Alexander Tikhomirov

DEVELOPING AN ONLINE MULTIPLAYER GAME IN UNITY

Bachelor's thesis

Bachelor of Engineering

Information Technology

2023

South-Eastern Finland
University of Applied Sciences

XAMK
South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author(s)	Alexander Tikhomirov
Thesis title	Developing an online multiplayer game in Unity
Commissioned by	-
Year	2023
Pages	74 pages
Supervisor(s)	Timo Mynttinen

ABSTRACT

The objective of this thesis was to examine the tools and methods for multiplayer game development offered by the Unity game engine – one of the most popular game engines currently on the market. Only tools provided by Unity Technologies, the developers of the engine, are explored in the thesis. Third-party tools are not covered.

The primary methods of research used were study of literature and online sources, including official Unity documentation, and practical experimentation in the engine. The thesis investigates the theoretical framework of online multiplayer games and common challenges of their development, and gives an overview of Unity engine, its networking solution and accompanying services. Although the explanation of Unity's basic concepts and workflow is provided, some knowledge of C# programming language is required for complete understanding of the material.

As a result of the experimentation with the engine, a prototype of an online multiplayer game was created. Its implementation is documented in the final chapter of the thesis. The prototype presents an example of practical application of Unity's networking toolset and demonstrates the engine's ease of use, accessibility and versatility.

Keywords: Unity, networking, online multiplayer, game development

CONTENTS

1 INTRODUCTION 5
2 CONNECTING PLAYERS 6
2.1 Network topology 6
2.2 Network Address Translation 9
2.3 Lobbies and matchmaking 12
3 COMMON CHALLENGES AND SOLUTIONS 13
3.1 Cheating 14
3.2 Latency 16
3.2.1 Latency in client-server topology 17
3.2.2 Latency in peer-to-peer topology 22
4 UNITY GAME ENGINE 25
4.1 General overview of the engine 26
4.2 Netcode for Game Objects 29
4.3 Unity Gaming Services 33
5 IMPLEMENTATION 37
5.1 Game concept 37
5.2 Starting the project 39
5.3 Main menu 41
5.4 Lobby and relay 47
5.5 Player Prefab 53
5.5.1 Input System 53
5.5.2 Player health 54
5.5.3 Player visuals 55
5.5.4 Player behavior 58
5.6 Projectiles 62

5.7 Match environment and flow 64

6 CONCLUSION 66

REFERENCES 69

LIST OF FIGURES

1 INTRODUCTION

Video games have come a long way since their inception. From basic games like Pong to huge immersive world simulations. From simple text-based and pixel graphics to hyper realistic 3D visuals. From a niche hobby to an enormous and rapidly growing industry.

In fact, the game industry currently represents one of the entertainment industry's sectors showing the most significant growth. From 2017 to 2021 the annual revenue of this sector increased from 120.4 billion USD to 214.2 billion USD and is projected to reach $321.1 billion by 2026. (PwC 2022, 8.) For comparison, the movie industry's revenue in 2021 was 99.7 billion USD without pay TV revenue, and $328.2 billion with it (Motion Picture Association, Inc. 2022, 8-9). So even though it might be too soon to say that video games have overtaken cinema, they're well on their way to catch up with it. As such, the video game industry will remain an attractive career path for programmers and other relevant specialists for the foreseeable future.

Just as the games continue to evolve and the industry continues to develop, so do the tools for their creation. In the past the developers had to program each new game from scratch. Now a wide selection of game engines exists to ease and streamline the process, among them such examples as Unity, Unreal, Source, Godot, etc.

One important aspect of video games is the multiplayer element. Whether as their primary focus or simply an extra mode, many games offer the ability for multiple people to play together, adding the component of social interaction, cooperation and competition to the gameplay. Online multiplayer over the Internet allows players from different geographical locations – friends or strangers – to come and enjoy the game together despite the distance between them.

However, developing an online multiplayer game comes with a set of unique challenges not present when making a single-player or local multiplayer game. The goal of this thesis is to explore the tools and methods for developing an

online multiplayer game using the Unity game engine – one of the most popular game engines in the industry.

The thesis project is largely motivated by my personal interest in game development. As a result of the project, a functional online game prototype will be created. The intention for this prototype is to potentially be developed further into a complete product for release later.

First, the thesis will first examine the theoretical framework of the task: the mechanics of connecting players over the Internet, the common issues present when making an online multiplayer game and the ways to solve them, and finally the Unity engine tools for online multiplayer game development. After that, the implementation of the working game protype will be described.

2 CONNECTING PLAYERS

When it comes to the development of online multiplayer games, some fundamental questions need to be asked first. How do the players find each other? How is the connection between them established? How is it organized?

In this chapter, I will start by explaining the network topologies used in online multiplayer games. Then I will cover the problem of establishing a connection between peers caused by Network Address Translation. After that, the methods of finding players online will be discussed.

2.1 Network topology

A network topology is the specific arrangement of the elements of a network (TIA, n.d.). In the context of online multiplayer games, network topology describes how connection and interaction between the game instances of different players is organized. The topologies used can be grouped into two categories: peer-to-peer and client-server (Figure 1).

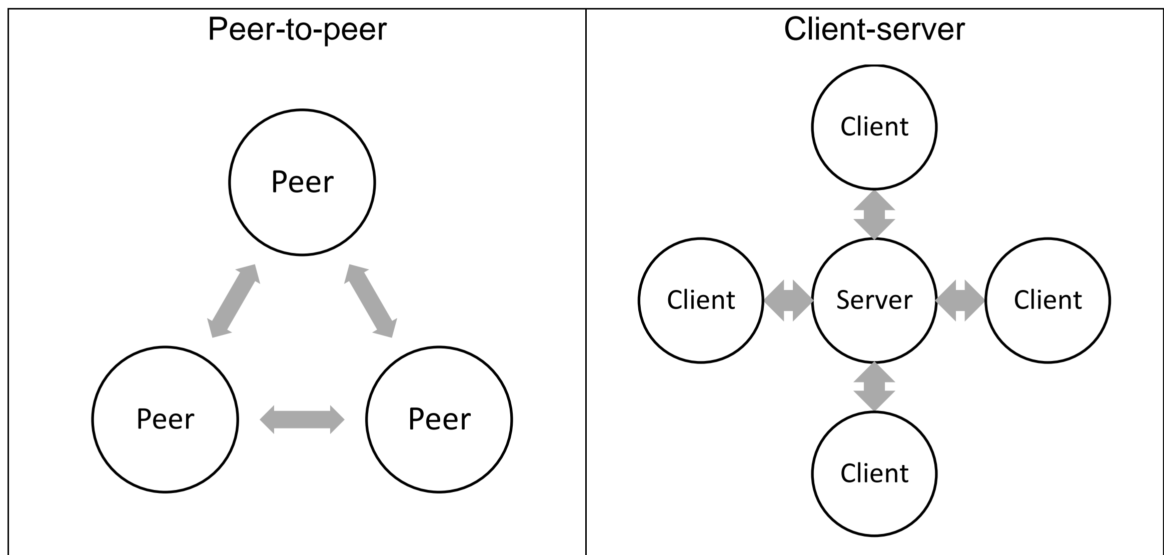

Figure 1. Network topologies

In a peer-to-peer (P2P) topology all the participants are equal, and each communicates directly with all the other ones. This creates a significant bandwidth load that is equally distributed among players. Since the number of connections every player needs to maintain increases with the number of players in the game, so does the amount of packets each participating machine needs to send and receive per second. (Glazer & Madhav 2016, 168-169.)

In a client-server topology, on the other hand, there is one participant that acts as a “server”, which all the other participants – called “clients” – connect to. The server is typically not one of the players, but a dedicated machine in charge of running the game world simulation and ensuring synchronization of the game state between clients. In this topology, overall bandwidth load is decreased as there are only as many active connections as there are clients, and most of it falls on the server, which needs to communicate with all of them, while each client only communicates with the server. (Glazer & Madhav 2016, 166-167.)

Both described topology types come with their own benefits and drawbacks. If the peers are located close enough geographically, the P2P model offers decreased latency – since all traffic travels directly from one player to another with no need to go through a central server. It also lacks a single point of failure that the server would be – if any of the peers disconnect, the game can continue without them,

while a problem with the server will bring down a client-server game without use of failover servers. Additionally, P2P topology doesn't need the infrastructure required by the client-server topology, reducing the upkeep costs for the developers. (Pandey, 2022.)

Alternatively, the client-server model provides better protection against cheating and cyber-attacks. Since the server has authority over the game state and can validate data sent by the players, it is more difficult for cheaters to manipulate the game to their advantage. Since the clients only communicate with the server and don't know each other's Internet Protocol (IP) addresses, they are shielded from a potential cyber-attack by one of the game's participants. Moreover, dedicated servers located in datacenters have faster Internet connection and can handle more simultaneous players than a peer in a P2P game that has to work with a consumer-grade "last mile" connection. This means that the client-server topology is more scalable and can accommodate more players in a single match. (Pandey, 2022.)

A compromise solution combining elements of the two topology models is also possible. Some P2P games assign one of the players to be the middleman in all the communications between other peers and for matchmaking purposes. Similarly, the games using the client-server topology may allow one of the players to act as the server instead of a dedicated machine in a datacenter. The special player that combines the roles of a client and a server is called a "host" or a "listen server" (as opposed to a "dedicated server"). These cases blur the line between P2P and client-server topologies and offer a mixture of their pros and cons. For example, using the listen server can mitigate the single point of failure issue inherent in the client-server model. It is done by implementing the mechanism known as "host migration" – if the host player disconnects, it is possible for another one to take over the role of the listen server from them, allowing the match to proceed (Glazer & Madhav 2016, 168).

2.2 Network Address Translation

Every device connected to the internet is assigned an IP address which is used for sending and receiving data over the net. Internet Protocol version 4 (IPv4) address is a number that is 32 bits long, which means there are exactly 4,294,967,296 possible addresses in total. This may seem like a large number, but after IPv4 was created it became clear very soon that it will not be nearly enough to accommodate the explosive growth of the internet. (Panek 2020, 212.)

This forced the adoption of certain mechanisms to preserve the available addresses. Although Internet Protocol version 6 (IPv6) with a much larger address pool has been created as a replacement for IPv4 to resolve the address shortage issue, IPv4 will remain in widespread use for the foreseeable future. As such, it is important for game developers to consider the nuances of connecting players while using IPv4.

Network Address Translation (NAT) is one of the mechanisms created to preserve the address pool, and the one that makes connecting players online quite tricky. The purpose of NAT is to map a group of private IP addresses to a single public one. The player’s gaming devices – either personal computers, gaming consoles or mobile phones – are usually connected to a local area network (LAN), which connects to the wide area network (WAN) via a router. Each device connected to the player’s LAN will have its own private IP address from a specific IP range reserved for LAN usage. However, for communication with the internet, all of the player’s LAN devices will use a single public address allocated to the player by their internet service provider (ISP). The router will do the translation between private and public addresses by using port numbers and building a table mapping LAN address-port combinations to unique WAN address-port combinations. Table 1 provides an illustrative example of a NAT table.

Table 1. NAT table example

Device	Private (LAN)		Public (WAN)	
	IP	Port	IP	Port
PC1	192.168.0.2	100	150.150.0.1	5001
PC1	192.168.0.2	200	150.150.0.1	5002
PC2	192.168.0.3	100	150.150.0.1	5003
PC3	192.168.0.4	200	150.150.0.1	5004

As can be seen from the example, each device on the LAN has its own private IP. NAT creates public address-port pairs for each private address-port pair in use by adding a unique port number to the single available public IP address, like an index (Tanenbaum & Wetherall 2010, 454). This kind of NAT is also known as “Port Address Translation” (PAT) or “NAT Overload”.

The private addresses are reused by countless local networks across the world and are not publicly routable. It is impossible to communicate with a device over the internet while only knowing its private address. But even if the public address is known, the addressee’s router will not know which device on the LAN behind the public IP to direct the traffic to. Additionally, often the firewall or the router will reject incoming packets from the internet if they aren’t already expecting any from the sender. All of this makes establishing a direct connection between players for a P2P or a host-based client-server game a complicated matter. Note that client-server games with dedicated servers don’t have this issue, because in this case the server will have a static publicly routable address that the players can use to connect to it.

One way to circumvent this hurdle is by having the players configure appropriate entries in the NAT tables manually. This requires some knowledge that the players may not have, so it’s generally not a good idea to ask them to do that. (Glazer & Madhav 2016, 57.) Furthermore, the players may not even have access to the router to make these changes. Fortunately, there are other methods to get around the issue.

Session Traversal Utilities for NAT (STUN) is one such method. It is a procedure that allows two devices on the internet to establish direct connection through NAT by using a proxy. The proxy would be a publicly routable server that helps the connecting devices exchange the information (public IP addresses and proper ports) required to create the necessary entries in their routers' respective NAT tables to begin communicating directly with each other without further involvement of proxy. This is also called "NAT punchthrough" or "hole punching". (Glazer & Madhav 2016, 57-59.)

Depending on the specific implementations of NAT used by the connecting parties, punchthrough may not succeed. In particular, the so called "symmetric NAT" maps unique external address-port combinations based not just on the internal address and port, but also the destination of the outgoing packet. Ergo, it will assign a new public address-port combination for every new communication target and will use different address-port combinations for messages sent to the proxy and to the other player, making the STUN process impossible. (Glazer & Madhav 2016, 59-60.)

Traversal Using Relays around NAT (TURN) is the protocol that can be used in such an event. Unlike STUN, which uses a proxy only for the initiation of the connection between players, TURN uses a publicly routable proxy as a relay to pass the traffic from one peer to another. Since both players can communicate with the publicly routable proxy, this method works regardless of the NAT implementations involved. However, this increases the latency of the connection, and is more resource-heavy for the proxy.

STUN and TURN are often used in conjunction with another protocol called Interactive Connectivity Establishment (ICE). The purpose of ICE is to find the best route for connecting two machines on the internet through NAT. If ICE fails to establish a direct link through STUN, it falls back on a TURN connection via relay (Meddane, 2020).

2.3 Lobbies and matchmaking

So, if two (or more) people on the internet want to play a game together, how can they find each other online? Simply exchanging their IP addresses is certainly an option, but not a very good one. For starters – as explained above – that would require the players to make changes to their firewall and router configurations, which not all of them may know how to do. Then, the players would also have to use some method of communication to negotiate the match and swap the necessary information. This may be feasible if you're looking to play with a group of friends that you can talk to on the phone or using a messenger app, but what if you just want to play with random people online? Not only would you need to find them first, but potentially openly share your IP address with strangers as well, which might carry unfortunate security implications. Needless to say, the entire process of arranging a match manually becomes a serious hassle.

Thankfully, most online multiplayer games today implement much easier ways to find people to play with. The general idea is that there is a server on the internet that the game is preconfigured to connect to. For games like massively multiplayer online role-playing games (MMORPG) it can be the actual server that runs the game, since they are typically limited in number and known in advance. Other games might allow players to host game matches on their machines or even run their own dedicated servers, that the game would have no knowledge of. Such games may instead implement a "master server". A master server's task is to facilitate the connection between clients and servers, which can be organized in a few different ways. (Stagner 2013, 8.)

One way to arrange this process is through so-called "lobbies". Both game servers and clients contact the master server. The former are added to a list of active servers that the master server keeps track of and provides to the latter (Stagner 2013, 8.). The clients can then choose a server to connect to. The intermediate interface shown while connected to a game server before the match starts and usually displaying the list of participating players is called "lobby". The term is also applied to the game server itself together with the connected clients. The lobbies can be private of public. Any player can join a public lobby (provided

there are vacant player slots available), but joining a private lobby is more complicated and may require entering a special code or receiving an invitation from the lobby owner.

Another method that can be used is matchmaking. The players select certain parameters for the game they seek to play (a game mode or a map, for example) and connect to the master server. The server adds mutually compatible players in “buckets” and, once the bucket is full, chooses one player to be the host for the following match or connects the players to a dedicated game server. (Stagner 2013, 9.)

Other variations of these mechanisms for connecting players together exist as well. Many services and platforms used for this purpose – such as Valve’s Steamworks and Epic Games’ Epic Online Services – implement additional helpful features like user accounts, ability to add other players to a friend list, block them, create a player group (party) before matchmaking to play together, etc. Some games skip an explicit “lobby” screen in favor of a more direct server browser that otherwise functions similarly to the lobby system described above.

Although these methods necessitate the developers of the game to set up or rent a server, they do a lot to improve the player’s experience. With the process of looking for other people to play with streamlined and simplified, the players can focus on having fun with the gameplay and not concern themselves with the networking technicalities.

