application built under client-server architecture would have separate code bases for client and server. Communication between systems is done through means of exchanging messages in a request–response messaging pattern. To streamline and standardize the data exchange process even further, the server may implement an application programming interface (API)[1]. In this way, data exchange can be

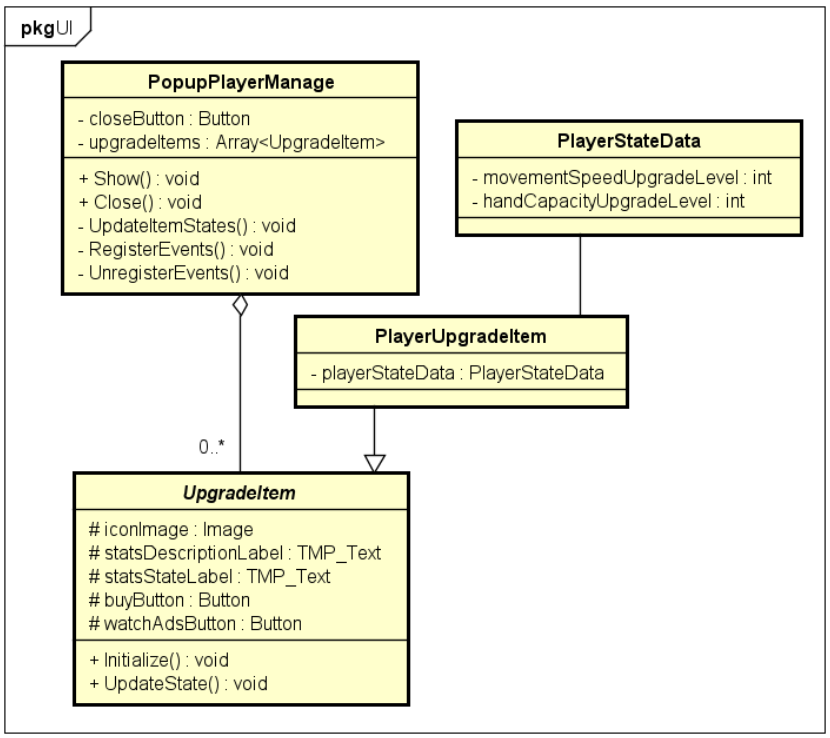


Figure 4.12: Detailed class design for PopupPlayerManage

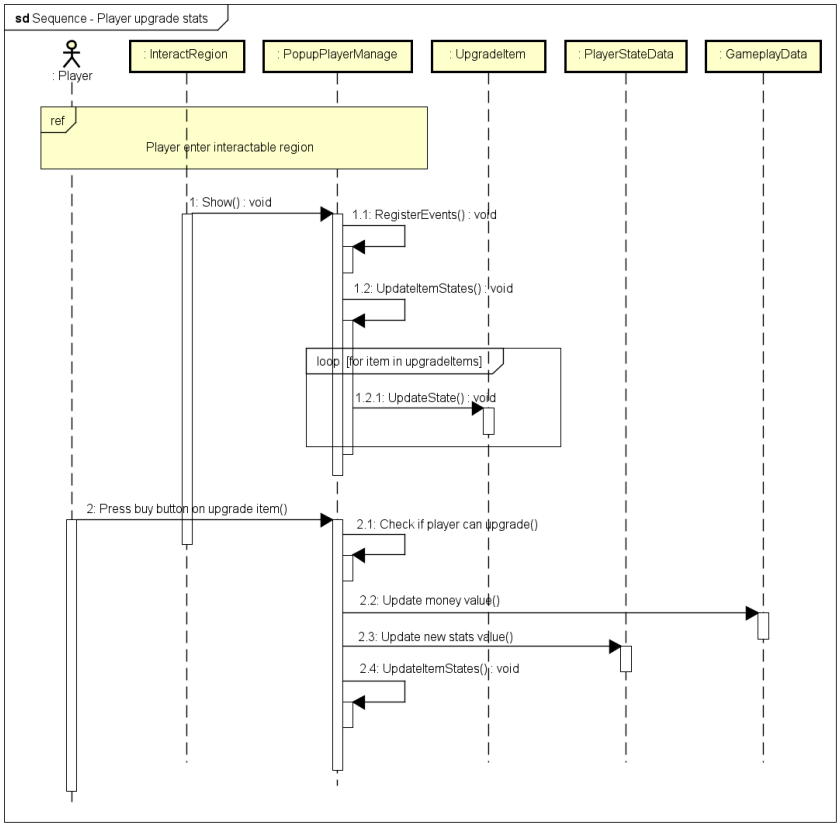


Figure 4.13: Sequence diagram for "player upgrades character stats" case

performed cross-platform - client and server could be built with radically different technologies. However, it is often beneficial to share code between systems

and minimize development friction. This is especially true for game development as game simulations at client-side and server-side generally need to do the same thing. Code sharing helps to maintain consistent and deterministic behaviors across network processes. MLAPI takes code sharing to another level by allowing game components logic for client and server to be defined in the same source file. The application can then check if it is running as either and switch to corresponding code paths.

