TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Jegor Kuznetsov 206572IAIB

CREATING A DYNAMIC MULTIPLAYER 2D GAME USING UNITY AND PHOTON ENGINE'S FUSION-SDK

Bachelor's Thesis

Supervisor: Ago Luberg
PhD

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jegor Kuznetsov 206572IAIB

Dünaamilise mitme mängijaga 2D-mängu loomine Unity ja Photon Engine'i Fusion-SDK-ga

Bakalaureusetöö

Juhendaja: Ago Luberg
PhD

Tallinn 2024

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Jegor Kuznetsov

14.01.2024

Abstract

The goal of this thesis is to develop a multiplayer 2D game using Unity Engine with Photon Engine's Fusion-SDK. The application was commissioned by GimmeBreak members who advised author to learn game-development via game creation.

Author aims to show that Photon Engine's Fusion-SDK is suitable for arcade games creation. Game has to provide fair gameplay for each user based only on player skill and reaction. Main idea behind creating Game is to provide a solid example of Photon Engine's Fusion-SDK implementation into Unity based project for beginners in multiplayer game-development.

In addition to developing Game, this thesis also aims to prove that Photon Engine's Fusion-SDK is suitable for small indie companies that include 1-10 members or developers who are not familiar with multiplayer game development to be able to create multiplayer game. Moreover author is going to show advantages and disadvantages of Photon Engine's Fusion-SDK in comparison to other SDK's available on the market for multiplayer game creation.

The thesis is written in English and is 31 pages long, including 6 chapters, 19 figures and 4 tables.

Annotatsioon

Dünaamilise mitme mängijaga 2D-mängu loomine Unity ja Photon Engine'i Fusion-SDK-ga

Selle lõputöö eesmärk on välja töötada mitme mängijaga 2D mäng, kasutades Unity Engine'i platvormi koos Photon Engine'i Fusion-SDK-ga. Rakenduse tellisid firma GimmeBreak liikmed, kes soovitasid autoril õppida mängude arendamist mängude loomise kaudu.

Autori eesmärk on näidata, et Photon Engine'i Fusion-SDK sobib arkaadmängude loomiseks. Mäng peab pakkuma igale kasutajale ausat mängu, mis põhineb ainult mängija oskustel ja reaktsioonil. Mängu loomise põhiidee on luua konkreetne näide Photon Engine'i Fusion-SDK rakendamisest Unity-põhises projektis algajatele mitme mängijaga mängude arendamisel.

Lisaks mängu arendamisele on käesoleva lõputöö eesmärk tõestada, et Photon Engine'i Fusion-SDK sobib mitme mängijaga mängu loomiseks väikestele ettevõtetele, kus on 1-10 liiget, või arendajatele, kes pole mitmikmängude arendamisega kursis. Lisaks näitab autor Photon Engine'i Fusion-SDK eeliseid ja puudusi võrreldes teiste turul saadaolevate mitme mängijaga mängude loomise raamistikega.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 31 leheküljel, 6 peatükki, 19 joonist ja 4 tabelit.

List of Abbreviations and Terms

CCU	Concurrent users
CPU	Central Processing Unit
MB	Mega Byte
RPC	Remote Procedure Calls
SDK	Software Development Kit

Table of Contents

List of Figures

List of Tables

1. Introduction

In the realm of multiplayer game development, an array of solutions inundates the market. However, for beginners, this abundance can present more of a challenge than an advantage. To select an appropriate solution, a comprehensive understanding of their differences becomes imperative. Some of the prominent free solutions among numerous others include:

- Mirror
- Unity Netcode for GameObjects (NGO/Netcode)
- UNet
- Fishnet
- GGPO

In contemporary gaming, multiplayer games have gained immense popularity. While single-player games maintain their allure in the gaming community, the enduring appeal of multiplayer gaming stands as a testament to their longevity. Titles like:

- DOTA (Defence of the Ancients)
- Counter-Strike (CS)
- Overwatch
- Players Unknown Battle Ground (PUBG)
- Call of Duty (COD)

Exemplify the enduring legacy of multiplayer games, sustaining their presence for over a decade.

Multiplayer games constitute a cornerstone of the gaming industry, fostering prolonged engagement among players. Their collaborative and competitive nature sustains interest and drives the continuous evolution of these games. As players seek dynamic experiences and social interactions, multiplayer games offer an avenue for enduring entertainment.

The enduring popularity of multiplayer games can be attributed to several factors. Their inherent social nature fosters connections among players, creating vibrant communities around these games. Additionally, regular updates, new content, and innovative gameplay mechanics contribute to their longevity by continuously engaging and reinvigorating player interest.

Successful multiplayer games exhibit adaptability to evolving trends by integrating new technologies, addressing player feedback, and introducing fresh content. This adaptability ensures their relevance amidst shifting gaming landscapes and changing player preferences.

Innovation serves as a catalyst for the sustained success of multiplayer games. Introducing novel features, modes, and gameplay mechanics not only captivates existing players but also attracts new audiences, thereby ensuring the game’s continued relevance and success.

2. Problem Statement

In this thesis the Author will create a multiplayer game based on Unity Engine[1] using Photon Engine's Fusion-SDK[2].

2.1 Clients' needs

The clients of this project are members from GimmeBreack company. Clients need an easy-to-use example that would provide overall understanding of basic tools and mechanics. Game that could have been used as a sample offering users with a guide book on leveraging Photon Engine's Fusion-SDK. The contact person between that group and the author is leading developer of the company, who advised where to start and what should be shown to gain and provide initial experience for the users to be able to create their first multiplayer game.