3 COMMON CHALLENGES AND SOLUTIONS

As was mentioned in the introduction already, developing an online multiplayer game comes with certain challenges not present when making a single player game. Dealing with these challenges significantly complicates the development. By Van Dongen’s (2015) assessment, adding online multiplayer to a game doubles the amount of work needed to program it.

In this chapter, I will discuss some of the main issues that need to be addressed when creating an online multiplayer game. The first one is cheating by the players. The second one is the communication delay between them.

3.1 Cheating

When people participate in any kind of competitive activities, some might be tempted to cheat in order to achieve victory. This is also very true for multiplayer games. A cheater can easily ruin the enjoyment of the game for everyone else – it is simply not fun to play with an opponent who doesn't obey the rules and has an unfair advantage as a result.

The main concepts that need to be discussed in the context of cheating are trust and authority. There is a wide variety of hacking tools that the players may use to send doctored messages to the server, alter the values of variables, and otherwise interfere with the operations of their local game instance (Stagner 2013, 207). Since we can't ascertain that any given game participant is not using such tools, we can't assume that whatever data a client sends out has not been tampered with. Ergo, we cannot blindly trust the clients.

The idea of authority deals with where the important decisions about the game state (objects' locations, players' health changes, etc.) are made – on the server or on the client. How much authority is given to each is a question that needs to be decided based on the specifics of the game being made. For some casual and cooperative games it might be acceptable to trust the clients and give them authority, but in a highly-competitive game where the participants play against each other, trusting the client may be disastrous and should never be done (Cardoso, 2022a).

To provide an example, if the clients are given authority over hit detection, the cheater may start sending messages to the server indicating that they hit other players when they haven't legitimately done so. Even without cheating, there may be situations where slight desynchronization between the clients' game states leads to one of them detecting a hit when there shouldn't be one. To avoid this

issue altogether, it is advisable to give the authority over hit detection and other important game mechanics to the server. This way, only the server has the final say on whether any player was hit, how much damage was dealt and whether the player character died as a result or not.

The authoritative server should at least be validating the messages coming in from the clients before applying them to the game state and distributing to other players. If the server receives a message that a certain player shot a gun, it should validate that the player character in question is controlled by the client sending the message, actually has a gun, the gun is loaded and not on cooldown, etc. (Glazer & Madhav 2016, 270). Alternatively, the server can simply handle all the game logic, with the clients only sending raw input like keyboard strokes and mouse clicks.

The server authoritative model does well at preventing clients from cheating by sending invalid messages. However, this assumes that the server itself is trustworthy. This is usually true with a dedicated server, but what happens when a listen server is used instead? There’s no guarantee that the player acting as a host will not try to cheat, and the clients normally don’t have the means to validate the data coming from the server. This is unfortunately an inherent weakness of using a listen server that must be accepted. The only certain way of avoiding it is to simply not allow players to host games in the first place. It is possible to mitigate the issue by implementing a sort of hybrid topology where the clients establish direct communication channels with each other, circumventing the server. The clients can then use these connections to exchange the data needed to validate the server’s messages. Sadly, due to possible difficulties with NAT and latency, this is not a very stable or reliable solution. (Glazer & Madhav 2016, 271.)

The concept of authority applies primarily to the client-server model, but in P2P topology it is moot. Since all the peers are equal, there can be no authoritative server to validate and calculate the one correct game state, but the peers still can’t be trusted. To address this, P2P games typically use input sharing and

deterministic architecture. The former means that the players exchange their intended actions, which are then independently executed on all participating machines. The latter means that the game is made in such a way that the same set of inputs or actions will always lead to the same result, with no random chances involved. (Glazer & Madhav 2016, 169.) The peers can validate the inputs they receive from other peers just like a server would (Glazer & Madhav 2016, 270), before executing the corresponding actions, and the deterministic architecture ensures that all of them arrive to the same conclusion and the game state remains synchronized.

All of the above is only relevant in situations when a player is trying to cheat by altering or manufacturing the packets their game instance is sending out. However, this is not the only way a player can cheat. There are certain things that can be done entirely locally. For example, the player can make their opponents visible through walls in an action game or remove the fog of war in a strategy, giving them an unfair advantage via access to extra information that an honest player wouldn’t have. Alternatively, the cheater may employ a bot program that will control their character more efficiently than a human can. The only way to deal with these types of cheats is to use an anti-cheat software application – a special program that detects the cheating tools installed on the player’s computer. (Glazer & Madhav 2016, 272.)

3.2 Latency

Another major problem all online games have to contend with is latency. Although in recent decades we went from home internet speeds measured in kilobits per second to megabits, it still takes time for a network packet to reach its destination over the internet. Even with the widespread adoption of fiber optic cables that use light to transmit data at incredible speeds, the reality remains that truly instantaneous data exchange is simply physically impossible. The time it takes for a message to reach its target and a response to come back to the original sender is called round trip time (RTT) and is used as a measure of latency (Glazer & Madhav 2016, 167).

The distance to travel through the cables and each hop between networks add to the time it takes for a packet to arrive at its destination. That's why the first step to reducing latency is to decrease the length and complexity of the path the data has to take. For this reason, many popular online multiplayer games employ regional servers to connect players from the same geographical zone. When people from Europe play with each other via a nearby server, they will enjoy decreased latency compared to if they were playing with people from America.

It is impossible to eliminate latency completely. Even RTT as small as a fraction of a second can cause issues and needs to be accounted for. Fortunately, there are a number of techniques and tricks that game developers can use to deal with this.

3.2.1 Latency in client-server topology

The concept of an authoritative server has been explained in Chapter 2.1. A fully authoritative server suggests the use of so-called "dumb clients" – meaning that the clients are only used to facilitate player interaction with the server (Engelbrecht 2021, 109) and do not simulate the game world themselves. But what are the implications of such an approach? If the player presses the "right" key on the keyboard, their game can't move the player character right and send the new coordinates to the server, as the client has no authority over the character's position – after all, we don't want a cheater to be able to teleport across the map by sending illicit coordinates to the server. Instead, the client sends a message that the player wants to move right, and then the server – after doing the necessary validation – moves the character and sends their new coordinates to all players (Figure 2). (Gambetta, n.d.a.)

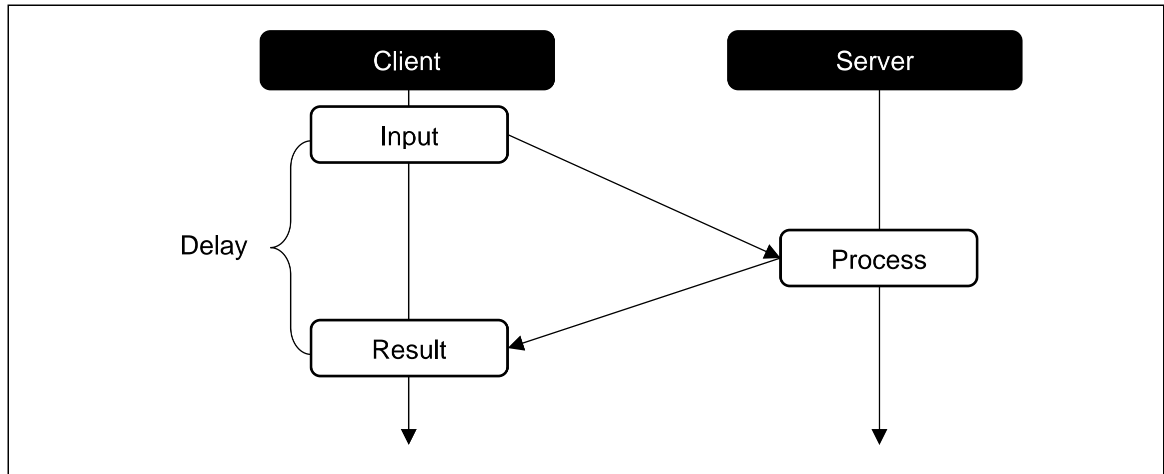

Figure 2. Client-server communication with a dumb client

This means that a full RTT has to pass before the player will see the results of their input. For some slower games – e.g., turn-based strategies – this may not be a big deal, but for a fast-paced real-time game such delay – or lag – can make the game feel frustratingly unresponsive. In a server-authoritative model the only true, canonical game state is the one dictated by the server, and it can't be delivered to the player faster than the internet connection will allow. What can be achieved instead is an illusion of fast response.

Client-side prediction is the technique to create such an illusion. The trick is that after receiving the player's input, instead of waiting for the response from the server their instance of the game will immediately display the expected result of the input. If the game world is deterministic enough, most of the time this result should be the same as the one that will eventually come in the server's response. The issue with this is that the player may – and often will – enter inputs faster than the responses from the server can arrive (Figure 3). (Gambetta, n.d.b.)

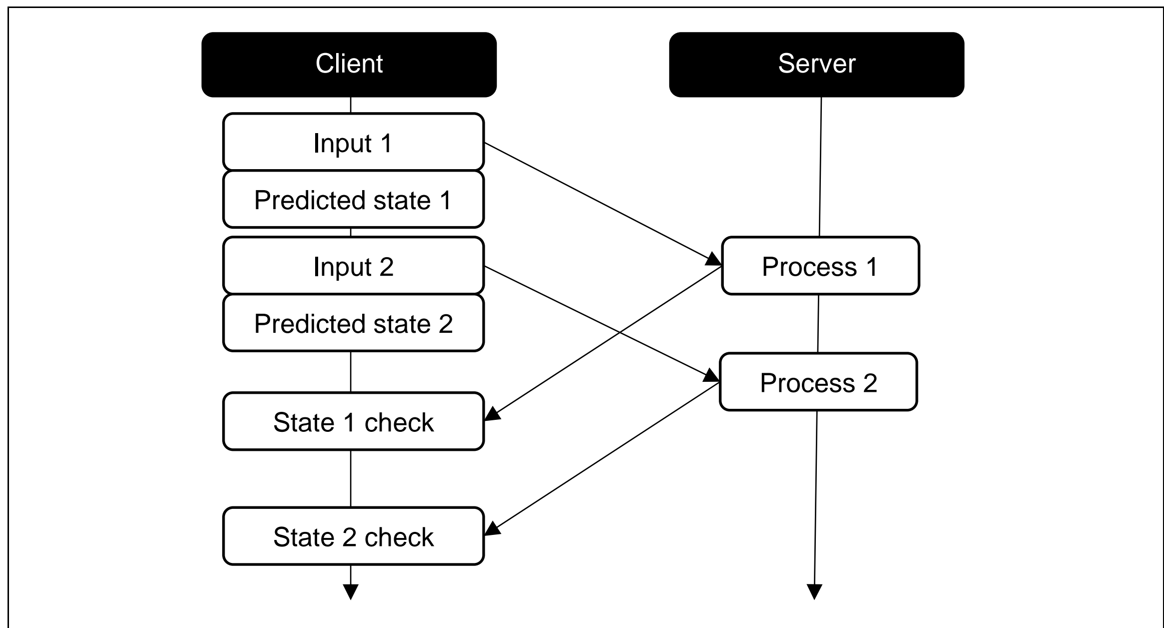

Figure 3. Client-side prediction

So, for example, if the player is continuously moving right, by the time the server's response to the first "move right" input comes, the player might have moved right again. In this situation, even though the client's game state doesn't coincide with the game state dictated by the server, it doesn't make sense to blindly apply the latter, since that would just make the player jump back to the result of the first movement, only to move again to the final position when the next state update from the server arrives. It is important to recognize that what is received from the server is a game state from the past. This discrepancy is resolved by adding a sequence number to the messages, so that when the player gets the response from the server, it can be compared with the previously predicted result of the corresponding input to check if the prediction was right and whether the game state requires correction. This is called server reconciliation. (Gambetta, n.d.b.)

Client-side prediction solves the problem of the player character feeling unresponsive due to lag. But what about the other players' characters? Unlike the local user playing the game, there's no way to know what they are doing in advance, before the server distributes their last state. This issue is exacerbated by the fact that in practice the server refreshes the game state and sends updates to clients at a limited frequency, called tick rate (Nbn, 2017). This is done

to conserve the CPU and bandwidth, and for easier game state synchronization. The unfortunate outcome is that each client receives information about the other clients' behavior not only with delay, but in discrete snapshots. Simply rendering the latest received state will result in choppy movement of the remote players' characters, which will not look good. Depending on the nature of the game, there are a couple of possible methods to mitigate the problem. (Gambetta, n.d.c.)

Some games involve highly predictable motion. E.g., in a racing game the car can't just abruptly change its movement vector unless it runs into something – it takes some time to accelerate, decelerate or make a turn. So, most of the time in the absence of up-to-date information, it is safe for the local game instance to simply keep moving the other players' cars in the same direction and at the same speed as the last update from the server dictates, until the next one arrives and the car's position and velocity can be corrected. This approach is called "dead reckoning", a term originating from naval navigation. (Gambetta, n.d.c.)

Dead reckoning method can only be applied to a specific subset of games. For games with unpredictable movement another technique may be used, called "entity interpolation". The main idea, as the name of the method suggests, is that instead of instantly applying the state updates received from the server, the client interpolates from the previous one over a certain period of time (Gambetta, n.d.c.). Ideally, the interpolation period equals the time between the server's updates – this way the client can expect to receive the next update by the time interpolation to the last one is finished on one hand, and on the other the player's view of the game does not fall behind the server too much due to accumulating updates faster than they can be interpolated (Glazer & Madhav 2016, 237). Figure 4 presents the change of the character state on the client controlling the character with client-side prediction, the server, and another client with entity interpolation.

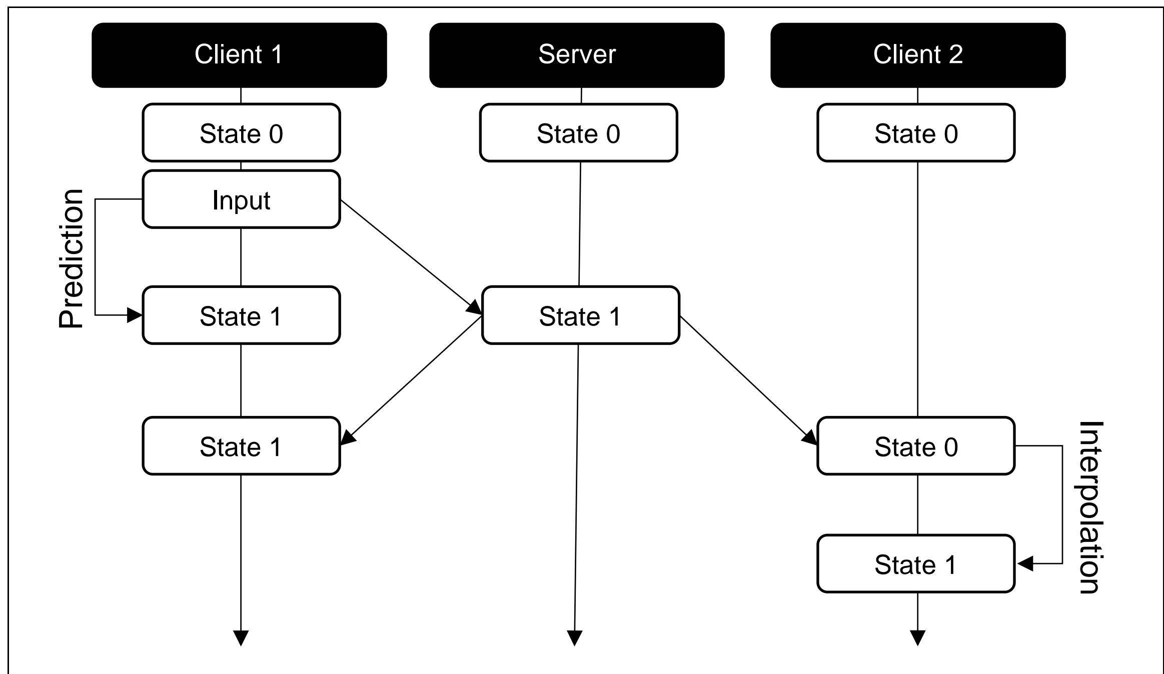

Figure 4. Character state on different machines with client-side prediction and entity interpolation

A client needs to receive the new state of the character from the server – which will happen with a delay – before it can start interpolating to it, which will take additional time. The inevitable consequence of this is that each player sees their own character in the present (or even technically “in the future” when compared to the server’s canonical game state), but the other players’ characters in the past. If the difference in time is not too great, this may be acceptable in some situations. However, when precision is important, this will cause issues. For example, when a player in a shooter game takes aim with a sniper rifle from a great distance trying to get a headshot, even a miniscule motion can change the result. But the truth is that the player is actually aiming at an image from the past. By the time the server receives the message about the shot, the attacked player might have moved away already, even though the shooter clearly saw their bullet hit the target. To resolve this, lag compensation must be implemented. Using timestamps, the server can rewind the game state to the moment the player saw when making the shot, and check for hit detection at that time. Unfortunately, this means that from the targeted player’s perspective, the shot can arrive when it seemingly shouldn’t be able to – like moments after taking cover. But that’s a compromise that needs to be made to mask the latency. (Gambetta, n.d.d.)