An demonstration of the above design is showed in diagram 4.14. MLAPI supports exchanging messages by invoking Remote procedure call (RPC). `NetworkGameplayManager` uses this feature to inform clients of various game events such as game started event and game ended event. In sequence diagram 4.15, a machine acts as a host and the other tries to join as a client. The order in which players initiate connection does not matter as Unity transport layer transparently handles low-level UDP networking. In client hosted model, both players are treated as clients and only one of them need to run additional server-side logic to handle incoming connections and setup the game.

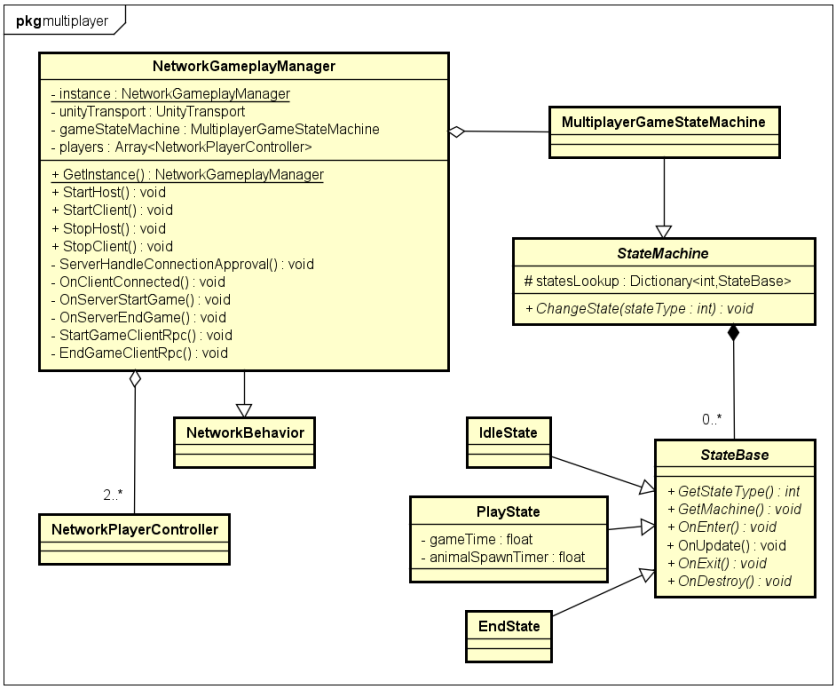


Figure 4.14: Detailed class design for network game manager

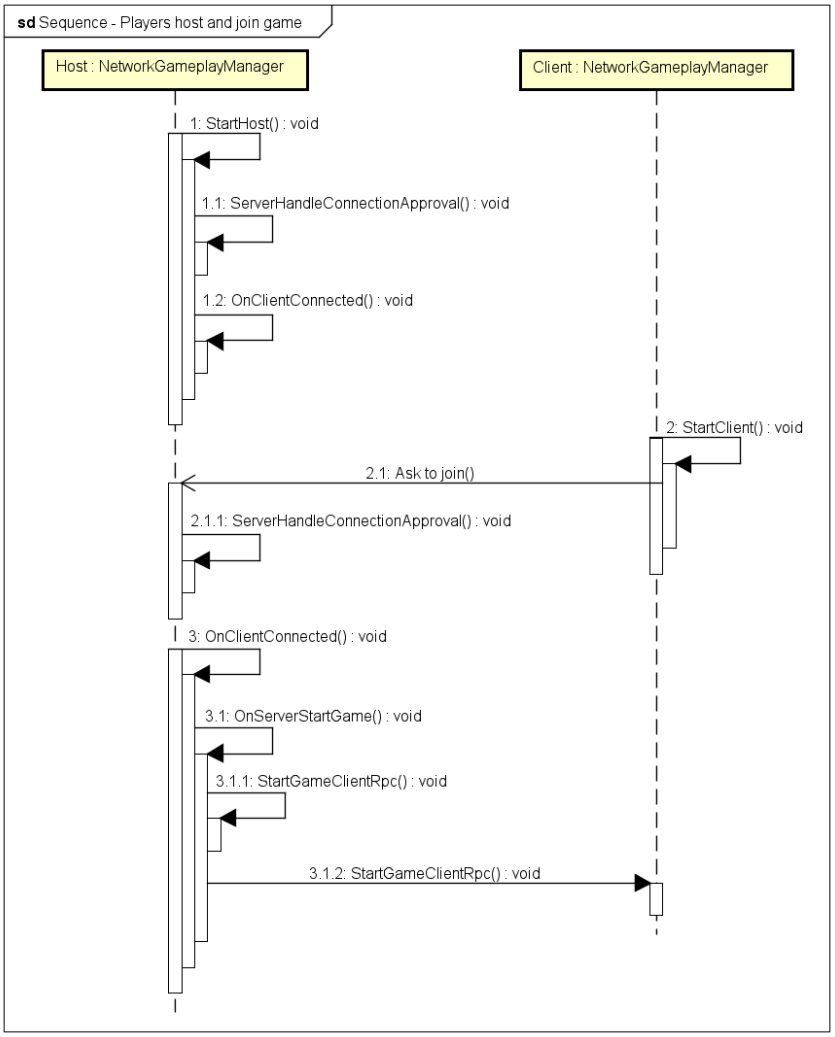


Figure 4.15: Sequence diagram for "players host and join game" case

4.3 Application building

4.3.1 Libraries and tools

Purpose	**Tools and libraries**	**URL**
Programming	Visual Studio 2022	`https://visualstudio.microsoft.com`
Prototyping game	Unity 2019.3.27f	`https://unity.com`
Provides networking capabilities	Netcode for GameObjects 1.0.0	`https://github.com/Unity-Technologies/com.unity.netcode.gameobjects`
Creating and managing animation sequences	Dotween V1.2.632	`http://dotween.demigiant.com`
Create tools and utilities in Unity Editor	Odin Inspector 3.1	`https://odininspector.com`
Handle reactive programming	UniRx 7.1.0	`https://github.com/neuecc/UniRx`
Build, test and deploy	Android SDK 26.0.2	`https://developer.android.com/studio/releases/sdk-tools`
Testing	NoxPlayer 7.0.3.2	`https://en.bignox.com`
Version control	GitHub	`https://github.com`

Table 4.5: List of libraries and tools

4.3.2 Achievement

A typical build output is an APK file. Table 4.6 shows uncompressed asset usage by category in Funny Zoo. Graphics (textures, meshes and animations) take up a substantial part of the total project asset bundle. While the uncompressed size is around 100 mb, Unity managed to produce a optimized build of 78.2 mb as the final product.