2.2 Thesis goals

This thesis aims to create a 2D game that would be easy to understand and would cover all the basics of multiplayer game development. The game should be preferably quickly developed and show potential usage of tools and features of Photon Engine's Fusion-SDK[3]. Such as:

- Hosted/Server Mode
- Shared Mode
- Network Runner
- Network Object
- Network Behaviour
- Simulation Behaviour
- Networked Property
- Built-in Network Behaviours
  - Network Transform
  - Network Rigid Body
  - Network Character Controller
- Remote Procedure Calls
- Tick-based Call-Backs
- Input handling

- Input Authority
- Network-Safe Counterparts
- Predicted Movement

For Authors' better understanding of game development as a whole, client advised to also learn and add such aspects as:

- Graphics
- Scene Management
- User Interface
- Visualisation
  - Particle System
  - Animation
  - Sprite rendering
- Collision detection

The end result should be a user-friendly 2D multiplayer game that could be understood without requiring extensive knowledge about Photon Engine's Fusion-SDK. This application will be open-source and available for any level of subscribers of Photon Engine to use.

2.3 Existing solutions

Few existing solutions in multiplayer game development can match Fusion-SDK in terms of both performance and features. However, SDKs like Mirror[4], Fish-Net[5], and NGO[1] have their own dedicated communities who prefer these resources[6]. The author intends to conduct a performance and feature comparison among some of these solutions[7][8].

3. Existing Solutions Comparison

3.1 NGO

Netcode for GameObjects (NGO) serves as a powerful networking tool designed specifically for Unity. Its primary purpose is to simplify networking complexities by abstracting the underlying logic. By utilizing NGO, developers can efficiently transmit GameObjects and world data across multiple players in a network session.

One of its core advantages lies in its ability to handle the communication of GameObjects seamlessly, liberating developers from delving into the intricacies of low-level protocols and networking frameworks. Instead, it empowers them to concentrate on the game's creation and functionalities.

3.2 Mirror

Mirror[9] is a battle-tested networking solution initially rooted in UNET, boasting eight years of industry application and renowned titles like Population: ONE in its portfolio.

It's a stable, modular, and user-friendly framework suitable for various game types, including small-scale MMORPGs. The architecture consolidates server and client elements within a single project, optimizing productivity.

Developers working with Mirror utilize NetworkBehaviour instead of MonoBehaviour, simplifying coding with specialized tags like [Server] or [Client], facilitating server-client communication through designated functions like [Command], [ClientRpc], and [TargetRpc]. It also incorporates SyncVar and SyncList for automatic server-client variable synchronization.

3.3 Fish-Net

Fish-Networking, commonly known as Fish-Net, emerges as a versatile, original networking solution for Unity, offering an extensive array of features not typically found in other free solutions. It's built from scratch to support various network topologies through its Transport system, allowing seamless communication between servers, clients, and third-party entities.

Primarily designed to be server authoritative, Fish-Net enables the use of dedicated servers while allowing users to alternate between server and client roles for faster development and testing.

The platform boasts a high-level API, ensuring swift access to synchronize states, logic, and objects without diving into intricate technical details. Even though it prioritizes a user-friendly approach, it offers access to low-level functionalities through events or inheritance for advanced users.

Fish-Networking emphasizes stability and consistency in its updates. It commits to a no-break promise, ensuring minimal API or behavior changes between major versions, which are released no more frequently than every six months unless crucial. Any necessary changes are kept straightforward, detailed in their Break Solutions section.

3.4 Photon Engine's Fusion

Fusion stands as a cutting-edge networking library designed explicitly for Unity, offering a seamless integration process into Unity's workflow. Its core focus is simplicity, yet it packs advanced functionalities like:

- Tick-based Simulation
- Built-in data compression
- Client-side prediction
- Lag compensation
- Snapshot Interpolation
- Replication Systems

Within Fusion, network state and Remote Procedure Calls (RPCs) are effortlessly defined using attributes directly on MonoBehaviour methods and properties, eliminating the need for explicit serialization code. Even network objects can be designated as prefabs, utilizing Unity's latest prefab capabilities.

Underneath its surface, Fusion optimizes bandwidth usage with a sophisticated compression algorithm, ensuring minimal CPU load. It manages data transfer through compressed snapshots or partial chunks for eventual consistency, supported by a customizable area-of-interest system that accommodates large player counts seamlessly.

Fusion operates in two modes: Shared Mode and Hosted Mode. Each mode determines the authority over network objects and subsequently influences available SDK features, all while maintaining a robust tick-based simulation for consistent performance.

One of the noticeable features of Fusion is various compatibility with such platforms as:

- Microsoft Windows
- macOS (Intel 64 & Apple Silicon)
- WebGL
- Android (including Quest)
- iOS
- Linux
- Nintendo Switch
- Xbox One
- Xbox Series X & S
- PS4
- PS5

3.5 Performance comparison

3.5.1 Bandwidth

100 CCU Bandwidth:

- FishNet Server per second sent 7.3Megabytes(MB) and received 1.1MB.
- Mirror Server per second sent 31.8MB and received 3.2MB.

200 CCU Bandwidth:

- FishNet Server per second sent 29.9MB and received 2.2MB.
- Mirror Server per second sent 94.3Mb and received 4.8MB.

Concurrent Users refers to the number of players present at the same time within a game session. Amount of sent and received data per second affects the strain on the networking layer. Elevated loads often result in causing delayed updates in the game. Notice significant advantage of Fusion-SDK in comparison to MLAPI and Mirror in terms of bandwidth(see Table 1).

Table 1. *Bandwidth Usage KB/s (lower is better)*

	100 objects	**250 objects**	**500 objects**
Fusion	12	28	54
MLAPI	61	151	281
Mirror	67	169	334

3.5.2 Allocations per Frame

Memory allocations involve reserving space in the computer's memory to store data during the execution of a program, and they are essential for managing data structures and objects. In a comparative analysis with other solutions(see Table 2), Fusion-SDK demonstrated superior results in efficiently managing memory allocations.

Table 2. *Allocations per frame KB (lower is better)*

	500 objects, 32 clients
Fusion	0
MLAPI	619.24
Mirror	311.91

3.5.3 CPU Usage On Server

CPU usage on server defines what is the load on CPU for hosting a server(see Table 3).