3.2.2 Latency in peer-to-peer topology

P2P games have their own practices for solving the latency issue. Perhaps the simplest one is the lockstep model. The games utilizing this model don't actually do anything very interesting and deal with latency in the most straightforward way possible – by simply waiting for all the peers to exchange messages before moving the game a step forward (Gao, 2018). Figure 5 illustrates this architecture.

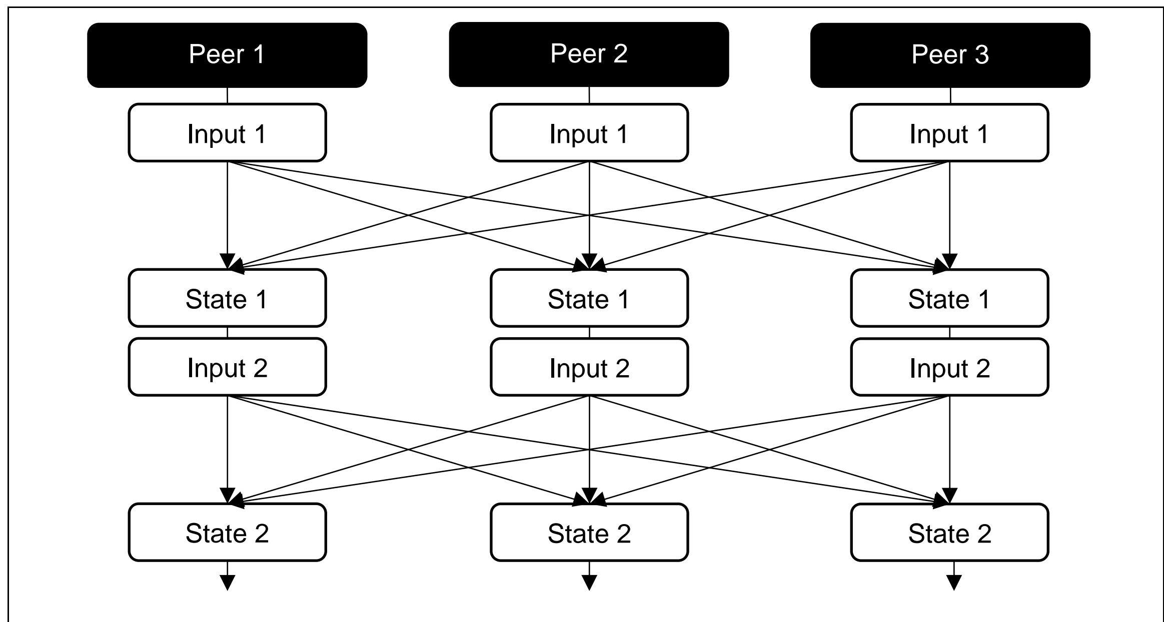

Figure 5. Lockstep P2P architecture model

The game needs to be deterministic and have a fixed tick rate that all peers agree on and use to collect and send inputs in discrete batches (Mattis, 2023a). Each tick the game instances exchange these batches with each other, and upon receiving them from all other participants, calculate and display (or interpolate to) the next frame. On one hand, this ensures that the game state remains perfectly synchronized across all peers. On the other hand, if the latency is high enough, this will cause the game to slow down or even freeze while waiting for updates from all the other players. In fact, even a single player with a slow computer or internet connection will slow down the game for everyone else (Gao, 2018).

To mitigate this problem somewhat and keep the game moving, a deliberate input delay may be implemented. Instead of processing input packets immediately and

then waiting for the next message exchange to be complete before moving ahead, the game can keep registering the player's inputs and sending them out while building a buffer of its own and incoming input packets. The hope is that processing the inputs with a small delay will provide the time for the next message exchange to be complete while the result of the previous one is being displayed to the players, keeping the game from stopping to wait. This can also alleviate the effects of jitter. The connection speed may fluctuate, causing some data exchanges to be completed faster than others. If the game processes the next frame as soon as it has a full set of all the players' inputs, the speed of the game will fluctuate too. Having a buffer allows the game to process frames at an even pace. (Fiedler, 2014.)

The usefulness of this technique has its limits. To prevent the game running without pause, the input delay must be at least equal to latency. So, if the latency is too great, the game will run out of buffered inputs and either be forced to stop and wait for more, or increase the input delay to cover the latency, making the game feel unresponsive. For certain genres, like strategies, it might still be tolerable, but faster-paced P2P games that require highly responsive controls, like shooters and fighting games, may have to opt for another netcode architecture instead of lockstep (Mattis, 2023a). Rollback P2P architecture, as seen in Figure 6, is a popular alternative for such cases.

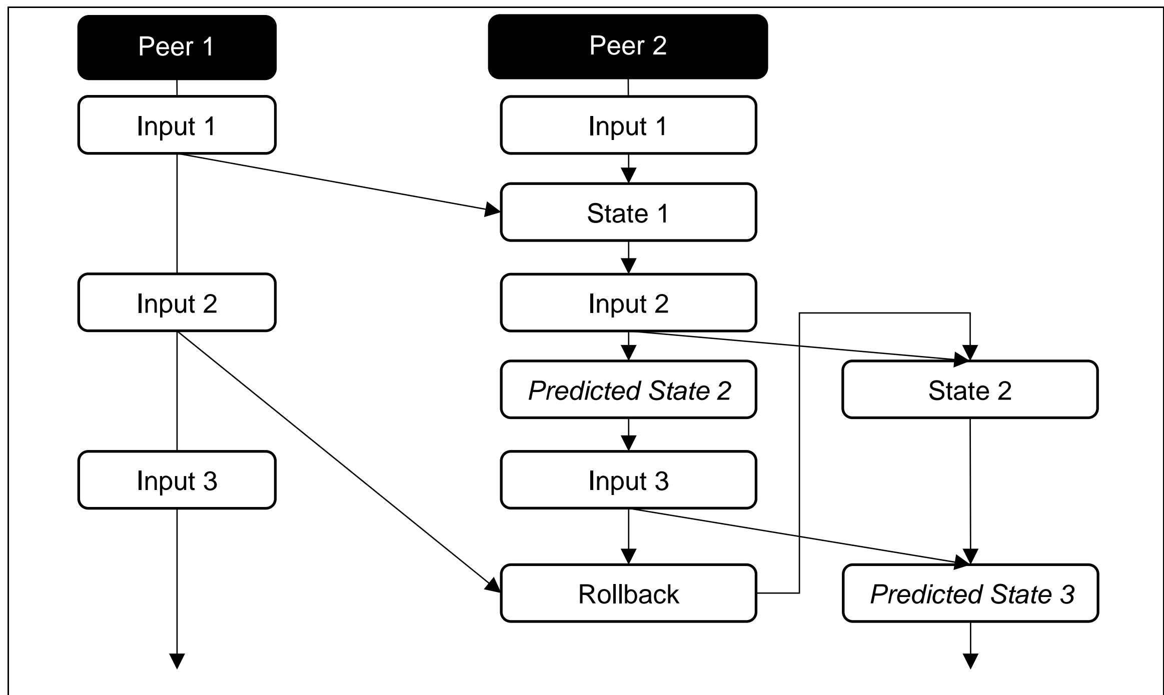

Figure 6. Rollback example

The idea of this model is that instead of waiting for the remote player's messages like in lockstep architecture, the game continues running even if no input was received from the other player. What's done in this case is called "client prediction" – the game simply assumes that the other player's input remains the same as the last received one (Mattis, 2023b). When a late packet finally arrives, the game "rolls back" to the corresponding frame and retroactively recalculates the following frames taking the newly received information into account. This way, although the predicted states can differ between players, the final result after all network packets come through is synchronized.

Though this prediction technique may seem rudimentary, it is actually surprisingly effective. Even during the most active phases of the game, the player would typically change their input only a few times per second. If, for example, it happens 5 times per second and there are 60 ticks in a second, that means the assumption that the input hasn't changed will be correct about 92% of the time. (Pusch, 2019.)

The unfortunate side effect of the rollback recalculation procedure is the possible appearance of various visual artifacts and glitches, like rubber banding,

teleportation, etc. (Gao, 2018). There are various techniques that can be used to mitigate this. For one, rollback can make use of input buffering in much the same way as lockstep architecture does, to reduce the effects of latency and necessity for game state recalculation. Another possible trick is input decay – meaning that when predicting the other player in the absence of expected network packets, the effect of the last known input gradually weakens. This helps in cases where, for example, the other player drastically changed movement vector during the lapse, making their predicted and real positions move in opposite directions, away from one another. (Mattis, 2023b.)

Rollback netcode is great for games that are very sensitive to input latency, fighting games in particular. However, it adds a lot of development complexity and requires serious optimization. The game needs to be able to recalculate several rollback frames in the time normally allotted for a single one, otherwise it might enter the so-called “spiral of death”, when the number of rollback frames builds up faster than they can be processed. (Mattis, 2023b.) Additionally, it necessitates the ability to reliably save and reload the complete game state, as well as certain other features. Generally, it is much harder to implement with an already existing game that hasn’t been made with rollback in mind. (Pusch, 2019.)

4 UNITY GAME ENGINE

Developing a game is a complicated task. There are a lot of components that need to be built – game logic, user interface, graphics visualization, audio playback, physics system, etc. Fortunately, game developers don’t have to create all of this from scratch anymore. There is little sense in reinventing the wheel, and instruments to solve a lot of common problems have already been made and are available for use.

Game engines are an indispensable tool for game developers. They are frameworks that streamline the development process, reduce the cost, complexity and time needed for the development of a game, and add a layer of abstraction for implementing often-needed mechanics (Halpern 2018, 1). Though

some studious still create their own in-house engines, there is a diverse selection of premade game engines available nowadays. Some are proprietary and some are freeware. Some are complex and highly customizable, others are simple and easily accessible. Some require knowledge of a programming language and others don’t. Some only handle one aspect of a game (graphics rendering engines, physics engines, etc.), others provide more complete solutions (Halpern 2018, 3).

Unity game engine, developed by Unity Technologies and first released in 2005, is without a doubt one of the most popular engines on the market today. It is user-friendly, versatile, supports both 2D and 3D, as well as a large variety of platforms – personal computer (PC), mobile phones, game consoles, web, etc. – and is widely used by professional and indie developers alike. Unity is especially popular with the latter, in no small part because it can be used free of charge, with some stipulations. It offers a range of licenses based on the developer’s revenue from their game. If the revenue is less than $100 000, the free license can be used at no cost, though it does not grant access to some extra features that paid tiers do. Thanks to Unity’s popularity, a vast and prolific community has formed around it that is actively producing countless tutorials, courses, extensions and assets. All of these factors make Unity very easy to pick up and start building games in. (Dealessandri, 2020.)

In this chapter, I will first provide a general overview of Unity game engine. After that, its current networking solution will be examined. Finally, I will discuss the services Unity Technologies offer for running online multiplayer games.

4.1 General overview of the engine

Unity game engine provides all the necessities for creating a full-fledged game. It can handle graphics, audio, physics, etc. The process of game development is done in the game editor, which consists of a number of windows and views that serve various purposes (Figure 7).

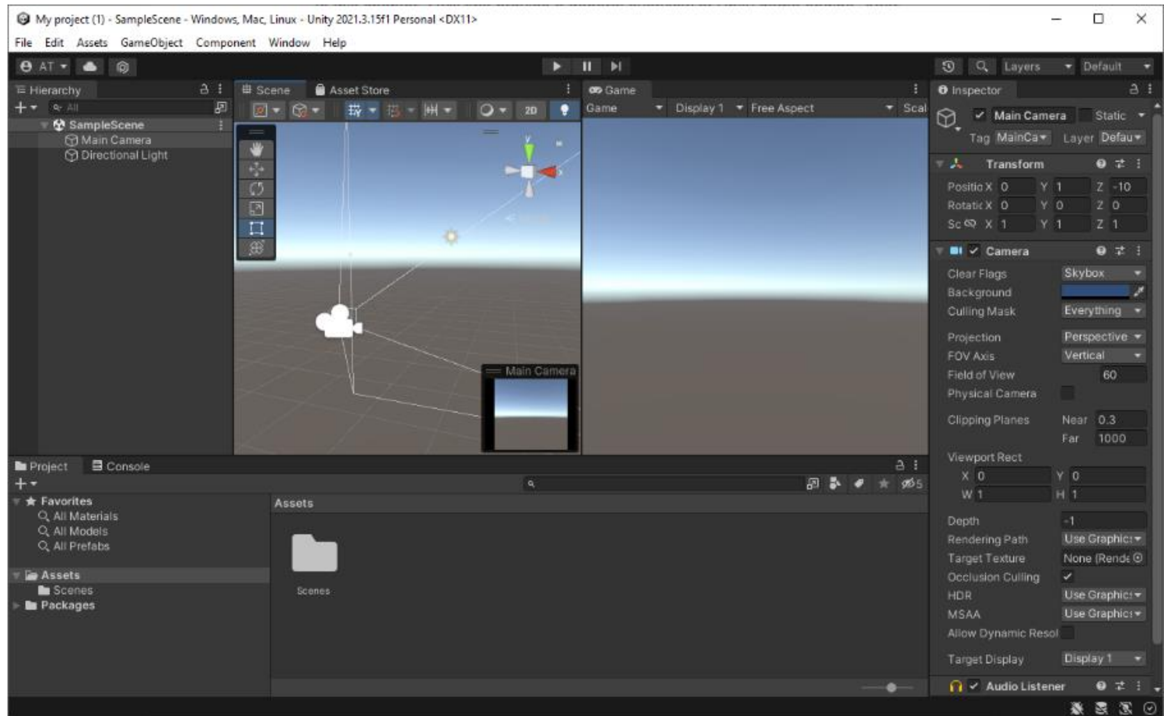

Figure 7. Unity window layout example

The layout can be freely rearranged to the user’s liking. Some windows can be added or removed, but a few key ones are typically always included. Among them are:

- Scene View – is used for editing the current scene and visualizing things that would be invisible for the player, such as colliders;
- Game View – represents the game as it would be seen by the player, is also used for running the game in the editor for testing purposes;
- Hierarchy Window – shows the list of game objects in the current scene;
- Project Window – shows the contents of the “Assets” folder which contains all the assets (sprites, sounds, scripts, prefabs, scenes, etc.) used in the game;
- Console View – displays the text output of the game, such as debug log entries and error messages;
- Inspector Window – shows all the components attached to the selected game object and their editable parameters.

Above these windows there’s the Tool Bar, which contains game flow controls (for starting and pausing the game) and Account Selector (for signing in with a Unity account), among other things. Below is the Status Bar which provides notifications about processes. (Halpern 2018, 18-21; Unity Technologies, 2023d.)

A game made in Unity consists of scenes, which are a type of asset that encompasses all or part of the game. Some simple games can be created in a single scene, while more complex games can be made with several scenes. For example, a game can have separate scenes for each level. (Unity Technologies, 2023c.)

The scenes contain GameObjects, which are the main building blocks of the game. Everything in a Unity game is a GameObject – from characters and items to special effects and user interface (UI) elements. A GameObject is a container for components, which provide it with functionality. (Unity Technologies, 2023b.) Unity includes many premade components such as renderers to add visuals to an object, or colliders to mark an object's boundaries for physics or hit detection purposes. GameObjects can also be grouped under another GameObject, which is called "parenting". "Children" objects' position, rotation and scale (together called object's "transform") in the world is relative to their "parent" object, i.e., any movement or rescaling of the parent object will also affect the children. This allows for engineering of more complex objects consisting of several semi-independent parts or can be used for better organization of GameObjects in a scene.

Aside from simply existing in a scene, GameObjects can be made into Prefabs – which are essentially GameObject templates. This is very useful for when several instances of the same object are expected to exist or have to be spawned dynamically during gameplay. Changes made to a Prefab can be applied to all of its instances, and it can be used in different scenes. (Halpern 2018, 63.)

In addition to preexisting components, the developers can create their own in the form of scripts. Unity natively supports C# programming language. Though all kinds of C# scripts can be created and used within the game, the scripts used as GameObject components must derive from the built-in MonoBehaviour abstract class. This grants access to a number of functions that will be automatically called by the engine on specific occasions. Some of the more important of such functions are:

- Update() – called each frame (frequency varies with the game's framerate and is not consistent);
- FixedUpdate() – called at fixed time intervals (consistent frequency, unlike Update());
- Start() – called before the first Update() call;
- Awake() – called when the script instance is loaded, before Start();
- OnDestroy() – called when the instance is destroyed.