Category	**Size**	**Percentage**
Textures	64.2 mb	64.3%
Meshses	4.9 mb	4.9%
Animations	1.6 mb	1.6%
Sounds	1011.7 kb	1.0%
Shaders	1.2 mb	1.2%
Other Assets	2.4 mb	2.4%
Levels	2.4 mb	2.4%
Scripts	8.4 mb	8.4%
Included DLLs	13.6 mb	13.6%
Total User Assets	99.9 mb	100.0%

Table 4.6: Uncompressed asset usage by category

4.3.3 Illustration of main functions

This section will focus on the main features of the game Funny Zoo.

The figure 4.16 describes the main screen, where the players choose to play offline or online mode.

Figure 4.16: Main screen to choose play mode

The play scene of the single player game session is illustrated on the figure 4.17. There is the main characters in this scene, which is under the control of the player. The player can buy food at the production sites, feed the animals by bringing the food to the trough, collect money on the checkout table, buy more animal and open more production sites.

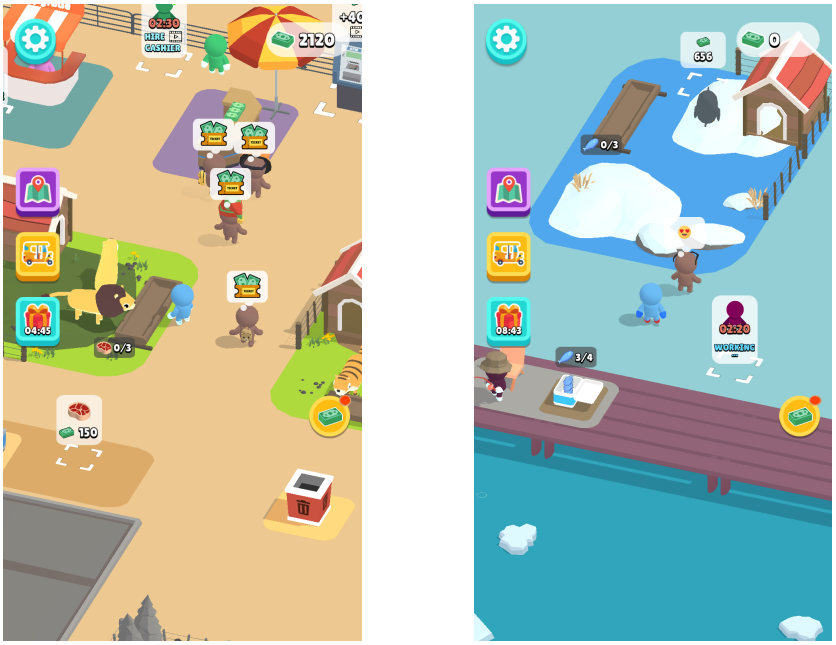


Figure 4.17: Single player scenes where players manage their own zoos

Figure 4.18: Minimap popup

As the map of the single player mode is quite large, in order to aid the player in navigation and locating objectives, the game provides a minimap which can be accessed though a button on the play screen. When the button is pressed, the minimap popup appears and displays the minimize version of the scene.

Another feature of the game is the ability to upgrade main character and staffs

by entering the corresponding upgrade region, as illustrated on figure 4.19.

Figure 4.19: Upgrade main character and upgrade staffs screens

When the players choose online mode, they need to first setup the connection to host or connect to a previous hosted game by entering their IP address and host address, and then choose to select or host a game. After that, player will choose the display name and color of their main character, as illustrated on figure 4.20.