Table 3. *CPU usage on server ms (lower is better)*

	500 objects, 32 clients
Fusion	0.514
MLAPI	8.112
Mirror	23.716

3.5.4 CCU Scaling

100 CCU Scaling:

- FishNet Server lost 1.5% of it's performance.
- FishNet Client ran at 1350 FPS.

- Mirror Server lost 29.5% of it's performance.
- Mirror Client ran at 791 FPS.

200 CCU Scaling:

- FishNet Server lost 7.4% of it's performance.
- Mirror Server lost 83.2% of it's performance.

3.6 Overall Comparison And community's Opinion

3.6.1 NGO

- Advantages:
  - Emphasizes abstraction of networking logic for GameObjects.
  - Supports messaging and synchronization.
- Disadvantages:
  - Serious features shortage.
  - Slow performance.

There is not much to say about Netcode for GameObjects as it is new and fresh SDK that was released recently in June 2022 by Unity. Huge advantage among other solutions is that NGO is developed by Unity for Unity. In recent future NGO might become if not the best then reach top 3 solutions overall.

3.6.2 Mirror

- Advantages:
  - Stable, widely used for Unity networking.
  - Provides functionalities across messaging, synchronization, GameObjects, Scenes, and Utility.
  - Supports various transports and offers features like transform synchronization and network profiler.
  - Open source solution.
- Disadvantages:
  - Out-dated solution.
  - In comparison to other solutions lack in functionality.
  - Comparing to other networking tools looses in performance

Overall it can be said that Mirror is not beginner-friendly and doesn't provide efficient[10] and full of features networking solution out of the pocket. It is recommended to use only for advanced developers with sufficient experience in networking. For a group of well -skilled developers Mirrors' open source code can be used and sharpened for a specific game which can lead to improved efficiency.

3.6.3 Fish-Net

- Advantages:
  - Open-source solution with multiple transport options including WebSocket, Steam, and Epic Transport.
  - Focuses on utility functions like lag compensation and supports GameObjects, but with variations compared to Mirror.
  - Dominates in performance in comparison to any other free solution available on the market.
- Disadvantages:
  - Not all the features provided for users are free.
  - Some advanced features require deep understanding of the networking logic.

Overall Fish-Net is the best free choice available on the networking solutions market, that can satisfy both beginners and hardcore developers. Provides functionality that can be found useful in A-AA games. According to official statistic provided both by fishnet and mirror:

- FishNet has 70% more FPS in comparison to Mirror.
- In terms of bandwidth FishNet used 67-78% less bandwidth than Mirror.
- In server performance test for 200 CCU Fishnet retained 92.6% server performance, while Mirror retained 16.8% performance.
- In server performance with 4000 idle objects Fishnet retained 100% server performance, while Mirror retained 36% performance.

3.6.4 Fusion-SDK

- Advantages:
  - Provides servers to run game on.
  - Optimized bandwidth usage.
  - Low CPU consumption. compared to MLAPI or Mirror or even Fish-Net.
  - Supports different network topologies and replication algorithms, offering a

    consistent API.
  - Simplifies integration into Unity workflow and includes advanced features out of the box like:
    * Data compression.
    * Client-side prediction.
    * Lag compensation
  - Supports various data transfer modes
  - Has a configurable area-of-interest system for high player counts.
  - Implements robust tick-based simulation and operates in Shared Mode or Hosted Mode, determining authority over network objects and available SDK features.
- Disadvantages:
  - Only 20 CCU are available on free version.
  - High price for indie game developers/studios starting from 125$ per month.
  - Doesn't have full open source code.
  - Small community.
  - Was released quite recently in 2022.

Fusion stands as a prominent choice for small to medium-sized enterprises seeking to introduce online gaming experiences without extensive networking expertise. Its swift development capabilities offer an expedited solution at an accessible price point, catering specifically to indie game studios. By offering server provision, Fusion mitigates expenses and alleviates the burden of server setup, hosting, and maintenance, further streamlining operations for these studios. It strikes a balance, being user-friendly for beginners yet robust enough to satisfy the demands of seasoned developers.

3.6.5 Featrue comparison

Combining insights from the game developers' community[11] and the author's knowledge, augmented by mentor support, a feature comparison table(see Table 4) has been crafted.

Table 4. *Overall comparison of SDKs*

Feature	**Fusion**	**Fish-Net**	**Mirror**	**NGO**
Source Code Available	Y(Free/Paid)	Y	Y	Y
Listen Server (Host)	Y	Y	Y	Y
RPC	Y	Y	Y	Y
Large Packets	Y	Y	N	N

Continues...

Table 4 – *Continues...*

Feature	**Fusion**	**Fish-Net**	**Mirror**	**NGO**
Player Prefab Spawning	Y	Y	Y	Y
Nested Network Behaviours	Y	Y	N	N
Client-side Prediction	Y	Y	N	N
Animation Synchronization	Y	Y	Y	Y
Scene Network Objects	Y	Y	Y	Y
Lag Compensation	Y	Y(Paid)	N	N
Host migration	Y	N	N	N
Automatic Object Pooling	Y	Y	N	N
Server hosting	Y	N	N	N

4. Game Development Process

4.1 Conceptualization Phase

In pursuit of developing a game that strikes a balance between simplicity and competitiveness, extensive deliberations with senior developers and mentors led the author to stick with classic game principles. Recognizing the hallmark of a classic game involves enduring testing and validation over several decades—games like chess or checkers embody the core principles of enduring success in competitive gaming. Central to their essence are replayability and equal conditions enabling multiple players to vie for victory. Ensuring sustained player engagement necessitates a unique experience for each gaming session, a feat achievable through a flexible in-game environment that provides wide range of maneuvers.