The developer can write implementations for these and other functions in their script to define the behavior of the GameObject. It must be noted that Unity's preexisting components are also simply built-in C# classes, and often have their own properties and functions that can be called from other scripts. (Halpern 2018, 69-73, 312; Unity Technologies, 2023a.)

4.2 Netcode for Game Objects

Although Unity does not have network capabilities by default, there are numerous networking solutions available for the engine. Some are made by third-party developers (these include Mirror Networking, Photon Fusion, Fish-Net, and others), but there is one created by Unity Technologies themselves too. Unity's own networking framework has gone through several iterations at this point. First it was UNet/High-Level Application Programming Interface (HLAPI), then Mid-Level Application Programming Interface (MLAPI), which in 2022 was upgraded and renamed into Netcode for GameObjects (NGO). Unity Technologies have also recently been working on an alternative called Netcode for Entities, intended for their new Data-Oriented Technology Stack (DOTS), but this thesis will focus on Unity's GameObject-based development process with NGO.

NGO is a high-level networking library for Unity engine that abstracts networking logic, allowing the developer to focus on creating the game without having to build their own networking protocols and frameworks (Unity Technologies, n.d.a). It relies on the Unity Transport package, which is a low-level networking library for managing connections that uses User Datagram Protocol (UDP) and WebSockets (Unity Technologies, n.d.b). NGO uses client-server topology (with

either dedicated or listen server) and provides various features and tools for creating an online multiplayer game.

NetworkManager is the most important network-related component. It contains all of the network settings and serves as a hub for all netcode-related activity. Only one object with this component must exist in the game, and it is set to not be destroyed on scene load by default, so it will carry over from scene to scene. Since there is only one NetworkManager in the game, it employs Singleton programming pattern and can be addressed from any script directly as “NetworkManager.Singleton”. This component holds references to a number of netcode-related subsystems such as PrefabHandler, SceneManager, SpawnManager, NetworkTimeSystem, NetworkTickSystem and CustomMessagingManager, which can all be accessed via Singleton. The subsystems are only instantiated when NetworkManager is started with a special command in one of 3 possible modes: Client, Server or Host. Naturally, Client mode is used for clients and Server mode is used for the server. Host mode is used for a listen server and combines the functions of the other two modes. (Unity Technologies, n.d.f.)

Not all GameObjects in a game need to be aware of the network connection or interact with it. For example, the floor and other static objects that remain unchanged in gameplay will stay the same for all players and are usually not affected by any online features. On the other hand, there are certain things like health bar or other UI elements that are only visible to one player and don’t need to be synchronized across the clients. Even if they need some networked data (e.g., the player’s health), it can simply be passed to them by a network-aware object.

The GameObjects that do need to interact with the network connection should have the components to facilitate that. There are a few such components included with NGO, and game developers have the ability to program their own network-related scripts as well. However, all of these require that the GameObject also has a NetworkObject component.

NetworkObject is a component that enables other netcode-related components attached to a GameObject. If said GameObject has children with netcode-related components, they do not need to have NetworkObject components of their own – it is enough that the parent object has it. Among other things, this component takes care of identifying the GameObject for networking purposes and tracking ownership. (Unity Technologies, n.d.g.)

The Prefabs that have a NetworkObject and other network-related components attached to them and are expected to be added to the scene dynamically during gameplay must be registered in a special list inside NetworkManager (Unity Technologies, n.d.f.). Typically, when an object is created, Instantiate() function is used, but in an online multiplayer game it will only create the object on the local machine. To propagate the object to all clients and synchronize it, it must also be “spawned” with a special function of NetworkObject class. Spawning and despawning of NetworkObjects can only be done by the server. (Unity Technologies, n.d.i.)

All NetworkObjects have an “owner”, which can be either the server or one of the clients. This is specified when spawning the object by using one of the spawning functions. Ownership can also be changed or taken away later. The owner enjoys certain privileges regarding the control of the object, though the game developer can usually override them where necessary. (Unity Technologies, n.d.g.)

For the actual behavior of networked objects NGO provides several components, such as NetworkTransform, NetworkAnimator and NetworkRigidbody, intended for synchronizing the object’s transform, animation and physics respectively. In addition to these, game developers can create their own scripts using NetworkBehaviour abstract class. It is a network-enabled extension of MonoBehaviour class discussed in Chapter 4.1 and can be used the same way for writing custom C# scripts.

NetworkBehaviour adds some new functions, such as OnNetworkSpawn(), which is called when the GameObject is spawned on the network. An important detail is that this function is called at different times for dynamically spawned objects and in-scene placed objects. For the former, OnNetworkSpawn() is called after Awake() and Start(), but for the latter it's called after Awake(), but before Start(). (Unity Technologies, n.d.e.)

In addition to new functions, NetworkBehaviour grants use of NetworkVariables. These are containers for values of various types that will be automatically synchronized between clients when altered. NetworkVariables support a wide selection of data types, among them unmanaged C# primitive types (bool, int, float, etc.) and Unity built-in types (Vector2, Vector3, Quaternion, Color, etc.), as well as any enum types. They can also support any custom types that implement INetworkSerializable interface for data serialization. Notably, NetworkVariables do not support String type. Instead, Unity fixed-length string types can be used, such as FixedString32Bytes, FixedString64Bytes, etc. By default, a NetworkVariable's value can be read by all clients, but written only by the server. These permissions can be adjusted in the constructor's parameters when creating a NetworkVariable. For example, a NetworkVariable can be set to be writeable only by the owner of the NetworkObject that the NetworkBehaviour script is attached to, even if the owner is not the server. (Unity Technologies, n.d.h.)

Remote Procedure Calls (RPC) are another great tool provided by NetworkBehaviour class. They are a way to remotely tell another machine to execute certain code. There are two types of RPCs: Client RPCs and Server RPCs. They are declared like normal methods but must have a corresponding attribute before the declaration and suffix at the end of the name. RPCs can have parameters like any method, which will be passed to the target machine with the call. The parameters must be serializable data types, reference types can't be used. (Unity Technologies, n.d.l.)

Client RPCs must have the [ClientRpc] attribute and “ClientRpc” suffix. They can only be called by the server and are executed on the clients. Client RPCs can either be called on all clients (this is the default behavior) or only on certain ones, specified in parameters. (Unity Technologies, n.d.c.)

Server RPCs must have the [ServerRpc] attribute and “ServerRpc” suffix. Unlike Client RPCs, they can only be called by clients and are executed on the server. Server RPCs can usually be called only by the owner of the NetworkObject, but this can be overridden via the attribute, so as to allow non-owner clients call the RPC. (Unity Technologies, n.d.m.)

All RPCs are reliable by default. They are guaranteed to reach the target machine and be executed in the same order as they have been called in. However, the order of execution is guaranteed on a “per NetworkObject” basis only – it is possible that RPCs on different objects will be executed out of sync with each other. For non-essential RPCs reliability can be disabled by specifying the option in the attribute, to decrease the load on the network. (Unity Technologies, n.d.k.)

NGO has all the tools needed for building an online multiplayer game. However, it is not very suitable for large-scale games. There’s a limit to how big of a game it can realistically handle, how many players, objects, etc. due to bandwidth constraints and lacking certain features expected of larger games (e.g., “Area of Interest” – which means only sending data for the objects in a particular player’s vicinity, and not the entire world) (Lemay, 2022). Nevertheless, the development of NGO continues, and its performance can be expected to improve in the future. In the meantime, it is possible for developers to create their own workarounds for any issues they face or use a third-party solution together with or instead of NGO.

4.3 Unity Gaming Services

NGO is a handy tool for developing an online multiplayer game, but it doesn’t cover all the aspects of running such a game. Developers still require servers, a method of connecting players with each other over the internet, possibly

monetization avenues, etc. Naturally, it is possible to create custom solutions for such issues. However, that is not necessary, thanks to Unity Technologies' Unity Gaming Services (UGS).

UGS is a complex of services and tools that helps game developers to handle a multitude of challenges related to developer operations (diagnostics, version control), live operations (analytics, authentication, cloud, etc.), growth and monetization. Naturally, for online multiplayer games the most important part of UGS would be the Multiplayer Services. Among them are Lobby, Matchmaker, Game Server Hosting (Multiplay), Voice and Text Chat (Vivox) and Friends. As a matter of fact, NGO is considered a part of Multiplayer Services too. The names of these services are fairly self-explanatory. (Unity Technologies, n.d.j.) Not all of them are free, but most offer a limited, if rather generous free tier which should be enough for developing and testing purposes, or even running a small game with a limited player base until it gets traction and can be monetized (Unity Technologies, n.d.n).

For the purposes of this thesis, Relay and Lobby services are of particular interest. They present solutions to issues postulated in Chapters 2.2 and 2.3 respectively. Both of these require installation of a corresponding package for the game project and a Unity account to link the project to the services. UGS is accessed and controlled via a web interface called "dashboard" (Figure 8).

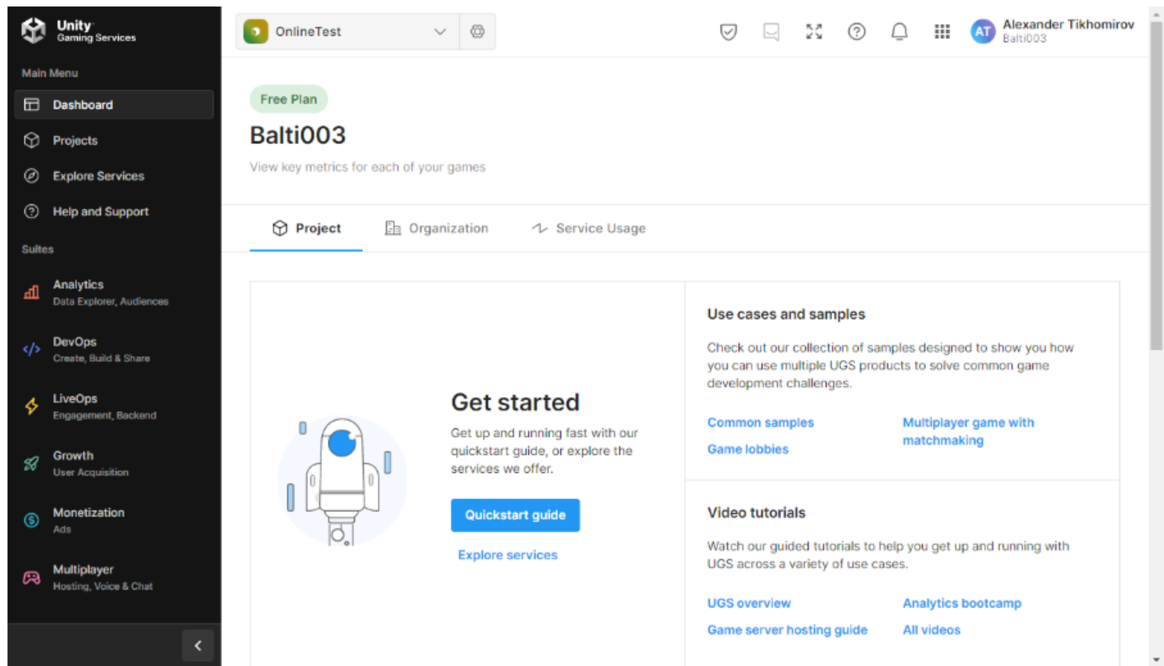

Figure 8.Unity Gaming Services dashboard

Dashboard is where the various services of UGS can be enabled and disabled for any projects tied to the developer's Unity account. It also provides access to statistics, configurations, setup guides and other useful features. After the services are enabled and the project linked, the scripts for using them must be written.

Before any services can be used, UnityServices singleton must be initialized using UnityServices.InitializeAsync() method. After that, all participants of the game – clients and the host – must authenticate. The Authentication service (which does not require separate setup procedure) offers several options for this. It is possible to authenticate using several existing platforms (Steam, Google, Facebook, Apple, etc.) or a custom id. It is also possible to authenticate anonymously, without any credentials. (Unity Technologies, n.d.o.)

After successful authentication, Lobby and Relay services can be used. While neither of them requires the other, the two work very well together to arrange a connection between players. Lobby service provides the framework for players to find each other, and Relay service allows them to connect via Unity's proxy relay server, avoiding issues with NAT and firewalls.

LobbyService class, included with the Lobby package, has a number of functions to work with lobbies – create them, join, leave, request a list of available lobbies from the master server, filter them, etc. Each lobby is represented with a Lobby C# object that contains a list of connected players and other assorted data. This object is not updated in real time or automatically on the local machine – updates need to be requested, and no more frequently than once per second. This is because to control the network traffic, the service limits the rate at which requests can be performed. The rate varies for different kinds of requests, and it should be taken into account. (Cardoso, 2022b.)

Lobbies have a limited lifespan. A lobby's owner must periodically send a "heartbeat" signal to the server to keep the lobby active. A lobby that has not received heartbeats or updates for a certain period of time (normally 30 seconds) is marked as inactive and will not show up in query results. After being inactive for an hour, the lobby is considered expired and will eventually be deleted. (Unity Technologies, n.d.d.)

Similarly, RelayService class, included with the Lobby package, handles relay connections. First, the host must make an allocation request, to allocate the resources on Unity's relay server. After this procedure succeeds, the host will receive a join code, which must be shared with the clients so they can connect to the relay too. This is where Lobby service comes in handy – it is possible to include the join code in the lobby's data, so all joined players can see it. (Cardoso, 2022c; Unity Technologies, n.d.o.)

One important detail is that most of the methods related to Lobby and Relay services require data exchange over the internet, which takes time. That's why they should be executed asynchronously, to prevent the game from freezing while it's waiting for a server's response. Additionally, these methods can fail for a variety of reasons (request rate limit or maximum number of players exceeded, bad connection, etc.), so the corresponding exceptions (LobbyServiceException

and RelayServiceException) must be properly handled to prevent crashes. (Cardoso, 2022b; Cardoso, 2022c.)

Using UGS, it is very easy to connect remote players. There is no need to implement a custom lobby system or bother with NAT-related troubles. Moreover, the burden of organizing the server infrastructure is taken off the developers' shoulders.

5 IMPLEMENTATION

This chapter will cover a practical demonstration of Unity's online multiplayer capabilities. For this purpose, an implementation of a simple game prototype will be described. First, I will explain the concept of the game and then the various technical aspects.

5.1 Game concept

The game I aim to create is a competitive combat game. The general idea of the gameplay is inspired by games like Shovel Knight Showdown (Figure 9) and Awesomenauts (Figure 10). It should be noted that the former is technically only a local multiplayer game, though it supports Steam's Remote Play Together feature.

Figure 9. Shovel Knight Showdown gameplay (Shovel Knight Showdown 2019)

Figure 10. Awesomenauts gameplay (Awesomenauts 2012)

Although these two are quite different games, they share multiple features. Both are multiplayer games where each player controls a character in a 2D platformer-style environment to fight against other teams or individual players and achieve a certain winning condition – either taking out all the enemies, destroying their base, or something else. Also, both have a cast of playable characters the players can choose from, each of whom has different abilities.

This is essentially what the finished game is intended to be – a 2D platformer-style fighting game with character selection. However, making a full game with all the implied features is beyond the scope of this thesis project. As such, only a proof-of-concept prototype will be created, which can be later developed further into a complete product for release.

To set the scope of the project, it is necessary to decide which key features must be implemented. The required features for a functional demo are the following:

- the players should be able to find online matches with a lobby system;
- the players should be able to connect through a relay;
- the players can choose a team to join;
- each player controls a simple character that has a certain number of hit points (HP) and can run, jump and shoot;
- the bullets can hit and hurt players from other teams;
- the game ends when all but one team are eliminated.

This should be enough for a working prototype that can serve as the basis for a full future game. The following chapters will describe various aspects of its creation.

5.2 Starting the project

The first step of making a game using Unity engine is to create a Unity project. This is done in a separate application from the engine itself. The application is called Unity Hub (Figure 11).