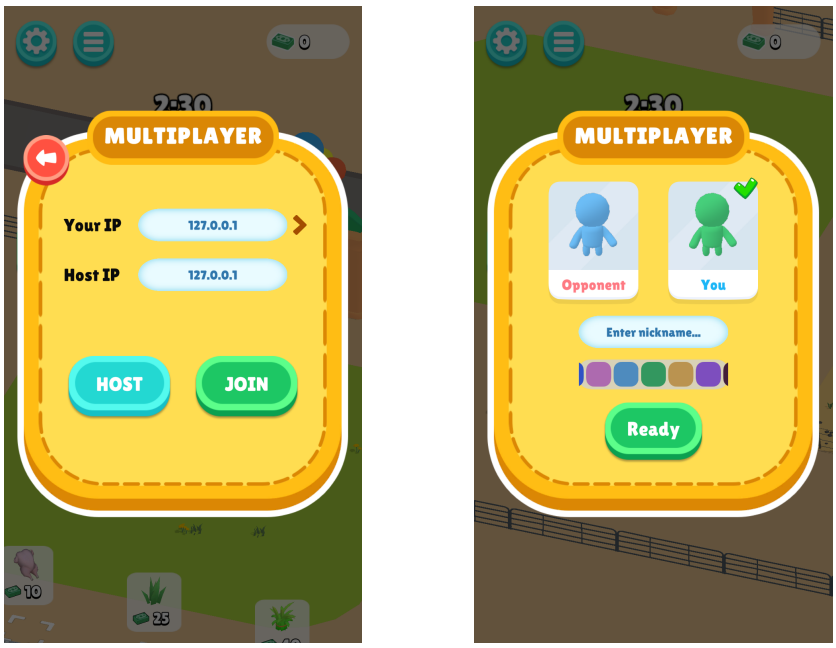


Figure 4.20: Setup connection screen

The play scene of the multiplayer game session is illustrated on the figure 4.21. There are two characters in this scene, which are the player and his opponent. The

player needs to move to the floating popup to purchase the food with the price labeled below to use as bait, and then move to the animal to catch it. There is also an upgrade button on the left for the player to spend money on upgrading main character's stats.

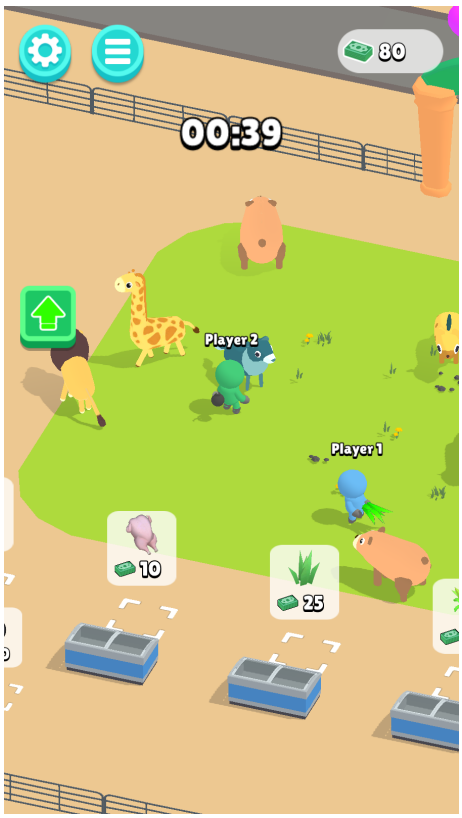


Figure 4.21: Multiplayer scene where players compete each other

4.4 Testing

In the testing process, we use a combination of manual and automated testing techniques to ensure the game works as expected.

Automated testing is integrated directly into the build process. Unity provides callbacks into the build pipeline and executes pre-build and post-build scripts. These scripts make sure application properties (e.g. version code, bundle code and other setting constants) are correct. After the final output artifact is produced, we use *Monkey* to stress-test application. *Monkey* is a program included in Android SDK that runs on emulators or devices and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. The game is expected to function well under these synthesized constraints.

While automated tests are robust and reliable, manual testing is still required to collect in-depth functional and performance metrics. Table 4.7 and 4.8 summarize 2 test suits that are done in person, by clicking through and interacting with the software.

No.	Setup		Output	Conclusion
	Step by step	Expectation		
Manage main character: Control main character and move main character to main character upgrade region to upgrade it				
1	• Step 1: Chose single player mode • Step 2: Move main character to main character upgrade region • Step 3: Click on upgrade button to spend money to upgrade main character	The gloves or boots of the main character change color (colors presents levels of the corresponding upgrade)	The color of the shoes and gloves of main character changed	PASS
Hire staffs: Control main character and move main character to staffs manage region to hire more staffs				
2	• Step 1: Chose single player mode • Step 2: Move main character to HR site • Step 3: Move main character to staff manage region • Step 4: Click on hire staff button to hire staffs	The newly hired staff appears on the map and automatically moves to working place	One new staff shows up and starts working	PASS
Data saving: Player makes changes in single-player session, exits the game and then comes back				
3	• Step 1: Chose single-player mode • Step 2: Spend an amount of money on arbitrary item • Step 3: Exit the game • Step 4: Open the game again in single-player mode	The amount of money player has before quitting the game is preserved	Player's money is saved across sessions	PASS