Graphics play a pivotal role in shaping players' perceptions of a game. To entice players to return, the visual presentation must be engaging. To avoid mere plain, nondescript elements, the author has identified several free asset packs that fulfill the fundamental graphical requirements for the initial concept phase of development. Additionally, incorporating straightforward particle systems and animations will enhance the overall visual appeal.

4.2 Planning and Pre-Production

Game development encompasses more than mere coding. It begins with conceptualizing core gameplay that is not just entertaining for first-time players but also engaging for dedicated fans who invest countless hours. This involves creating gameplay rules that are intuitive and enjoyable for all, checkers with its straightforward rules and balanced competitiveness, stands as a great example..

The initial concept revolved around introducing a captivating gameplay experience featuring diverse characters, each possessing unique abilities within a dynamically changing environment. However, this idea encountered substantial criticism, particularly regarding the fundamental mechanics of the game and the distribution of resources. Challenges emerged concerning character selection, potentially leading to imbalances in abilities and fairness in gameplay. Additionally, the proposed 1 vs 1 or 2 vs 2 (maximum 4 vs 4) gameplay raised concerns about effectively countering opponents, which could result in favoritism toward one player over another. Criticism of titles like DOTA 2, Mortal Kombat,

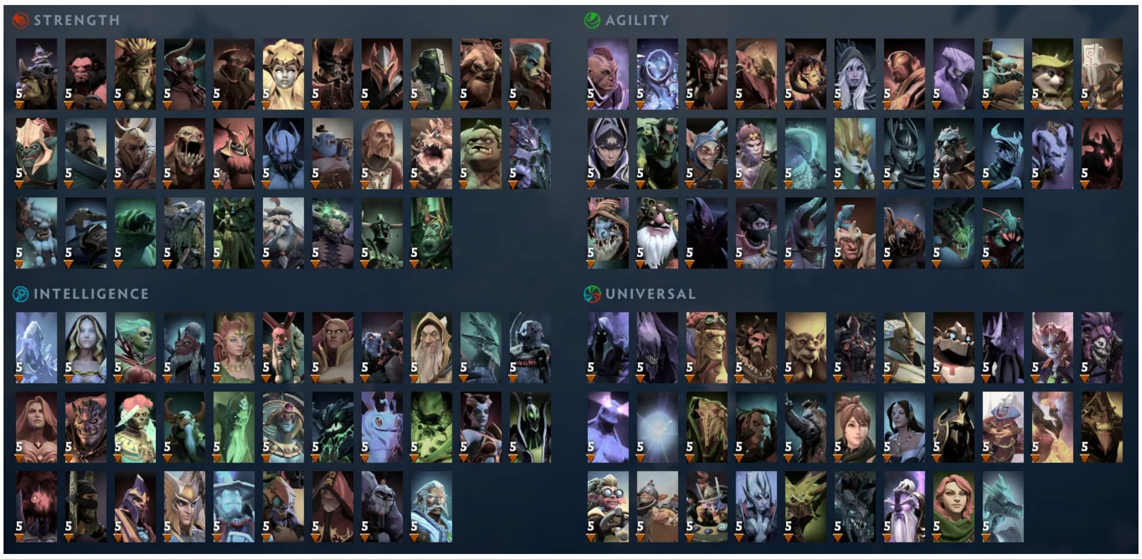


Figure 1. DOTA2 all(124) heroes

and many others often revolves around issues related to characters(see Figure 1) balance.

The concept of diverse characters with varying abilities was eventually discarded to ensure a balanced playing field. Similarly, limitations in resources, specifically time and manpower made it unfeasible to create a fully responsive environment without risking gameplay integrity.

4.3 Development Phase

4.3.1 Game design

To maintain simplicity and ensure fairness, the concept shifted from multiple characters to a single character, ensuring an even playing field for all. The final decision focused on a gameplay concept where players aim to force each other out of the game's borders. While keeping the gameplay straightforward, the goal was to offer players a sense of accomplishment and satisfaction, allowing them to creatively manipulate the environment using skills honed over countless hours of play. The core movement idea centered around skating, emphasizing smooth and gratifying maneuvers.

4.3.2 Level Design

In order to achieve balance between players in the game, design of the level went through several iterations. Since the emphasis was placed on providing players with enjoyment through interaction with the environment, the author needed to offer an environment that