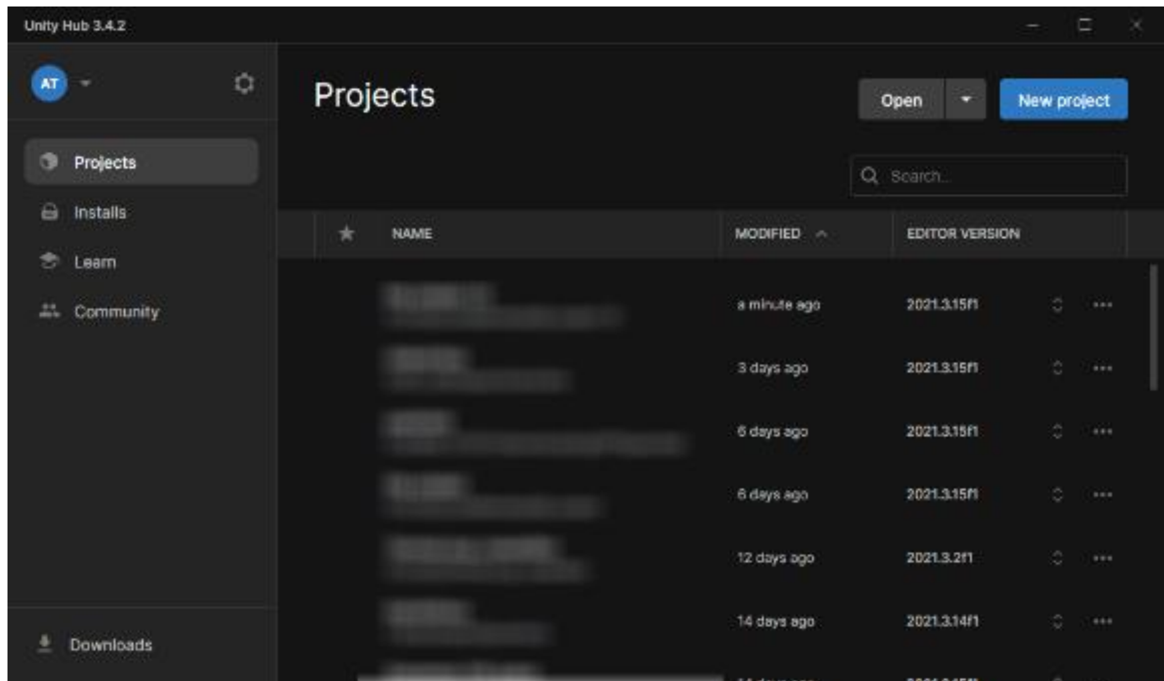

Figure 11. Unity Hub

Unity Hub is a tool that helps developers manage their Unity projects and installed versions of the engine (since different projects can use different versions), download updates, etc. When creating a new project, a few options need to be set: project name, location and a template to use. Unity provides several templates, most of which are empty projects preconfigured for a certain type of game. For my game, I chose “2D (URP)” template. Universal Render Pipeline (URP) makes it easier for developers to optimize the graphics for multiple platforms, so choosing this option will make the project more flexible.

When the project is created, the engine window will open. We can immediately install the packages that will be needed. Package Manager window can be opened through the menu above the Tool Bar (Window > Package Manager). Installation process is as simple as finding the required package in Unity Registry via a search bar and clicking “Install” button. The packages needed for the project are:

- Netcode for GameObjects – to implement the online functionality;
- Lobby – for the lobby system;
- Relay – to connect the players through the relay server;
- Input System – a useful package for processing player inputs.

After all the packages are successfully installed, the development proper can begin.

5.3 Main menu

Almost every video game in existence starts with a menu. That's where the player can choose various available options for starting the game, change settings or quit. In our case, the game will start with a menu where the player can create or join a lobby in order to play with other people. Figure 12 demonstrates the logical structure of the menu.

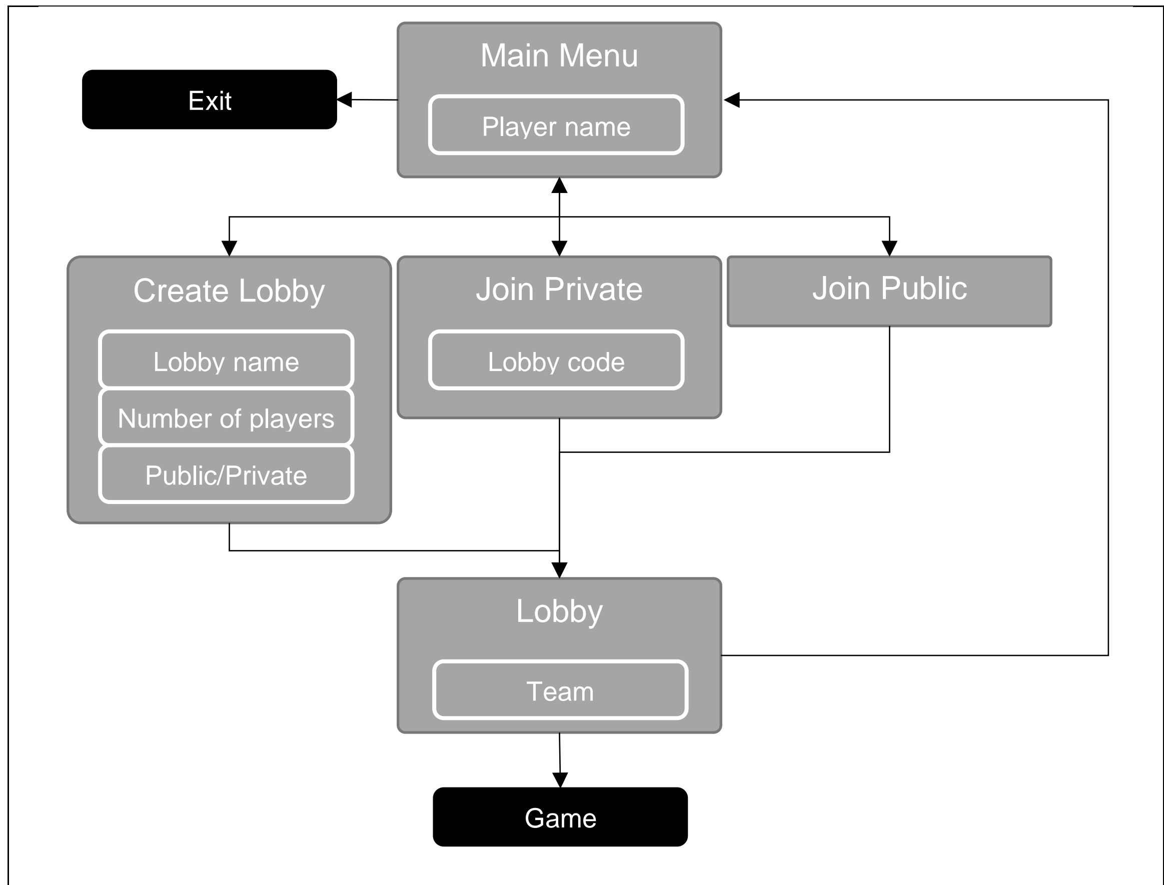

Figure 12. Menu structure

The menu consists of several submenus the player switches between. The first one is the Main Menu where the player can set their name and choose one of several options:

- Create lobby – to create a new lobby;
- Join with code – to join a private lobby with join code;
- Search lobbies – to see the list of public lobbies that can be joined;
- Exit game – to close the game.

Choosing one of the first three options will take the player to one of the intermediate submenus.

In the Create Lobby submenu, the player can set the name of the lobby, number of players, and whether the lobby should be public or private. After that, the lobby can be created, and the player will be taken to the Lobby submenu. In the Join Private submenu, the player must enter a join code in order to connect to a private lobby. In Join Public submenu the player is presented with a list of currently active public lobbies and can freely choose which one to join. All three of these intermediate submenus also give the player an option to return to the Main Menu.

In the Lobby submenu, the player can choose a team to join and see which team the other players have chosen. After all players connect, select a team and finalize their decision by pressing the “Ready” button, the game can begin. All the players will be taken to a different scene where the match happens. While still in the lobby, the player can choose to leave instead, which will take them back to the Main Menu.

To implement this, a Canvas (a type of GameObject for housing UI elements) has been created with nested empty GameObjects to contain each submenu (Figure 13).

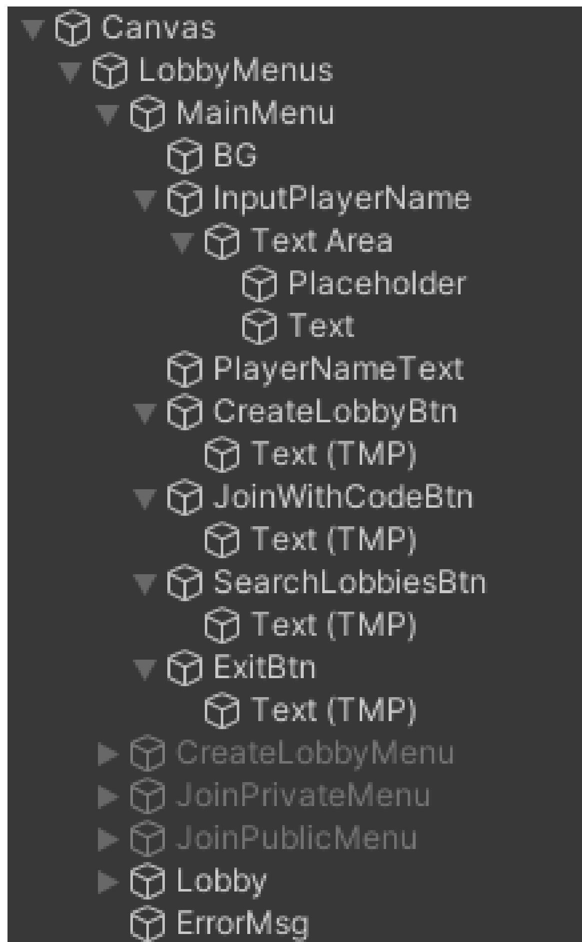

Figure 13. Menu structure in Unity, with Main Menu's object hierarchy fully unfolded

Each empty object representing a submenu has all the elements of this submenu (buttons, text input fields, etc.) as children. Additionally, it has a custom script providing the submenu's logic attached as a component. To facilitate switching between menus, I have created IMenuSwitchInterface interface that declares three methods:

- Show() – to activate the submenu-containing GameObject and make it visible;
- Hide() – to deactivate it and make it invisible;
- SwitchToMenu(IMenuSwitchInterface targetMenu) – to implement the switching from the current submenu to one specified in the parameter.

These methods are defined in the SubMenu abstract class (Figure 14).

```
3 references
public void Show()
{
    gameObject.SetActive(true);
    currentSubMenu = this;
}

2 references
public void Hide()
{
    gameObject.SetActive(false);
}

8 references
public void SwitchToMenu(IMenuSwitchInterface targetMenu)
{
    targetMenu.Show();
    this.Hide();
}
```

Figure 14. IMenuSwitchInterface implementation inside SubMenu class.

All but one submenu-controlling scripts derive from the SubMenu class. The script for controlling Lobby submenu is the only one that needs network functionality, so it is a NetworkBehaviour and not MonoBehaviour script like the others. As such, in order to be compatible with the menu switching system, it implements the IMenuSwitchInterface directly.

Each submenu-controlling script contains references to the active elements of its corresponding submenu and other required objects (Figure 15).

```
Unity Script (1 asset reference) | 0 references
public class UI_CreateOrJoinLobby : SubMenu
{

    //============== INPUT FIELD ===============

    [Header("Input field")]
    [SerializeField] private TMP_InputField nameInputField;

    //============== BUTTONS ===============

    [Header("Buttons")]
    [SerializeField] private Button createLobbyBtn;
    [SerializeField] private Button joinWithCodeLobbyBtn;
    [SerializeField] private Button searchLobbiesBtn;
    [SerializeField] private Button exitGameBtn;

    //============== SUBMENUS ===============

    [Header("Submenus")]
    [SerializeField] private SubMenu enterCodeMenu;
    [SerializeField] private SubMenu searchLobbyMenu;
    [SerializeField] private SubMenu createLobbyMenu;

    //============== LOBBY ===============

    [Header("Lobby")]
    [SerializeField] private LobbyConnect lobbyConnect;
```

Figure 15. Object references in the main menu script

[SerializeField] is an attribute that allows the variables’ initial values to be assigned through Unity’s Inspector Window. With references to all the objects set, their functionality can be described in the rest of the script. As an example, the rest of the main menu’s script can be seen in Figure 16.

```
Unity Message | 0 references
private void Awake()
{
    //Set input field to player name from playerprefs
    //If no name in playerprefs, set to "Anonimous"
    string currentName = PlayerPrefs.GetString("PlayerName", "Anonimous");
    SetName(currentName);
    nameInputField.text = PlayerPrefs.GetString("PlayerName");

    nameInputField.onValueChanged.AddListener(SetName);

    createLobbyBtn.onClick.AddListener(() => { SwitchToMenu(createLobbyMenu); });
    joinWithCodeLobbyBtn.onClick.AddListener(() => { SwitchToMenu(enterCodeMenu); });
    searchLobbiesBtn.onClick.AddListener(() => { SwitchToMenu(searchLobbyMenu); });
    exitGameBtn.onClick.AddListener(() => { Application.Quit(); });
}

2 references
private void SetName(string newName)
{
    PlayerPrefs.SetString("PlayerName", newName);
    lobbyConnect.PlayerName = newName;
    Debug.Log("New name = " + newName);
}
```

Figure 16. Main menu's methods

In the Awake() method, first I check if there is already a player name saved in the player preferences on the computer. If there's not, it's set to "Anonimous". Then the player name input field text is set to the current name, and a listener to update the name is added to the input field's onValueChanged event, which will trigger every time the player makes changes in the input field. Similarly, the listeners are added to all of the buttons' onClick events to give them functionality – either switching to a different menu or exiting the game. The only other method in this script is SetName(string newName), which updates the player name in preferences and in LobbyConnect.cs script, which will be explained in the next chapter. Figure 17 shows the resulting menu.

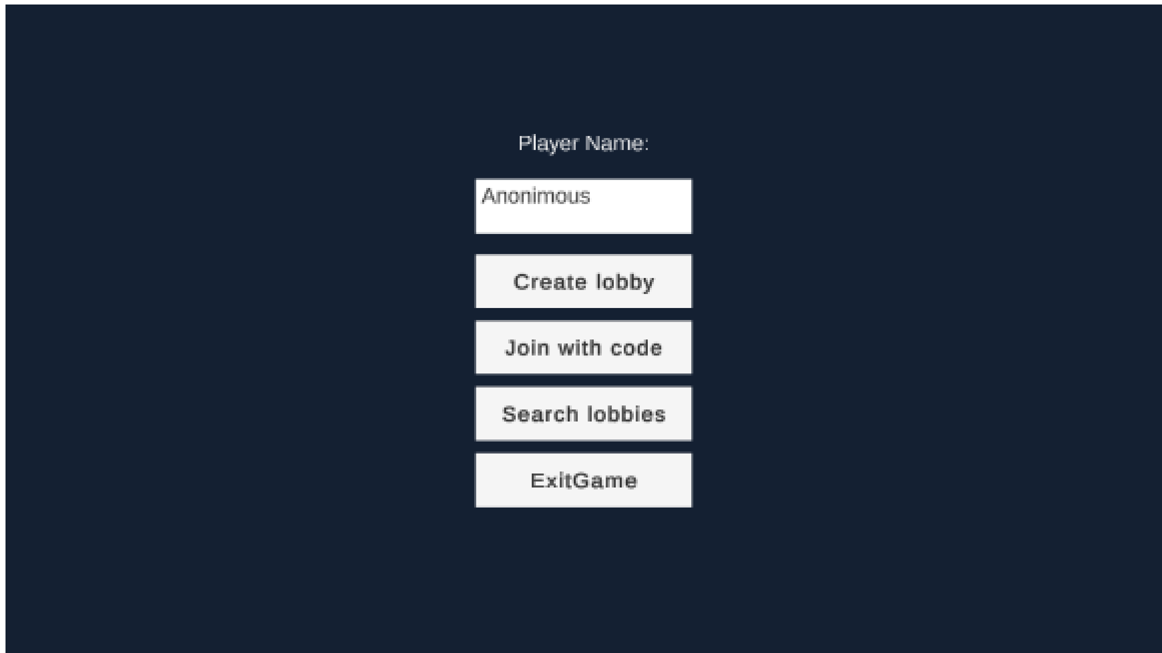

Figure 17. Main Menu

The other submenus follow similar logic, except for the fact that they interact with lobby-related scripts to arrange the connection between players. Also, Join Public and Lobby submenus represent information that is periodically updated (list of available public lobbies and the players' team selections, respectively). This is done by creating a template GameObject representing a datapoint (lobby or player) and creating a list of edited clones of it. Every time the information is updated, the list is purged and refilled anew to accurately represent the up-to-date situation to the player.

5.4 Lobby and relay

Arranging the connection between players will involve several scripts working together:

- LobbyConnect.cs will handle interactions with the Lobby and Relay services;
- PlayerDataManager.cs will keep track of game participants;
- UI_LobbiesList.cs and UI_LobbyListEntry.cs will control the Join Public submenu that displays a list of available public lobbies;
- UI_Lobby.cs, UI_Lobby_TeamListEntry.cs and UI_Lobby_PlayerListEntry.cs will control the Lobby submenu where players select teams.