Table 4.7: Single-player test suite

No.	Setup		Output	Conclusion
	Step by step	Expectation		
Test connection: A game has already been hosted on the local network, the player has the correct host IP address and use it to connect to the hosted game				
4	• Step 1: Chose multiplayer mode • Step 2: Fill in the local IP address • Step 3: Fill in the correct host IP address • Step 4: Click on the join button to attempt to join the game	The waiting popup disappears and the character configuration pop up shows up, meaning the player has successfully joined the game	The character configuration popup shows up for the player to customize the main character before the game starts	PASS
Test connection: A game has already been hosted on the local network, the player has the wrong host IP address and use it to connect to the hosted game				
5	• Step 1: Chose multiplayer mode • Step 2: Fill in the local IP address • Step 3: Fill in the wrong host IP address • Step 4: Click on the join button to attempt to join the game	The waiting popup appears and the application hangs here, waiting for the connection to be established	The waiting popup shows up and do not disappear, indicating that the application is trying to establishing the connection, but not yet succeeded	PASS
Test connection: When two players are in the game, one player disconnected, the other player is automatically disconnected by the system and returned to the main screen				
6	• Step 1: A player connect to a hosted game • Step 2: The host start the game • Step 3: A player disconnected from the game	The other player get disconnected from the game and returned to the main screen	The player whose opponent suddenly disconnected is returned to the main screen	PASS

Table 4.8: Multi-player test suite

4.5 Deployment

For testing purposes, APK (Android Package) is used as the target build format. APKs can either be installed on a emulator or a physical device. A different package format is required to deploy the game to end users. Google Play mandates that application be bundled in AAB format (Android App Bundles), while Apple app store permits the use of IPAs (iOS App Store Package) - application archive files which stores an iOS app.

The final product has been refined and successfully published on Google Play. After the first month, the game has been rated 4.0 stars over more than 1.2 thousand reviews, with quite many positive feedbacks from the community. It also gets downloaded for over 1 million time, which is a pretty impressive number for a casual game.

CHAPTER 5. SOLUTION AND CONTRIBUTION

Chapter 4 has discussed over the design, implementation and deployment of the system. In the following chapter, the contribution of this project will be proposed. Additionally, issues faced during development process, as well as the solution to those problems will be presented.

5.1 Object's position synchronization with smooth interpolation

5.1.1 Problem description

One of many targets that we aimed for when building Funny Zoo's multiplayer feature is a stable and smooth gameplay. A multiplayer game operating over the internet has to deal with several adverse factors that are not present when developing a single-player one. Latency, which in the context of games means the amount of time between a cause and it's visible effect, often shows up as the primary adverse factor. An example of latency can be a button press on joystick and the fighter character moves in response to said button press. For some game genres, the real-time nature implies that response times are important, and in a networked environment this means that round-trip delays must be kept to a minimum[5]. However, in case of Funny Zoo, RTT is hardly the principal source of latency as the game is mainly played over LAN setup. Still, the impact of jitter (the rate at which ping changes over a period of time) and packet loss has be taken into account.

To minimize the effect of latency, we implemented a mitigation strategy known as client-side interpolation.

5.1.2 Solution

A naive approach to synchronize object's position or any value between server and client would be turning off all simulation logic in the client and rendering received states from the server as is. This approach will cause some unresponsiveness as the world simulation depends entirely on state packets sent from server. Jitter and packet loss have full control over how choppy gameplay experience is for the client. The overall effect is that the game would only run at sub-optimal framerate regardless of what potential framerate that the client could achieve. Character objects standing at a position in this tick could randomly teleport to another position next tick. A simple and conservative approach that makes no attempt to mitigate delay is referred to as "dumb terminal". Figure 5.1 shows a simplified structure of a dumb terminal client. Note that only the latest state from the server is kept.

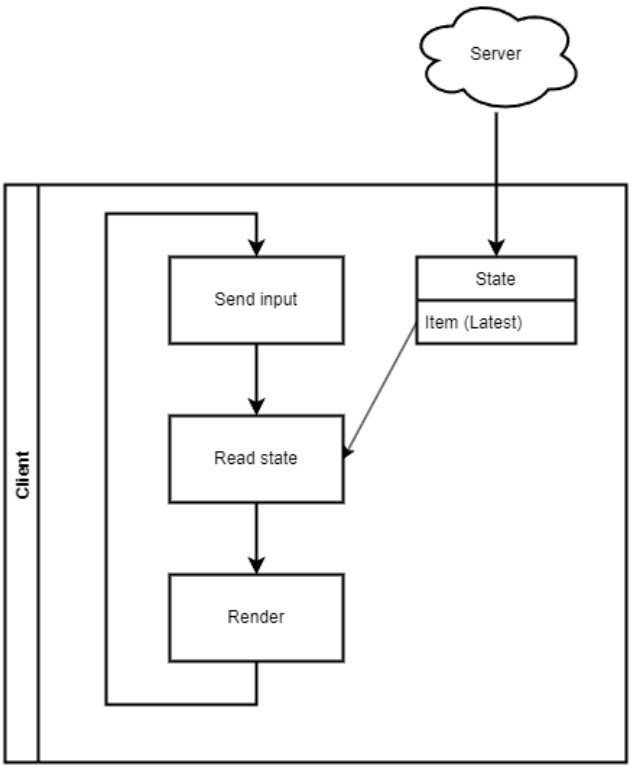


Figure 5.1: "Dumb terminal" client design

In the first iteration of implementing client-side interpolation, instead of just snapping objects to their positions that are transmitted from the server we smoothly interpolate to this state over time. Interpolation is the process of constructing an intermediate data point from few sampled data points. The ultimate goal of doing interpolation is to approximate complicated function (positions of a object) by a simple function. One of the simplest methods is linear interpolation (will be referred to as "lerp"):