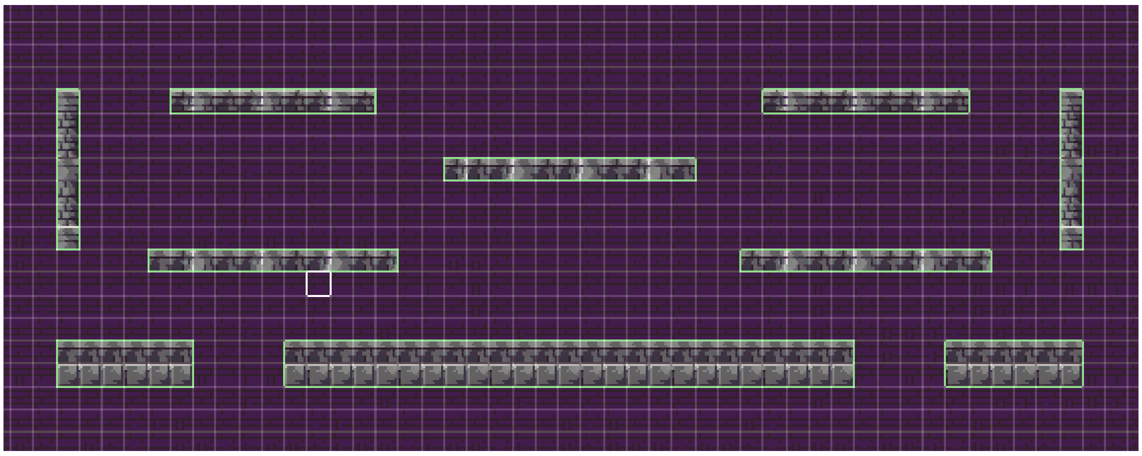


Figure 2. LevelDesign first iteration

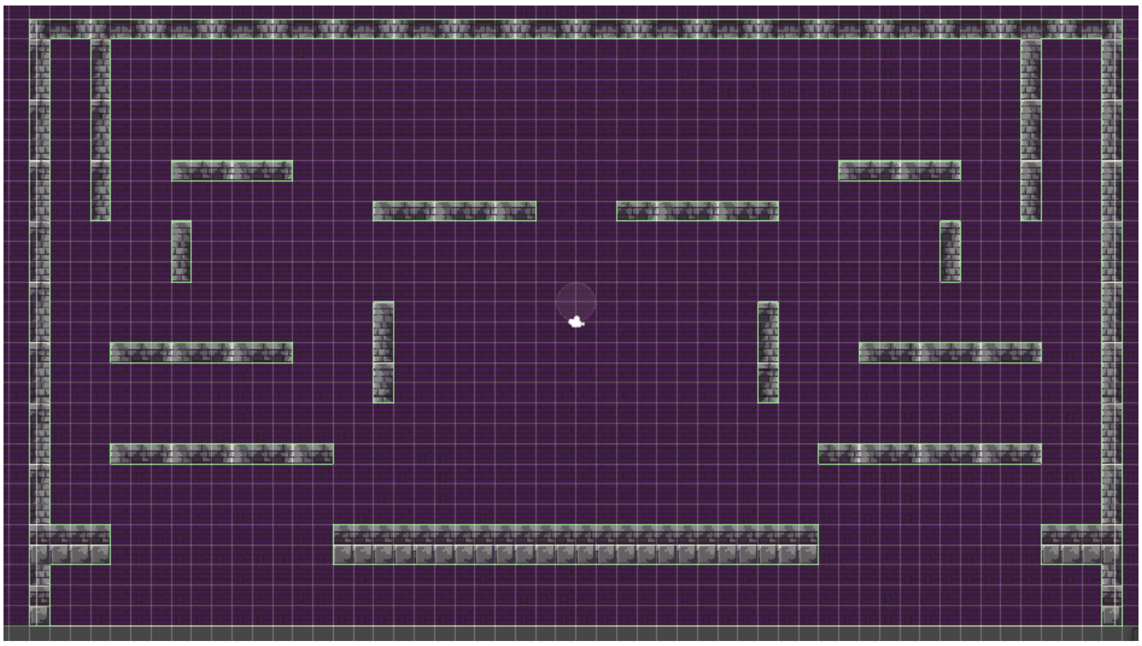


Figure 3. LevelDesign second iteration

allowed for the implementation of all possible movement mechanics. However, the initial iteration(see Figure 2) was too simplified to apply all potential mechanics. There was a lot of open space, primarily horizontal surfaces, which prevented the full realization of the potential for wall sliding and wall rebounding.

After reevaluating the environment prototype, the focus shifted from horizontal to vertical surfaces to maximize the possibilities of wall sliding and rebounding mechanics. However, the second iteration(see Figure 3) was confined, featuring enclosed walls and a roof, reducing the game's intensity and lessening the fear of being thrown out of the map. Additionally, there was excessive empty space in the middle of the map, which could have been utilized to offer players more variety and flexibility in their movement.

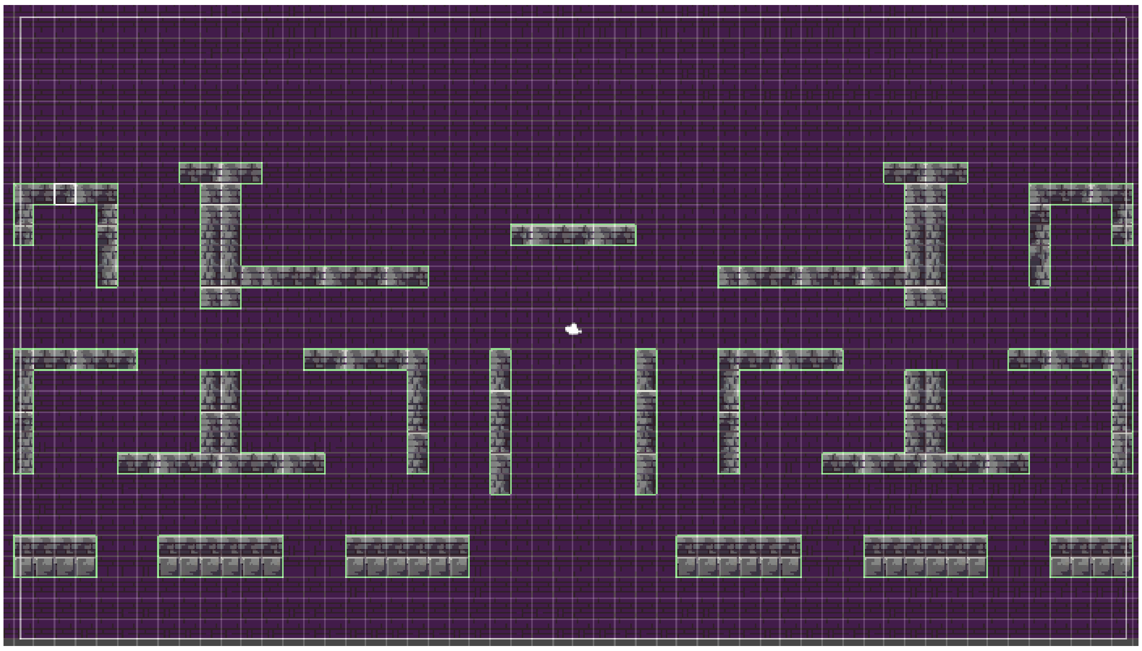


Figure 4. LevelDesign third iteration

Although the second iteration was fairly decent, the aim was to offer players the freedom to move in any direction without constraints like closed spaces, walls, or roofs.