The former two of these scripts are attached to otherwise empty GameObjects and are set to persist between scenes, and are accessible from other scripts by a static Instance property, e.g. “LobbyConnect.Instance”. The latter two are set up in a similar way to the main menu script described in the previous chapter, except UI_Lobby.cs is attached to an empty object separate from the Lobby submenu parent object, to avoid potential issues with NetworkObject synchronization causing the submenu to change scale upon connection.

On startup LobbyConnect.cs initiates Unity services and authenticates the player. For simplicity’s sake, I opted to use anonymous authentication that does not require setting up an account or any login credentials. In the Update() the script runs two methods – one that handles heartbeat, and another for updating the joined lobby. Both methods check if the player is currently connected to a lobby and run a timer to perform their respective tasks at required intervals – once every 15 seconds for the heartbeat, and once every 1.5 seconds for the lobby update. Aside from this, the script contains a number of methods for various other UGS-related activities, such as:

- CreateLobby() – to create a new lobby;
- GetLobbies() – to query active public lobbies;
- JoinLobbyById(string lobbyId) – to join a lobby by its ID (used for joining public lobbies);
- JoinLobbyByCode(string lobbyCode) – to join lobby by its join code (used for joining private lobbies);
- CreateRelay(int numberOfPlayersToConnect) – to create a new relay allocation;
- JoinRelay(string joinCode) – to join an existing relay allcation;
- An assortment of auxiliary methods.

Figure 18 demonstrates the CreateLobby() method as an example.

```
public async void CreateLobby()
{
    try
    {
        string lobbyName = LobbyName;
        int maxPlayers = NumberOfPlayersPerTeam * NumberOfTeams;

        joinRelayCode = await CreateRelay(maxPlayers - 1);
        if (joinRelayCode != "0")
        {
            CreateLobbyOptions clo = new CreateLobbyOptions
            {
                IsPrivate = LobbyIsPrivate,
                Player = GetPlayer(),
                Data = new Dictionary<string, DataObject>
            {
                { "NumberOfTeams", new DataObject(
                  DataObject.VisibilityOptions.Public,
                  NumberOfTeams.ToString())},
                { "PlayersPerTeam", new DataObject(
                  DataObject.VisibilityOptions.Public,
                  NumberOfPlayersPerTeam.ToString()) },
                { "RelayCode", new DataObject(
                  DataObject.VisibilityOptions.Member,
                  joinRelayCode) }
            }

            };
            Lobby lobby = await LobbyService.Instance.CreateLobbyAsync(
                lobbyName, maxPlayers, clo);
            joinedLobby = lobby;

            PlayerDataManager.Instance.StartHost();

            OnLobbyJoined?.Invoke(this, joinedLobby);
        }
    } catch (LobbyServiceException lse)
    {
        Debug.Log(lse);
        OnLobbyServiceError?.Invoke(this, lse);
    }
}
```

Figure 18. CreateLobby() method

The method uses a number of the script’s fields that are set through the menus, such as LobbyName, NumberOfPlayerPerTeam, NumberOfTeams, LobbyIsPrivate, etc. After calling CreateRelay() method (implemented separately) and waiting for it to return the created relay’s join code, CreateLobbyOptions object is created, which contains all the necessary information for creating a lobby, including player’s data (fetched by GetPlayer() method) and relay’s join

code (so the other players that join the lobby can also join the relay). Notably, by using DataObject.VisibilityOptions.Member it is ensured that only lobby members can see the relay’s join code. After receiving a Lobby object as a response from the Lobby service, it is saved in the joinedLobby field, and a call is made to PlayerDataManager.cs to start the host. The possibility of a LobbyServiceException error is handled by saving the error in the debug log and triggering OnLobbyServiceError event. Another script listens to this event to display the error to the player.

The main task of the PlayerDataManager.cs script is to maintain a list of connected players with some relevant information. Technically, this could be done within the Lobby object, but that’s a more cumbersome and less reliable solution, since accessing the data would be more complicated and neither clients nor host would be able to keep the list updated in real time due to query rate limit. It would also be much harder to avoid conflicts, since only players themselves can modify their own data in the Lobby object, which makes it unsuitable for handling certain operations that may require server validation. Ergo, doing it with a NetworkList, which functions similarly to NetworkVariables described in Chapter 4.2, is a much better way. A custom struct called PlayerData will represent a single player in the list (Figure 19).

```
public struct PlayerData : INetworkSerializable, IEquatable<PlayerData>
{
    public FixedString64Bytes playerName;
    public ulong clientId;
    public int teamId;
    public bool ready;
    public Color color;

    0 references
    public bool Equals(PlayerData other)
    {
        return
            playerName == other.playerName &&
            clientId == other.clientId &&
            teamId == other.teamId &&
            ready == other.ready &&
            color == other.color;
    }

    0 references
    public void NetworkSerialize<T>(BufferSerializer<T> serializer) where T : IReaderWriter
    {
        serializer.SerializeValue(ref playerName);
        serializer.SerializeValue(ref clientId);
        serializer.SerializeValue(ref teamId);
        serializer.SerializeValue(ref ready);
        serializer.SerializeValue(ref color);
    }
}
```

Figure 19. PlayerData struct

The struct includes the player's name (as a FixedString64Bytes variable), client ID (to identify the player on the network for things like Client RPC addressing), team ID and color, and whether or not the player is ready to begin the match. It also implements INetworkSerializable and IEquitable interfaces. PlayerDataManager.cs constructs such a struct for each player and saves them in a NetworkList. UI_Lobby.cs handles the visualization of this list for players before the match starts (Figure 20).

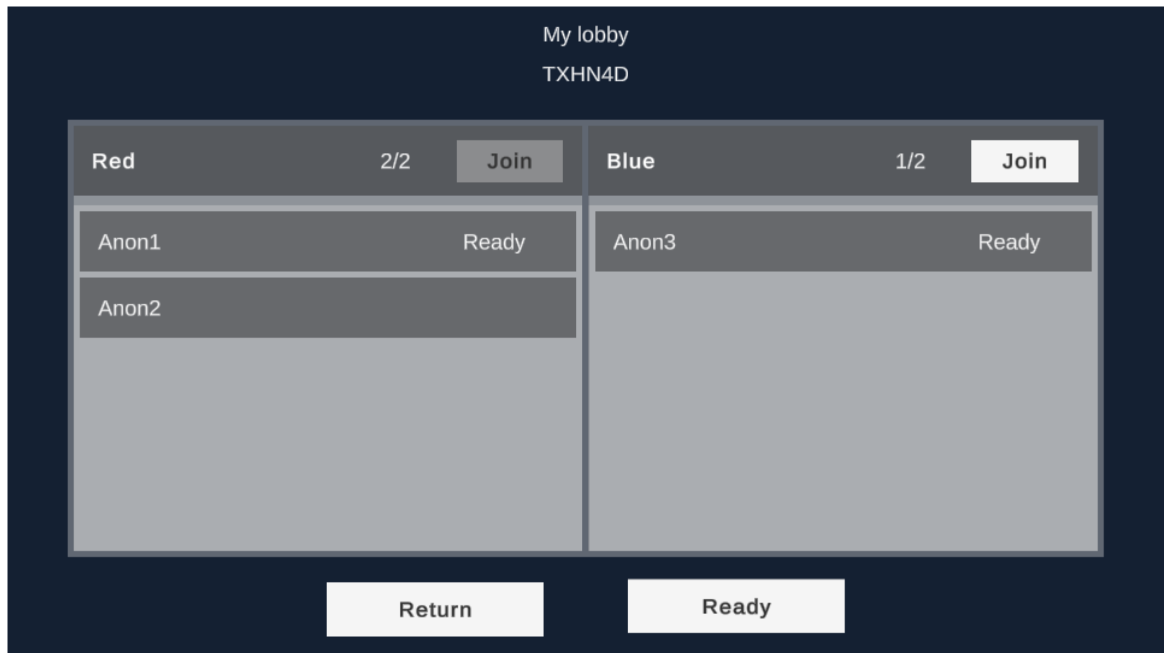

Figure 20. Lobby submenu

Figure 20 shows the lobby screen from the perspective of player named “Anon2”. Anon2 already chose to join Red team but haven’t finalized the choice by clicking the “Ready” button. There’s still an open slot in Blue team, so Anon2 can change their mind and switch teams. As such, “Join” button is inactive for Red team (since a player can’t join the team they’re already in), but active for Blue team. “Ready” button is also active until the choice is successfully finalized. The buttons work by asking PlayerDataManager.cs to call Server RPCs to make required edits to the corresponding PlayerData object in the NetworkList. Since all the changes are applied server-side, any possible conflicts are avoided. Even if both Anon1 and Anon2 decide to join the Blue team at the same time, the server will only let one of them do it – whoever’s Server RPC reaches the server first. The updated NetworkList will then be propagated to all clients. UI_Lobby listens to the list’s OnListChanged event and calls a method to update the submenu in accordance with the up-to-date situation. When all players have selected a team and finalized their choice, a call to NetworkManager’s SceneManager subsystem is made to load the scene where the gameplay happens.

5.5 Player Prefab

At the start of the match a character is spawned dynamically for each player to control. The character Prefab is built from several nested GameObjects. Figure 21 depicts the structure of the Prefab.

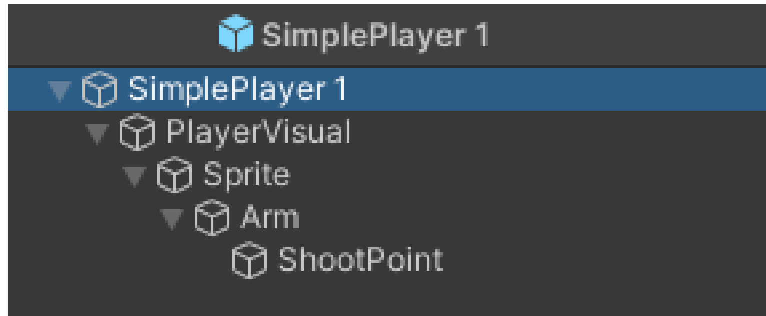

Figure 21. Player Prefab structure

It is generally a good idea to separate game logic from visuals. For this reason, the latter is handled by the nested PlayerVisual object, while the root object deals with the former. The root object has the following components attached:

- BoxCollider2D – for collisions;
- PlayerChar – custom script to handle behavior;
- PlayerHealth – custom script to handle health;
- NetworkObject – to facilitate networking.

Some of the children objects have other components attached to them, while others are empty objects. Together they provide the player character with all the required functionality.

5.5.1 Input System

Though not technically done inside the player Prefab, processing of raw inputs is closely related to it and warrants a brief explanation here. For handling inputs, I opted to use Unity's Input System package. This package allows developers to create so-called Action Maps (Figure 22).

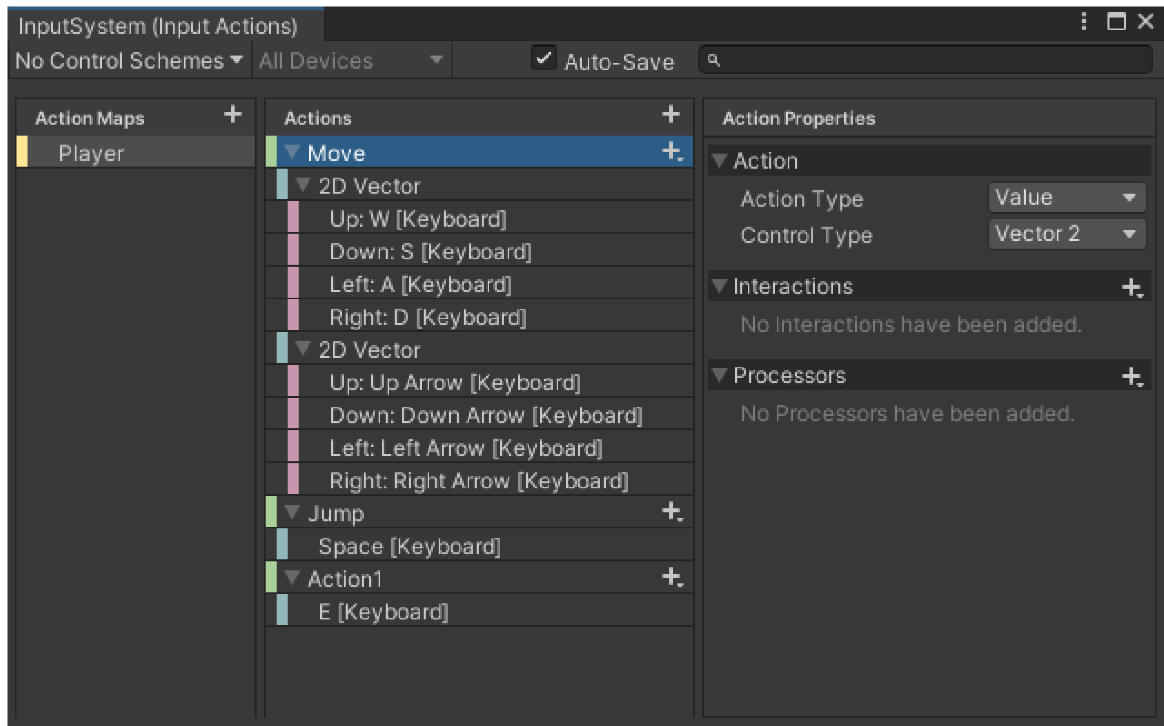

Figure 22. Action Map in Input System

Configuring various input Actions in the Action Map will make the game trigger specific events when the inputs are performed by the player. To provide an additional level of abstraction, an empty object with attached InputManager.cs script is placed in the scene, which serves as an intermediary between Input System and PlayerChar.cs. If the way the game accepts inputs needs to be reworked in the future, only InputManager.cs will have to be edited. InputManager.cs has a static Instance property which all player objects listen to, but only the player object owned by the local player actually reacts to.

5.5.2 Player health

PlayerHealth.cs script keeps the player’s current HP in a NetworkVariable, which ensures that its value stays synchronized between all clients and can only be changed by the server. This script has a public method ReceiveDamage(int dmg) (Figure 23).

```
public void ReceiveDamage(int dmg)
{
    if (IsServer)
    {
        currentHealth.Value = Mathf.Max(0, currentHealth.Value - dmg);
    }
}
```

Figure 23. ReceiveDamage method

ReceiveDamage() method reduces the current HP by the amount given as a parameter, but only if it is executed on the server. It also ensures that HP does not go below zero. Any successful change to the value of currentHealth NetworkVariable activates OnHealthChanged() method (Figure 24).

```
private void OnHealthChanged(int previousValue, int newValue)
{
    OnHealthChange?.Invoke(this, HealthRatio());
    if (newValue == 0) {
        OnHealthZero?.Invoke(this, ownerID);
    }
}
```

Figure 24. OnHealthChanged method

OnHealthChanged() method invokes OnHealthChange event and, if HP reached zero, OnHealthZero event. The former is used by the health bar to represent the local player's new HP. The latter is used to signal player character's death, show the game over screen to the player who just lost and check for team victory conditions.

5.5.3 Player visuals

Character visualization is handled by the child objects of player Prefab. For this purpose, I have created a set of pixel sprites (Figure 25).

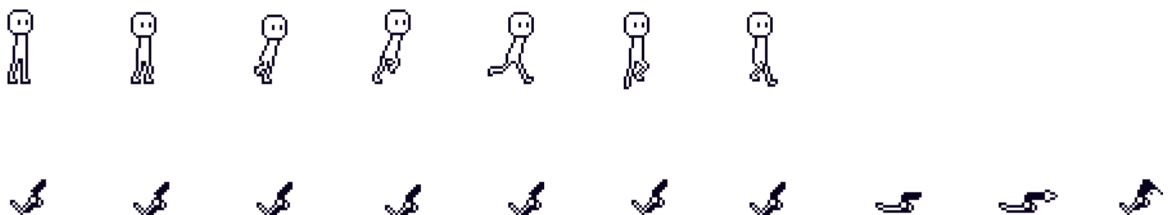

Figure 25. Character sprites

The playable character is a simple black and white figure that holds a gun. At the beginning of the match, the white color is changed to the color of the corresponding player's team. The body and the arms of the character are represented by separate sprites tied to separate GameObjects – Sprite and Arm respectively. This is done so that attack animation can play independently from movement animations – it does not matter for the shooting animation if the player is standing, running or jumping.