$$lerp(a, b, t) = a * (1 - t) + b * t,$$
$$\text{where } t \in [0; 1]$$

If we plugging in current position and server's new position for a and b, the client would be able to catching up to the most recent state passed to us from the server. Using lerp with this setup provides some improvement to the choppiness problem, but it still does not handle jittering well. Another problem with this implementation is that if an object teleports to a new position, the player will incorrectly see a rapid movement in that object rather than an instant jump. We could work around this by sending a flag in addition to the state packet to instruct a "reset", but there is a better way to do synchronization.

Instead of only storing the latest state, we buffer all incoming changes from the server. This provides the client with more data to produces even smoother gameplay at the cost of added processing. A state update packet now includes a timestamp - the time in which the state was snapshotted (see figure 5.2). At each client game loop, we try to read from the buffer and pick out a newest snapshot that is due (i.e. its sent time is less than reference time) as the end target for interpolation. The reading process can consume multiple snapshots from the buffer. The next step will be applying lerp on the last interpolation target and the new one:

$$p_{new} = lerp(p_a, p_b, \frac{t_{render} - t_a}{t_b - t_a})$$

Where p_{new} is the new position, p_a and p_b are old and new snapshot values, t_a and t_b are snapshot timestamps.

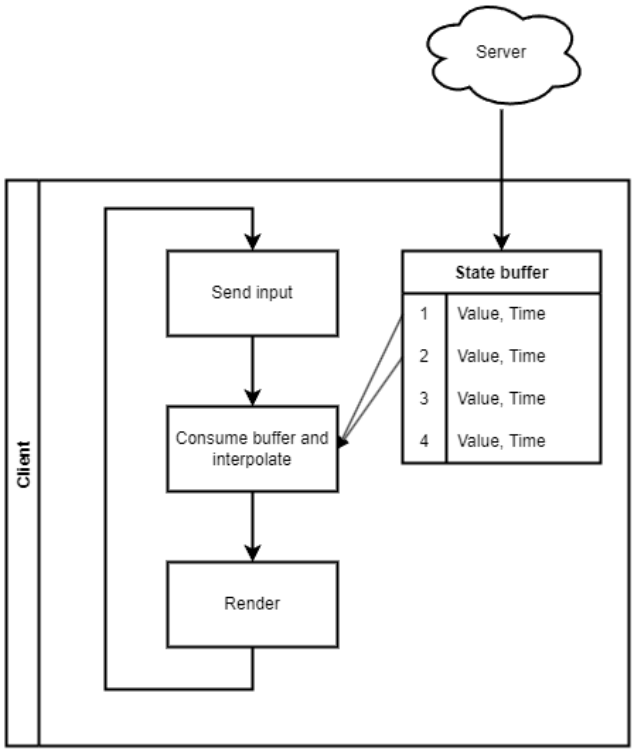


Figure 5.2: Client-side interpolation design

The benefit of storing multiple snapshots is that incoming changes from the server can be consumed at a regular rate client side. Any variation in the network latency is smoothed out over several render frames, giving the player a better experience. Diagram 5.3 elaborates on the effect of having a buffer for state update. At each tick, due to jittering, client can receive zero or more than one update packet from server. The state buffer is designed to ensure there is at least one packet available to render. On some occasions, if client runs out of update packets (due to

packet loss or lag spike), it will enter extrapolation mode. It is usually safe to assume that a moving object will continue in the same direction. When current render time exceeds the last snapshot's timestamp we received from the server, client makes an attempt to estimate a future game state. When the buffer is filled with new packets from the server again, the object's position is updated to the new position and client switch back to interpolation mode.