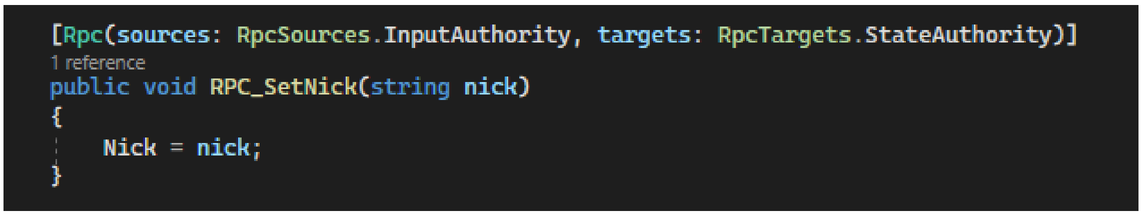
In the third and final iteration(see Figure 4), the focus was on integrating both vertical and horizontal surfaces to enhance maneuverability and flexibility for the players. The central area was redesigned with the intention of eliminating safe zones, turning it into the most challenging and risky part of the map. The final iteration was developed with a straightforward concept: it relies entirely on players' skill and reaction time. Side sections were revamped to create a dynamic playground that allows for swift traversal. The emphasis was on minimizing borders and limitations, maximizing available space, and introducing diverse obstacles for players to use in their own benefit.

4.4 Fusion-SDK usage

4.4.1 RPC vs Networked property

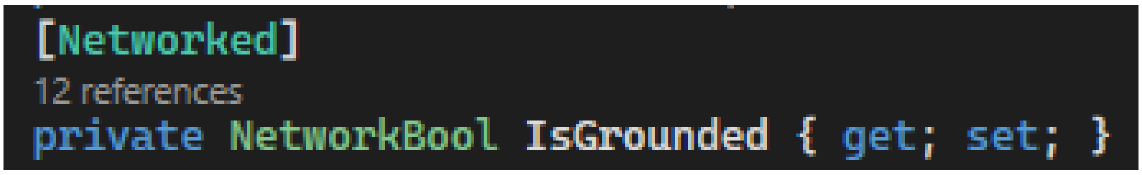
In the initial stages of development, the author grappled with various bugs and hurdles associated with Fusion-SDK implementation. Among these challenges, a key concern centered around comprehending the utilization of Remote Procedure Calls (RPC) and Networked properties. Initially, discerning the disparity between these two was perplexing.

In essence, Networked properties involve data transfer between players repeatedly, several times per second. In contrast, RPC ensures the delivery of an action once, with a guaranteed



```
[Rpc(sources: RpcSources.InputAuthority, targets: RpcTargets.StateAuthority)]
1 reference
public void RPC_SetNick(string nick)
{
    Nick = nick;
}
```

Figure 5. RPC example of usage



```
[Networked]
12 references
private NetworkBool IsGrounded { get; set; }
```

Figure 6. Networked property example of usage

reception by other players. For instance, consider a door that can be opened or closed, RPC assures that all players witness the same door state change, a singular action not occurring frequently. On the other hand, continuous player movement necessitates updated and synchronized information, facilitated by Networked properties.

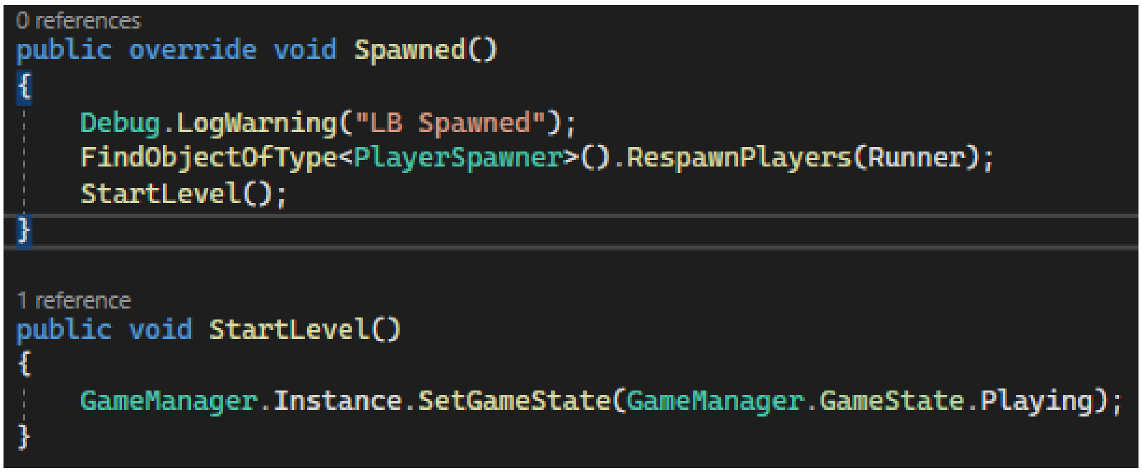
In game example of RPC usage was exchange of Nickname. Information about Nickname doesn't have to be passed from one player to another through game session, only once in the beginning of the game(see Figure 5).

The player's 'Grounded' networked property(see Figure 6) is consistently transmitted to other players, capturing the instance when the player makes contact with the ground and triggering subsequent animations or particle systems.

However, despite the straightforwardness of this mechanism, uncertainties arise due to internet instability or unforeseen PC issues across varied configurations. This unpredictability can result in the loss of certain player movement data. This predicament emphasizes the critical roles of Lag Compensation and Client-Side prediction, pivotal in mitigating issues that may affect gameplay due to such uncertainties.