Unity provides the Animator component where all the animations can be arranged with conditional transitions. However, I have found it much easier to simply control the Animator through a script. This is done with a custom script called CharAnimControl.cs attached to the Sprite GameObject (Figure 26).

```
1 reference
private void PlayerChar_OnJump(object sender, System.EventArgs e)
{
    animator.Play("blanky_jump");
}

1 reference
private void PlayerChar_OnFall(object sender, System.EventArgs e)
{
    animator.Play("blanky_fall");
}

1 reference
private void PlayerChar_OnWalking(object sender, System.EventArgs e)
{
    animator.Play("blanky_walk");
}

1 reference
private void PlayerChar_OnIdle(object sender, System.EventArgs e)
{
    animator.Play("Idle");
}

1 reference
private void PlayerChar_OnAttackPerformed(object sender, System.EventArgs e)
{
    arm.SetTrigger("shoot");
}
```

Figure 26. CharAnimControl Animator-controlling methods

CharAnimControl.cs listens to the events triggered by the PlayerChar.cs script depending on the actions performed by the player character. Walking, jumping,

falling and idling cause CharAnimControl.cs to command Sprite object's Animator component to switch to an appropriate animation and play it continuously until the next switch happens. Attacking causes CharAnimControl.cs to set the "shoot" trigger on the Arm object's Animator component, which makes the shooting animation play once. The Arm GameObject also has the ShootPoint GameObject as its child (Figure 27).

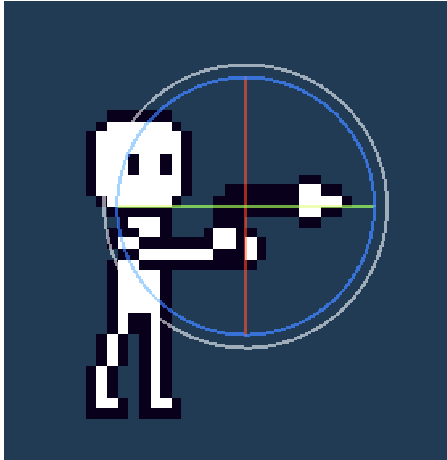

Figure 27. ShootPoint's location on the assembled player sprite

ShootPoint is an empty object. Its only purpose is to mark the location where bullets should appear when the player attacks. The PlayerChar.cs script holds a reference to it in order to retrieve ShootPoint's coordinates for bullet instantiation.

PlayerVisual is another empty object that serves as a simple container for all visual-related GameObjects. Its purpose is to facilitate character turning around. When the player turns, PlayerVisual is flipped horizontally, and all its children objects with it. Although similar effect can be achieved using SpriteRenderer component's flip functions, rotating the PlayerVisual GameObject keeps the Sprite and Arm objects' sprite positions perfectly in-sync with each other, and appropriately changes the location of the ShootPoint object. Additionally, it makes it easier to retrieve the direction for bullet movement.

5.5.4 Player behavior

PlayerChar.cs script is the most important of the player Prefab's components. Like PlayerHealth.cs it derives from NetworkBehaviour in order perform the network synchronization of characters. It also controls the player character behavior in general.

Since NGO uses a client-server topology, at first, I tried to implement traditional netcode with client-side prediction and entity interpolation. I utilized Unity's NetworkTransform component for character position synchronization, since it does entity interpolation, and NetworkAnimator for synchronizing animation. However, this proved to be a suboptimal solution for the type of game I am making. Since the player's movement can change rapidly and interpolation simply moves a character between positions taken at certain intervals, this made the remote players' movements floaty and led to unfortunate visual glitches like character passing through corners instead of properly going around them. After some experimentation, I chose the input sharing model for synchronizing players instead.

Input sharing requires the game world to be deterministic. Achieving true determinism in a game is a highly complicated matter due to a multitude of factors. For example, even the same exact calculation with floating point numbers can give slightly different results on different machines because of differences in hardware (Fiedler, 2010). Although the difference might be insignificant, using the resulting number for new calculations can lead to the difference accumulating over time and subsequent desynchronization. As such, achieving perfect determinism is beyond the scope of this thesis. Instead, the goal is to create a game that is simply "deterministic enough", with some fail-safe measures for desynchronization.

Usually, character-controlling scripts in Unity are written in such a way that most actions (e.g., movement) are performed in Update() method and rely on calling Time.deltaTime to scale any changes proportionally to time passed since last Update() call. Since Time.deltaTime can vary wildly even on the same machine, it

cannot be relied upon. All actions must be performed at fixed intervals. It is possible to create a custom solution for consistently timed updates, but for the purposes of prototyping I decided to simply use FixedUpdate() method that runs at fixed Time.fixedDeltaTime intervals.

PlayerChar.cs interacts with InputManager.cs to save the information about player’s inputs in a list. The information is saved in the form of InputEvent structs (Figure 28).

```
10 references
public enum InputEventType
{
    Move,
    JumpPress,
    JumpCancel,
    AttackPress
}

16 references
public struct InputEvent : INetworkSerializable
{
    public InputEventType eventType;
    public Vector3 inputVector;

    4 references
    public InputEvent (InputEventType iet, Vector3 iv)
    {
        this.eventType = iet;
        this.inputVector = iv;
    }

    0 references
    public void NetworkSerialize<T>(BufferSerializer<T> serializer) where T : IReaderWriter
    {
        serializer.SerializeValue(ref eventType);
        serializer.SerializeValue(ref inputVector);
    }
}
```

Figure 28. InputEventType enum and InputEvent struct

InputEvent is a simple struct type which only contains an InputEventType enum field to identify what kind of input event took place, and a Vector3 for movement inputs. For the momentary input events like JumpPress, JumpCancel and AttackPress, PlayerChar.cs listens to the events triggered by InputManager.cs. When FixedUpdate() is called, PlayerChar.cs queries InputManager.cs for the current movement input (e.g., arrow keys), and saves it in a list with whatever momentary inputs were saved there since the last FixedUpdate() call. With the

list complete, the local client processes the inputs and sends the list to the server together with the resulted position and velocity, via a Server RPC. The server, upon receiving this data, also processes the inputs and double checks the client's result. If there is significant deviation, the server sends its result back to the client to correct it. After that, the server distributes the input list to all other clients with a Client RPC.

In lieu of classical entity interpolation, the other clients save input lists distributed from the server in another list that serves as a buffer, and process one of them each FixedUpdate(). If for some reason one of the remote players' characters falls behind too much – e.g., if due to network issues it doesn't receive any inputs from the server, and then receives many at once – causing the buffer to accumulate a lot of entries, the speed they're processed at will be temporarily doubled to make that character catch up with the simulation.

The player behavior is defined by states. There are several states the player can be in, identified by CharState enum (Figure 29).

```
33 references
public enum CharState
{
    Walking,
    Idle,
    Jump,
    Fall
}
```

Figure 29. CharState enum

The states are:

- Walking – the character is moving horizontally on solid ground;
- Idle – the character is standing on solid ground without moving;
- Jump – the character jumped and is moving vertically up;
- Fall – the character has no solid ground under it and is moving vertically down.

Every time a list of inputs is processed, it starts with determining the correct current state of the character. The game checks for a number of conditions, such

as if the character is on solid ground, if there is vertical movement input, what was the previous state, etc., to decide what state the player should be in. For example, if the jump button was pressed, the game needs to check if the player can currently jump (Figure 30).

```
2 references
private bool CanJump()
{
    return (IsGrounded() || (koyoteTimerGround > 0));
}
12 references
private bool IsGrounded()
{
    return Physics2D.BoxCast(box.bounds.center, box.bounds.size, 0f, Vector2.down, .1f, solidGround);
}
2 references
private void KoyoteCheck()
{
    if (IsGrounded())
    {
        koyoteTimerGround = koyoteTime;
    }
    else
    {
        koyoteTimerGround = Mathf.Max(0, koyoteTimerGround - Time.fixedDeltaTime);
    }
    if ((koyoteTimerJumpButton > 0) && CanJump())
    {
        StartState(CharState.Jump);
        koyoteTimerJumpButton = 0;
    }
    else
    {
        koyoteTimerJumpButton = Mathf.Max(0, koyoteTimerJumpButton - Time.fixedDeltaTime);
    }
}
```

Figure 30. Jump validation logic

The main condition for the player's ability to jump is that they stand on the ground. However, I have also implemented the so-called "coyote time", which means the player has a little leeway with the timing of the jump input. If the button was pressed a little bit before the player landed on the ground or a little bit after falling from it, the game will still let them perform the jump. This trick is often used by platformer games to make the controls feel more responsive – there is no sense in punishing the player for pressing the button a frame too early or too late, if the result is the player thinking the game just ignored the input.

After the current state has been decided, the game makes the character move accordingly. Instead of relying on Unity's in-built Rigidbody physics, I have opted to create my own simple and predictable system. This will also make it easier to

do conditional collisions. The movement is performed by calculating a vector representing the distance and direction of the movement for the current FixedUpdate(). When the vector is calculated, PlayerChar.cs calls MoveWithCollisionCheck(Vector3 moveVector) method (Figure 31).

```
private void MoveWithCollisionCheck(Vector3 moveVector)
{
    RaycastHit2D boxcastX = Physics2D.BoxCast(box.bounds.center, box.bounds.size, 0f,
        new Vector2(moveVector.x, 0), Mathf.Abs(moveVector.x), solidGround);
    if (boxcastX)
    {
        moveVector.x = Mathf.Sign(moveVector.x) * (Mathf.Max(0, boxcastX.distance - 0.1f));
        velocity.x = 0;
    }
    RaycastHit2D boxcastY = Physics2D.BoxCast(box.bounds.center, box.bounds.size, 0f,
        new Vector2(0, moveVector.y), Mathf.Abs(moveVector.y), solidGround);
    if (boxcastY) moveVector.y = Mathf.Sign(moveVector.y) * (Mathf.Max(0, boxcastY.distance - 0.1f));

    RaycastHit2D boxcastXY = Physics2D.BoxCast(box.bounds.center, box.bounds.size, 0f,
        moveVector, moveVector.magnitude, solidGround);
    if (boxcastXY) moveVector = moveVector.normalized * boxcastY.distance;

    transform.position += moveVector;
    ResolveCollisions();
}
```

Figure 31. MoveWithCollisionCheck() method

MoveWithCollisionCheck() first verifies if the character can move along x and y axes separately. If there’s an obstacle, the movement vector is adjusted accordingly, to prevent the character from phasing into any objects. After this, the complete adjusted vector is checked too. The separate checks are done so that, for example, if the player is jumping against the wall, it will not impede the vertical motion even though horizontal motion is impossible. The final check of the complete vector is done to prevent edge cases where the character might get stuck on a corner that would not be detected by horizontal and vertical checks. With the movement vector corrected to avoid collisions, it is added to the player’s position. Afterwards, a fail-safe ResolveCollisions() method is called that “pushes” the character out of any intersecting colliders in case something went wrong after all.

5.6 Projectiles

For player attack, the PlayerChar.cs script checks if the player can attack at the moment, triggers the shooting animation and creates a bullet at the ShootPoint

object's location. The player's Client and team IDs, as well as current direction is also passed to the bullet, so it knows which player and team it belongs to and what direction to move in. The Prefab used for creating bullets is a very simple one and only has three components: SpriteRenderer, BoxCollider2D and Projectile.cs. The last one is a custom script which controls the bullet, with most of the logic done in the Update() method (Figure 32).

```
void Update()
{
    Vector3 moveVector = direction * speed * Time.deltaTime;
    transform.position += moveVector;
    traveledDistance += moveVector.magnitude;
    if (traveledDistance > range)
    {
        Destroy(gameObject);
    }
    Collider2D[] collisions = Physics2D.OverlapBoxAll(transform.position, box.size, 0);
    foreach (Collider2D collision in collisions)
    {
        if (collision == box)
            continue;
        if (solidObstacle == (solidObstacle | (1 << collision.gameObject.layer)))
        {
            Destroy(gameObject);
        }
        IHealthInterface ihi = collision.gameObject.GetComponent<IHealthInterface>();
        if ((ihi != null) && (!SameTeam(ihi.GetObject())))
        {
            ihi.ReceiveDamage(damage);
            Destroy(gameObject);
        }
    }
}
```

Figure 32. Projectile.cs Update() method

Since all the bullets do is fly in a straight line, I opted to not make them NetworkObjects. This also ensures that their appearance on all players' screens coincides with the shooting animation and there's no chance of delay. Since the bullet GameObject is local, there's no need to tie its operation to FixedUpdate(), and Update() method can be used instead. All it does is move the object in the direction the bullet was shot, while counting the distance traveled and checking for collisions. The bullets have limited range and are destroyed once they exceed it. They are also destroyed if they hit a wall or a player from another team. In the latter case, they also call the hit player's ReceiveDamage() method to reduce HP. ReceiveDamage() method checks if it's running on the server and doesn't

execute any code if called on a client. This way hit detection and damage dealing is fully server-authoritative.

5.7 Match environment and flow

The match takes place in a scene separate from the lobby. It contains a few GameObjects for facilitating gameplay, among them:

- An object with InputManager.cs script discussed earlier, for detecting player inputs;
- An object with FightManager.cs script for spawning players and checking for victory conditions;
- A Canvas object that contains the health bar, loss and victory screens.

The scene also contains the game stage (Figure 33).

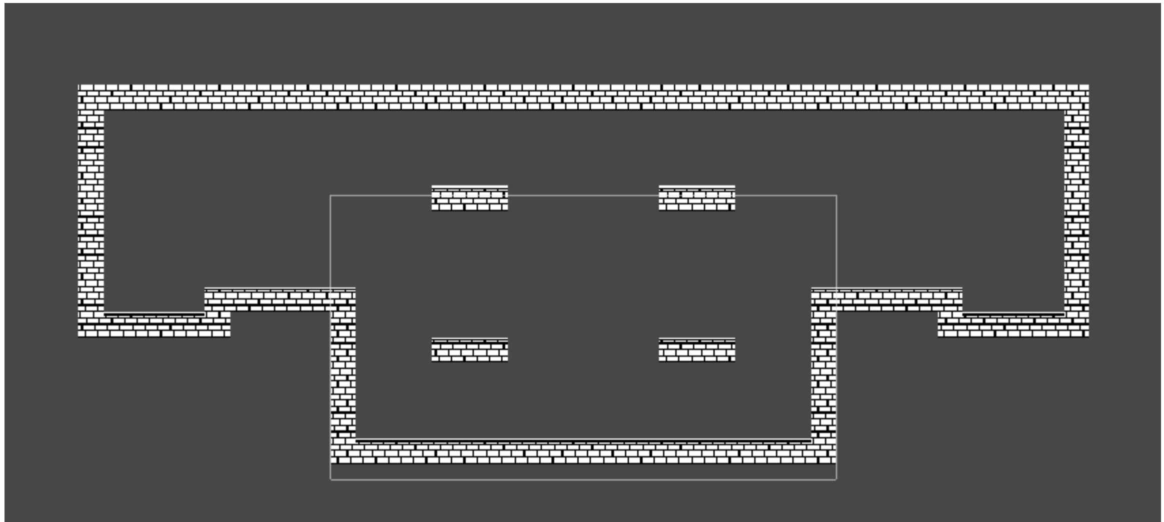

Figure 33. Game stage

The stage is made using Unity's tilemap system, which allows developers to build 2D environments from reusable "tiles", which is especially handy for games with pixel art graphics. Adding TilemapCollider2D component will generate colliders for the stage. The stage object is set to "Ground" layer, which PlayerChar.cs and Projectile.cs scripts are configured to detect as solid obstacles in their collision checks.

When all clients finish loading the scene, SceneManager's OnLoadEventCompleted event is triggered. FightManager.cs listens to this event to call SceneManager_OnLoadEventCompleted() method (Figure 34).

```
private void SceneManager_OnLoadEventCompleted(string sceneName,
    UnityEngine.SceneManagement.LoadSceneMode loadSceneMode, List<ulong> clientsCompleted, List<ulong> clientsTimedOut)
{
    if (IsServer)
    {
        Debug.Log("Spawning players: " + PlayerDataManager.Instance.GetPlayerDataList().Count);
        int i = 0;
        foreach (PlayerData player in PlayerDataManager.Instance.GetPlayerDataList())
        {
            Debug.Log(player);
            GameObject go = Instantiate(playerPrefab, spawnPoints[i].position, spawnPoints[i].rotation);
            go.GetComponent<NetworkObject>().SpawnWithOwnership(player.clientId);
            go.GetComponent<PlayerHealth>().OnHealthZero += PlayerHealth_OnHealthZero;
            TeamPlayer tp = new TeamPlayer { clientId = player.clientId, teamId = player.teamId, alive = true };
            teamPlayerList.Add(tp);
            i++;
        }
    }
}
```

Figure 34. SceneManager_OnLoadEventCompleted() method

On the server, SceneManager_OnLoadEventCompleted() gets the player list from the PlayerDataManager.cs script, instantiates player Prefabs for each of them, and spawns them with proper ownership. It also subscribes to every player's OnHealthZero event to detect deaths and builds its own list of players with their Client and team IDs and living status.