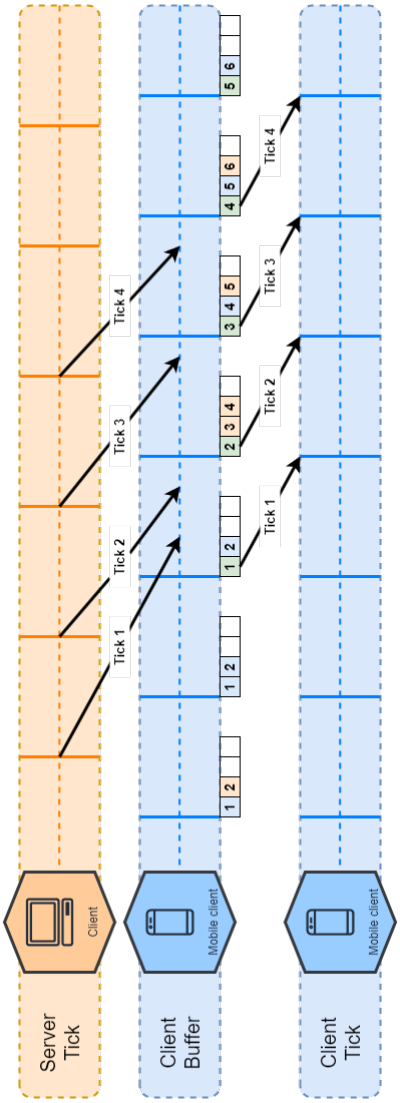


Figure 5.3: Buffered state update

To prevent the state buffer from growing too big, a configurable limit on maximum number of stored snapshots is defined. If this limit is crossed (might happen if client is paused and the buffer is not drained), we immediately clear the buffer and apply the latest received snapshot. It is also a good idea to reduce the frequency of synchronization updates by introducing a threshold value. Changes below threshold values will not be synchronized, which helps lowering CPU usage and reducing bandwidth consumption.

5.2 Reactive user interface architecture for Unity

5.2.1 Problem description

Reactivity is the ability of system to update itself whenever state has changed. In the context of user interface (UI), contents of visual elements should be up to date with the application state changes. The goal of adding reactivity to UI is to eliminate elusive bugs of unsynchronized view and application state, when one forgets to update the view when updating the data. On the language layer, C#/.NET has a built-in event model[1] that follows the observer design pattern, which enables a subscriber to register with and receive notifications from a provider. However, this design pattern is often too complex and requires a significant amount of boilerplate code. On the application layer, Unity offers proper data binding with the advent of *UI Toolkit* package. Nevertheless, at the time of writing *UI Toolkit* is still missing features found in *uGUI* - the UI package that Funny Zoo uses.

5.2.2 Solution

Instead of building a binding layer to bring data binding mechanism to *uGUI*, we opted for reactive extensions with *UniRx* library. In reactive programming, application state is converted to data streams and changes are propagated from producers to consumers. Rather than constantly pulling data sources to update content, recipients await for the arrival of incoming changes. This means that we can design data source in isolation without needing to worry about how UI accesses data. To illustrate the final design, the rest of this section will focus on an example of updating and displaying player's money. Listing 1 displays a small section of `GameplayData` (the global application data store) that is related to money.

```
public class GameplayData : IGameData
{
    public IObservable<int> MoneyObservable => moneySubject;
    private readonly BehaviorSubject<int> moneySubject;
}
```

Listing 1: Example of a data source

Using UniRx, the value of money is wrapped in a mechanism for retrieving and transforming the data in the form of an "Observable". `BehaviorSubject` is a special implementation of an Observable that immediately emits the most recent item and then continues to emit any other items emitted later. This is the desired behavior for data sources as observers should receive the latest state upon sub-

[1] `https://docs.microsoft.com/en-us/dotnet/standard/events`

scribing. Rather than using `GameplayData` as the concrete implementation of the data source, we use dependency injection to provide UI layers with an abstract implementation - `IGameData`. Dependency injection allows gameplay data to be unit-tested without much effort.

To display current money value on a label, the UI presenter need to subscribe to `MoneyObservable`. Listing 2 demonstrates how `PlayScreen` presents a up-to-date money label on screen. For an overview of the UI layer's structure, refer to package diagram 4.6 and section c.

```
public class PlayScreen : UIPanel
{
    private TMP_Text moneyLabel;
    private IGameData gameData;
    private CompositeDisposable eventSubscriptions;

    protected override void RegisterEvent()
    {
        gameData.MoneyObservable
            .SubscribeToText(moneyLabel)
            .AddTo(eventSubscriptions);
    }

    protected override void UnregisterEvent()
    {
        eventSubscriptions.Clear();
    }
}
```

Listing 2: Example of a data consumer

Reactive extensions provided by UniRx integrate seamlessly with FunnyZoo's UI system. By placing subscription calls inside `RegisterEvent`, subscription lifecycle is automatically managed. Data recipients should only be consuming resources when they are active rather than listening all the time. Nevertheless, apart from convenient lifecycle management, Observables seems like just another way to do event handling. The real power comes with the "reactive extensions" - operators that allow one to transform, combine, manipulate, and work with the sequences of items emitted by Observables. These operators allow composing asynchronous sequences together in a declarative manner with all the efficiency benefits of callbacks but without the drawbacks of nesting callback handlers that are typically associated with asynchronous systems. For instance, we can add a smooth update animation to the money label by applying a custom operator to *MoneyObserv-*

able before subscribing. In another example, the buy button associated with a purchasable item is usually made conditionally enabled depending on whether player has enough money. The same `MoneyObservable` can be transformed to a "has enough money" stream with `Select` operator and then applied to the button's state. The end result is exhibited in figure 5.4.