4.4.2 Player spawning

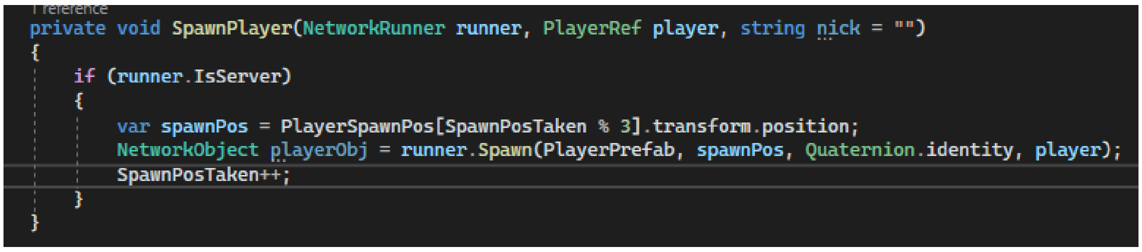
In typical game development within Unity without the use of additional SDKs or engines, the tool provided for object spawning is known as 'Instantiate()'. Unity's Instantiate method clones an existing game object, creating a duplicate within the scene. However, an issue arises as the spawned object remains unsynchronized across multiple players and is only visible locally. To resolve this synchronization challenge and ensure proper



```
0 references
public override void Spawned()
{
    Debug.LogWarning("LB Spawned");
    FindObjectOfType<PlayerSpawner>().RespawnPlayers(Runner);
    StartLevel();
}

1 reference
public void StartLevel()
{
    GameManager.Instance.SetGameState(GameManager.GameState.Playing);
}
```

Figure 7. Using NetworkBehaviour callbacks to trigger Spawned function



```
1 reference
private void SpawnPlayer(NetworkRunner runner, PlayerRef player, string nick = "")
{
    if (runner.IsServer)
    {
        var spawnPos = PlayerSpawnPos[SpawnPosTaken % 3].transform.position;
        NetworkObject playerObj = runner.Spawn(PlayerPrefab, spawnPos, Quaternion.identity, player);
        SpawnPosTaken++;
    }
}
```

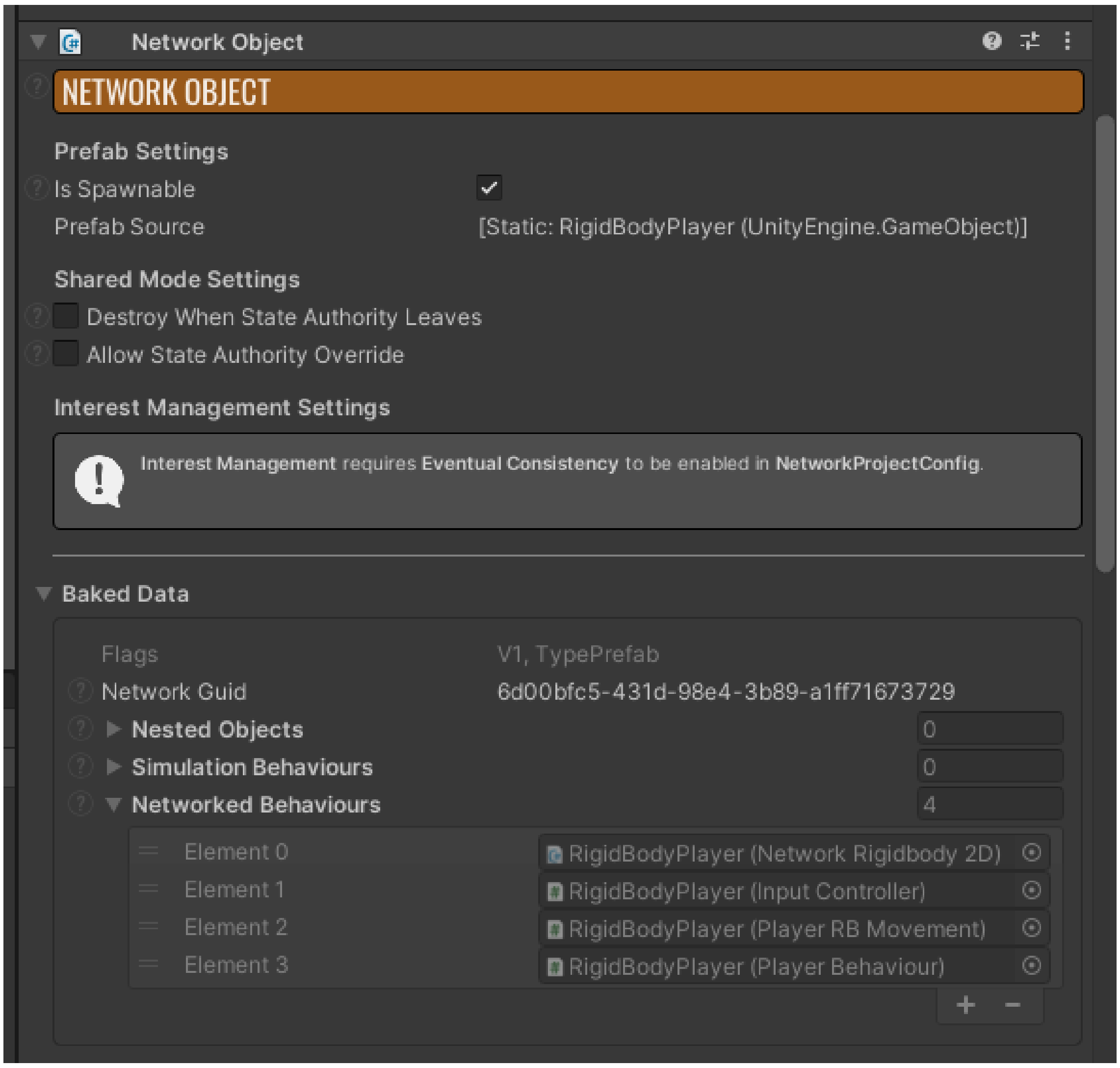
Figure 8. Using Spawn method to spawn a player

networking, Fusion recommends leveraging callbacks(see Figure 7) and a specific method known as ’Spawn()’(see Figure 8).