When each player Prefab spawns, on the owner's machine it sets the game camera as its child to make it follow the player, and commands the health bar UI object's HealthBar.cs script to subscribe to its OnHealthChange and OnHealthZero events. OnHealthChange event passes the new current to maximum health ratio as an argument, which HealthBar.cs uses to scale the green bar representing health to reflect the local player's HP. When a player's OnHealthZero event is triggered, the loss screen is shown to this player.

FightManager.cs reacts to every player's OnHealthZero event by calling PlayerHealth_OnHealthZero() method (Figure 35).

```
private void PlayerHealth_OnHealthZero(object sender, int ownerId)
{
    Debug.Log("FIGHT MANAGER - REG DEATH " + ownerId);
    ulong trueId = (ulong)ownerId;
    int i = GetIndexByClientId(trueId);

    TeamPlayer temp = teamPlayerList[i];
    temp.alive = false;
    teamPlayerList[i] = temp;

    int winningTeam = -1;
    for (int j = 0; j < teamPlayerList.Count; j++)
    {
        if (teamPlayerList[j].alive == true)
        {
            winningTeam = (winningTeam == -1) ? teamPlayerList[j].teamId : -2;
        }
    }

    if (winningTeam >= 0)
    {
        for (int j = 0; j < teamPlayerList.Count; j++)
        {
            if (teamPlayerList[j].teamId == winningTeam) VictoryClientRpc(new ClientRpcParams {
                Send = new ClientRpcSendParams { TargetClientIds = new List<ulong> {
                    teamPlayerList[j].clientId } } });
        }
    }
}
```

Figure 35. PlayerHealth_OnHealthZero() method

PlayerHealth_OnHealthZero() starts by finding the killed player in the list, and setting their living status to “false”. After this, it checks if only one team currently has any still living players. If that is the case, Client RPCs are sent to all members of the winning team to show them the victory screen. This finishes the match.

6 CONCLUSION

The goal of this thesis was to explore the tools and methods for developing an online multiplayer game using the Unity game engine. For this purpose, a prototype of an online multiplayer game has been created in said engine. Figure 36 shows the finished protype.

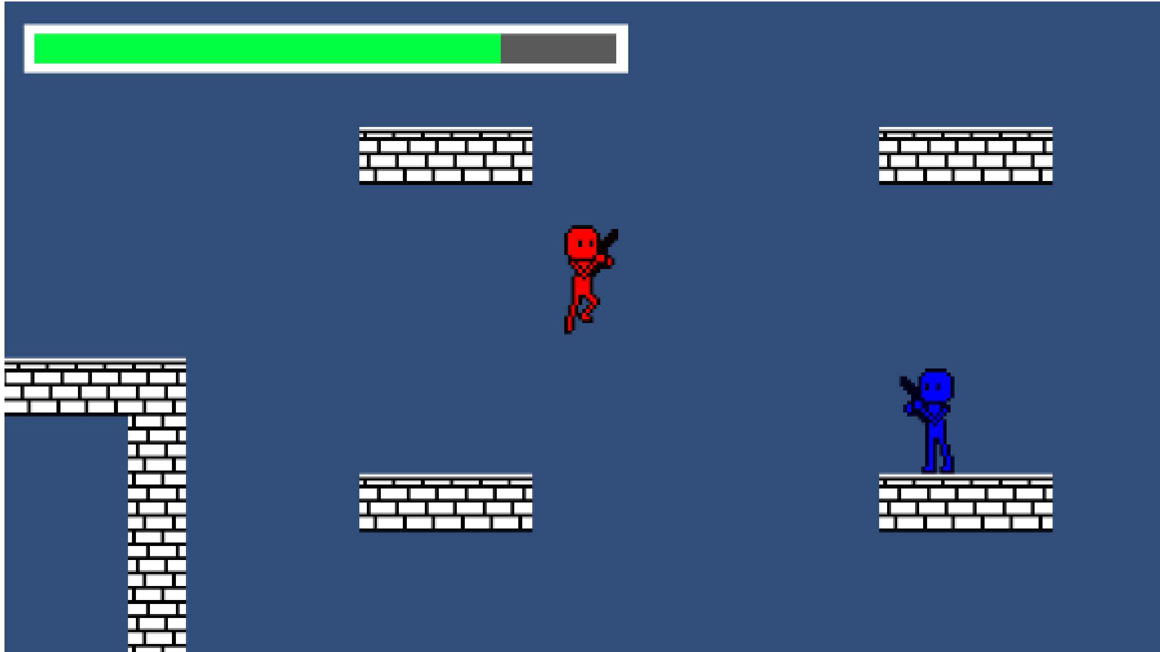

Figure 36. Finished game prototype

The prototype has all the functionality that was required. The players can find each other using the lobby system and connect through a relay without having to deal with the intricacies of networking. After choosing a team in the lobby, the match begins, where players control characters to run, jump and shoot. When only players from one team are left standing, victory is declared and the match ends. The players' movements are synchronized, and the important game mechanics are kept server-authoritative.

Although the prototype's implementation of netcode is somewhat simplistic and there is room for further development and improvement, it does the job of demonstrating the power and potential of the tools provided by the Unity engine. The prototype makes use of NGO instruments such as NetworkVariables and NetworkLists, Server and Client RPCs, as well as services like Lobby and Relay. The use of the engine came at no financial cost due to falling well within the allowances of the free license. However, developing the prototype into a complete game and its release may incur costs in the future.

The toolset offered by Unity is highly versatile and affordable. The engine significantly simplifies a lot of aspects of game development that would have taken much work to implement from scratch, netcode being just one of them.

Creating online multiplayer functionality would have been an enormous task – instead, the developers can spend their time building the game around an already made and tested networking solution. Notably, this thesis only explored tools and services provided by Unity Technologies exclusively – no third-party packages or assets were utilized. Making use of such resources could expand the engine's abilities even further.

To conclude, it cannot be overstated just how easy modern tools make game development compared to the times of the game industry's inception, even for something as idiosyncratic as online multiplayer. Though it still requires significant work and study, the barrier for entry has been greatly lowered. The tools will continue to improve, and new ones will be created. The online community will continue making these tools more accessible for new people by creating countless tutorials and exchanging tips on internet forums. And as this never-ending evolution goes on, more and more people will be able to put its products to use and make their game ideas into reality.

REFERENCES

Cardoso, H. 2022a. COMPLETE Unity Multiplayer Tutorial (Netcode for Game Objects). Youtube. Video clip. 26 September 2022. Available at: https://www.youtube.com/watch?v=3yuBOB3VrCk [Accessed 6 March 2023].

Cardoso, H. 2022b. Making a MULTIPLAYER Game? Join your Players with LOBBY! Youtube. Video clip. 29 November 2022. Available at: https://www.youtube.com/watch?v=-KDlEBfCBiU [Accessed 6 March 2023].

Cardoso, H. 2022c. How to use Unity Relay, Multiplayer through FIREWALL! (Unity Gaming Services). Youtube. Video clip. 2 December 2022. Available at: https://www.youtube.com/watch?v=msPNJ2cxWfw [Accessed 6 March 2023].

Dealessandri, M. 2020. What is the best game engine: is Unity right for you? Web page. Available at: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you [Accessed 14 May 2023].

Dongen, J. van. 2015. Why adding multiplayer makes game coding twice as much work. Blog. 9 August 2015. Available at: http://joostdevblog.blogspot.com/2015/08/why-adding-multiplayer-makes-game.html [Accessed 6 May 2023].

Engelbrecht, D. 2021. Building Multiplayer Games in Unity: Using Mirror Networking. Berkeley, CA: Apress.

Fiedler, G. 2010. Floating Point Determinism. Blog. 24 February 2010. Available at: https://gafferongames.com/post/floating_point_determinism/ [Accessed 20 May 2023].

Fiedler, G. 2014. Deterministic Lockstep. Blog. 29 November 2014. Available at: https://gafferongames.com/post/deterministic_lockstep/ [Accessed 11 May 2023].

Gambetta, G. n.d.a. Fast-Paced Multiplayer (Part I): Client-Server Game Architecture. Web page. Available at: https://gabrielgambetta.com/client-server-game-architecture.html [Accessed 12 March 2023].

Gambetta, G. n.d.b. Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation. Web page. Available at: https://gabrielgambetta.com/client-side-prediction-server-reconciliation.html [Accessed 12 March 2023].

Gambetta, G. n.d.c. Fast-Paced Multiplayer (Part III): Entity Interpolation. Web page. Available at: https://gabrielgambetta.com/entity-interpolation.html [Accessed 12 March 2023].

Gambetta, G. n.d.d. Fast-Paced Multiplayer (Part IV): Lag Compensation. Web page. Available at: https://gabrielgambetta.com/lag-compensation.html [Accessed 12 March 2023].

Gao, Y. 2018. Netcode Concepts Part 3: Lockstep and Rollback. Blog. 22 April 2018. Available at: https://meseta.medium.com/netcode-concepts-part-3-lockstep-and-rollback-f70e9297271 [Accessed 11 May 2023].

Glazer, J. L. & Madhav, S. 2016. Multiplayer game programming: Architecting networked games. New York, N.Y.: Addison-Wesley.

Halpern, J. 2018. Developing 2D Games with Unity: Independent Game Programming with C#. Berkeley, CA: Apress.

Lemay, S. 2022. Another thing that may limit the scale of games made with NGO […]. Unity Forum. Forum post. 23 September 2022. Available at: https://forum.unity.com/threads/what-makes-ngo-limited-to-small-scale-games-only.1338971/#post-8461043 [Accessed 18 May 2021].

Mattis, J. 2023a. Netcode Architectures Part 1: Lockstep. Blog. 28 March 2023. Available at: https://snapnet.dev/blog/netcode-architectures-part-1-lockstep/ [Accessed 11 May 2023].

Mattis, J. 2023b. Netcode Architectures Part 2: Rollback. Blog. 18 April 2023. Available at: https://snapnet.dev/blog/netcode-architectures-part-2-rollback/ [Accessed 11 May 2023].

Meddane, R. 2020. STUN TURN and ICE Demystified. Cisco. PDF Document. Available at: https://community.cisco.com/kxiwq67737/attachments/kxiwq67737/4691-docs-collaboration-voice-video/10059/1/STUN%20TURN%20and%20ICE%20for%20NAT%20Traversal.pdf [Accessed 8 May 2023].

Motion Picture Association, Inc. 2022. 2021 THEME Report. PDF document. Available at: https://www.motionpictures.org/wp-content/uploads/2022/03/MPA-2021-THEME-Report-FINAL.pdf [Accessed 8 April 2023].

Nbn. 2017. Tick rate: What is it and how can it affect your gaming experience? Blog. 1 February 2017. Available at: https://www.nbnco.com.au/blog/entertainment/what-is-tick-rate-and-what-does-it-do [Accessed 10 May 2023].

Pandey, H. 2022. Peer-to-peer vs client-server architecture for multiplayer games. Hathora. Web page. Available at: https://blog.hathora.dev/peer-to-peer-vs-client-server-architecture/ [Accessed 30 April 2023].

Panek, C. 2020. Networking Fundamentals. Hoboken, NJ: John Wiley & Sons, Inc.

Pusch, R. 2019. Explaining how fighting games use delay-based and rollback netcode. Web page. Available at:

https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/ [Accessed 12 May 2023].

PwC. 2022. Perspectives from the Global Entertainment & Media Outlook 2022–2026. PDF document. Available at: https://www.pwc.com/gx/en/industries/entertainment-media/outlook/downloads/PwC_Outlook22.pdf [Accessed 8 April 2023].

Ronimo Games. 2012. Awesomenauts. Video game. Utrecht: Ronimo Games.

Stagner, A. R. 2013. Unity Multiplayer Games. Birmingham: Packt Publishing.

Tanenbaum, A. & Wetherall, D. 2010. Computer Networks. 5th ed. Boston: Prentice Hall.

TIA. n.d. Telecom Glossary. Web page. Available at: https://standards.tiaonline.org/resources/telecom-glossary [Accessed 29 April 2023].

Unity Technologies. 2023a. Unity – Manual: Creating and Using Scripts. Web page. Available at: https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html [Accessed 15 May 2021].

Unity Technologies. 2023b. Unity – Manual: GameObjects. Web page. Available at: https://docs.unity3d.com/Manual/GameObjects.html [Accessed 15 May 2021].

Unity Technologies. 2023c. Unity – Manual: Scenes. Web page. Available at: https://docs.unity3d.com/Manual/CreatingScenes.html [Accessed 15 May 2021].

Unity Technologies. 2023d. Unity – Manual: Unity's Interface. Web page. Available at: https://docs.unity3d.com/Manual/UsingTheEditor.html [Accessed 15 May 2021].

Unity Technologies. n.d.a. About Netcode for GameObjects. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/about/ [Accessed 16 May 2021].

Unity Technologies. n.d.b. About Unity Transport. Web page. Available at: https://docs-multiplayer.unity3d.com/transport/current/about/ [Accessed 16 May 2021].

Unity Technologies. n.d.c. ClientRpc. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/message-system/clientrpc/ [Accessed 17 May 2021].

Unity Technologies. n.d.d. Heartbeat a lobby. Web page. Available at: https://docs.unity.com/lobby/en/manual/heartbeat-a-lobby

[Accessed 18 May 2021].

Unity Technologies. n.d.e. NetworkBehaviour. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/basics/networkbehavior/ [Accessed 16 May 2021].

Unity Technologies. n.d.f. NetworkManager. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/components/networkmanager/ [Accessed 16 May 2021].

Unity Technologies. n.d.g. NetworkObject. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/basics/networkobject/ [Accessed 17 May 2021].

Unity Technologies. n.d.h. NetworkVariable. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/basics/networkvariable/ [Accessed 17 May 2021].

Unity Technologies. n.d.i. Object Spawning. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/basics/object-spawning/ [Accessed 17 May 2021].

Unity Technologies. n.d.j. Overview of services. Web page. Available at: https://docs.unity.com/ugs-overview/en/manual/overview-of-services [Accessed 18 May 2021].

Unity Technologies. n.d.k. Reliability. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/message-system/reliability/ [Accessed 17 May 2021].

Unity Technologies. n.d.l. Sending Events with RPCs. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/messaging-system/ [Accessed 17 May 2021].

Unity Technologies. n.d.m. ServerRpc. Web page. Available at: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/message-system/serverrpc/ [Accessed 17 May 2021].

Unity Technologies. n.d.n. Unity Gaming Services Pricing - Start Building for Free. Web page. Available at: https://unity.com/solutions/gaming-services/pricing [Accessed 18 May 2021].

Unity Technologies. n.d.o. Use Relay with Netcode for GameObjects (NGO). Web page. Available at: https://docs.unity.com/relay/en/manual/relay-and-ngo [Accessed 18 May 2021].

Yacht Club Games. 2019. Shovel Knight Showdown. Video game. Los Angeles: Yacht Club Games.

LIST OF FIGURES

Figure 1. Network topologies....7
Figure 2. Client-server communication with a dumb client 18
Figure 3. Client-side prediction....19
Figure 4. Character state on different machines with client-side prediction and entity interpolation 21
Figure 5. Lockstep P2P architecture model....22
Figure 6. Rollback example....24
Figure 7. Unity window layout example 27
Figure 8.Unity Gaming Services dashboard....35
Figure 9. Shovel Knight Showdown gameplay (Shovel Knight Showdown 2019)38
Figure 10. Awesomenauts gameplay (Awesomenauts 2012)....38
Figure 11. Unity Hub 40
Figure 12. Menu structure 41
Figure 13. Menu structure in Unity, with Main Menu's object hierarchy fully unfolded....43
Figure 14. IMenuSwitchInterface implementation inside SubMenu class. 44
Figure 15. Object references in the main menu script 45
Figure 16. Main menu's methods 46
Figure 17. Main Menu 47
Figure 18. CreateLobby() method 49
Figure 19. PlayerData struct....51
Figure 20. Lobby submenu....52
Figure 21. Player Prefab structure....53
Figure 22. Action Map in Input System....54
Figure 23. ReceiveDamage method....55
Figure 24. OnHealthChanged method....55
Figure 25. Character sprites....55
Figure 26. CharAnimControl Animator-controlling methods 56
Figure 27. ShootPoint's location on the assembled player sprite 57
Figure 28. InputEventType enum and InputEvent struct 59
Figure 29. CharState enum 60
Figure 30. Jump validation logic....61

Figure 31. MoveWithCollisionCheck() method 62
Figure 32. Projectile.cs Update() method 63
Figure 33. Game stage..64
Figure 34. SceneManager_OnLoadEventCompleted() method...........................65
Figure 35. PlayerHealth_OnHealthZero() method 66
Figure 36. Finished game prototype..67