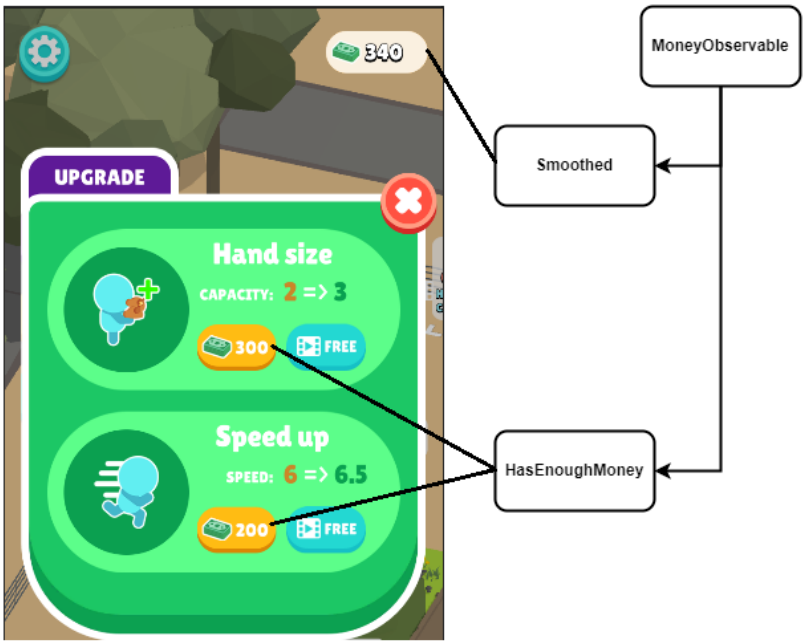


Figure 5.4: Reactive UI applied to popup

CHAPTER 6. CONCLUSION AND FUTURE WORK

Chapter 5 addressed the contributions, challenges, and creative solutions to several complicated issues faced during the application development process. This chapter will summarize achieved results and lessons learned during the graduation project. Chapter 6 then ends with some notes for future application development.

6.1 Conclusion

The Funny Zoo game was born with the aim to entertain mobile players in their free time. Built on top of Unity game engine, the development process was accelerated with the presence of a visual development environment, cross-platform support, and a modular system of components. Featuring an attractive gameplay and a colorful theme, Funny Zoo has been published on Google Play and received a lot of good feedbacks from the community. This is also attributable to an in-depth game flow and contents that some of the casual games on the store lack. Finally, the game comes with a multiplayer competitive game mode for players and their friends to enjoy the game together.

Aside from successfully delivering the final product, we gained some valuable experiences in the process of doing Graduation Thesis. The project opened a chance to come into contact with game design inside and out. Developing multiplayer functionality also gives us a deeper understanding of network architecture in the context of a real-time fighting game.

6.2 Future work

Within the graduation scope, the project has been developed and published successfully to serve the basic need of the players and acquired the target proposed. Still, there are many to improve in the project. Firstly, regarding the game design, the average time to unlock all the contents of the game is not very long, about forty five minutes. This is due to the constraint of development time and can be solved by adding more content in the future. The game is expected to have at least 2-hour length game sessions to bring back more profit. Secondly, the multiplayer game should be extended to serve more than just two players at the same time. Also, a lobby service or a match making system should be integrated into the game in order to make the game more competitive. This target could be achieved by a redesign of the multiplayer game mode, and a more in-depth researching in networking system. Lastly, the game still needs some optimization to run smoothly on low end devices. There are some events where the FPS drop slightly such as when the player interacts with a large amount of objects, or when there are more than 15 visitors in the

scene. Performance optimization requires interacting with low level interfaces of the game engine, as well as applying a more efficient algorithm for solving current problems.

REFERENCES

[1] Boualem Benatallah, Fabio Casati, and Farouk Toumani. “Web service conversation modeling: a cornerstone for e-business automation”. In: *IEEE Internet Computing* 8 (2004).

[2] Paul Bettner and Mark Terrano. “1500 archers on a 28.8: Network programming in Age of Empires and beyond”. In: *GDC*. Vol. 2. 2001. 2001, 30p.

[3] Tony Cannon. “Fight the Lag: The Trick behind ggpo’s Low-Latency Netcode”. In: *Game Developer Magazine* 19.9 (2012), pp. 7–13.

[4] Ivan Dubrovin. “Shared world illusion in networked videogames”. B.S. thesis. 2018.

[5] Tristan Henderson. “Latency and user behaviour on a multiplayer game server”. In: *International Workshop on Networked Group Communication*. Springer. 2001, pp. 1–13.

[6] Christopher M Kanode and Hisham M Haddad. “Software engineering challenges in game development”. In: *2009 Sixth International Conference on Information Technology: New Generations*. IEEE. 2009, pp. 260–265.

[7] Nicolas Porter. “Component-based game object system”. In: *Carleton University* (2012).

[8] Ricky Pusch. “Explaining how fighting games use delay-based and rollback netcode”. In: *Ars Technica* (2019), p. 32.

[9] Unity Technologies. *Choosing the right netcode for your Unity multiplayer game*. 2021. URL: https://images.response.unity3d.com/Web/Unity/%5C%7B305691e0-36c5-4b1a-ae4d-a2e43d4569cb%5C%7D_Unity-Choosing_Netcode-Research_Report-v1_1.pdf.

[10] *Unity documentation*. URL: https://docs.unity3d.com (visited on 06/15/2022).

[11] Alf Inge Wang and Njål Nordmark. “Software architectures and the creative processes in game development”. In: *International Conference on Entertainment Computing*. Springer. 2015, pp. 272–285.