To synchronize a spawned object with a specific player, the ’Spawn()’ method requires a ’PlayerRef’ parameter, essentially representing the player’s ID within a multiplayer game. It’s important to note that while ’Instantiate()’ can only spawn ’UnityEngine.Object’, ’Spawn()’ exclusively accepts ’NetworkPrefabRef’. This ’NetworkPrefabRef’ essentially denotes a prefab with an attached ’Networked Object’ component(see Figure 9). The ’Networked Object’ encapsulates data inherited from ’NetworkedBehaviour’, such as ’InputController’ used for gathering and processing inputs, and ’NetworkRigidbody2D’ employed to detect collisions between dynamic objects (e.g., players) and static elements (e.g., the map grid).

4.4.3 Input Controller

To accurately retrieve and synchronize InputData, an additional callback is necessary. However, to distinguish between players’ inputs, each player should exclusively represent itself. This is where ’InputAuthority’ proves beneficial. ’InputAuthority’ is utilized to synchronize input exclusively from local player who has authority over an object


Network Object

NETWORK OBJECT

Prefab Settings

Is Spawnable ✓

Prefab Source [Static: RigidBodyPlayer (UnityEngine.GameObject)]

Shared Mode Settings

Destroy When State Authority Leaves

Allow State Authority Override

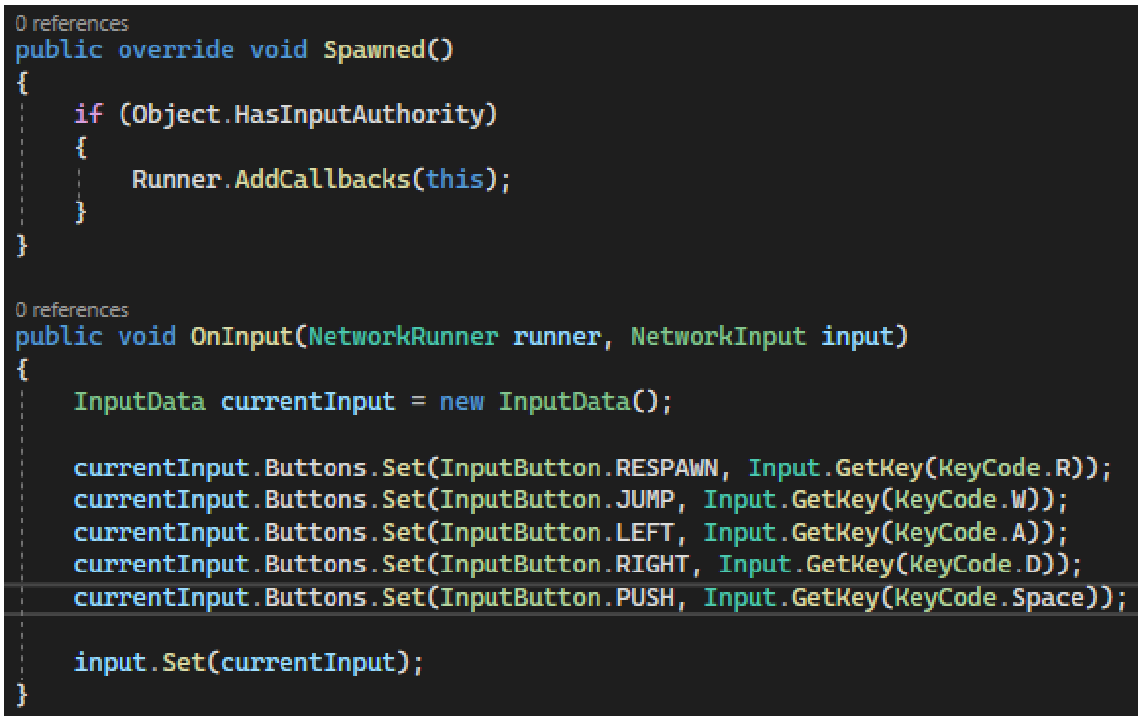
Interest Management Settings

Interest Management requires Eventual Consistency to be enabled in NetworkProjectConfig.

Baked Data

Flags	V1, TypePrefab
Network Guid	6d00bfc5-431d-98e4-3b89-a1ff71673729
Nested Objects	0
Simulation Behaviours	0
Networked Behaviours	4
Element 0	RigidBodyPlayer (Network Rigidbody 2D)
Element 1	RigidBodyPlayer (Input Controller)
Element 2	RigidBodyPlayer (Player RB Movement)
Element 3	RigidBodyPlayer (Player Behaviour)

Figure 9. Network Object component as a part of a prefab



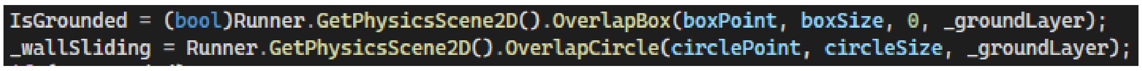
```
0 references
public override void Spawned()
{
    if (Object.HasInputAuthority)
    {
        Runner.AddCallbacks(this);
    }
}

0 references
public void OnInput(NetworkRunner runner, NetworkInput input)
{
    InputData currentInput = new InputData();

    currentInput.Buttons.Set(InputButton.RESPAWN, Input.GetKey(KeyCode.R));
    currentInput.Buttons.Set(InputButton.JUMP, Input.GetKey(KeyCode.W));
    currentInput.Buttons.Set(InputButton.LEFT, Input.GetKey(KeyCode.A));
    currentInput.Buttons.Set(InputButton.RIGHT, Input.GetKey(KeyCode.D));
    currentInput.Buttons.Set(InputButton.PUSH, Input.GetKey(KeyCode.Space));

    input.Set(currentInput);
}
```

Figure 10. Input Controller implementation



```
IsGrounded = (bool)Runner.GetPhysicsScene2D().OverlapBox(boxPoint, boxSize, 0, _groundLayer);
_wallSliding = Runner.GetPhysicsScene2D().OverlapCircle(circlePoint, circleSize, _groundLayer);
```

Figure 11. Static objects collision

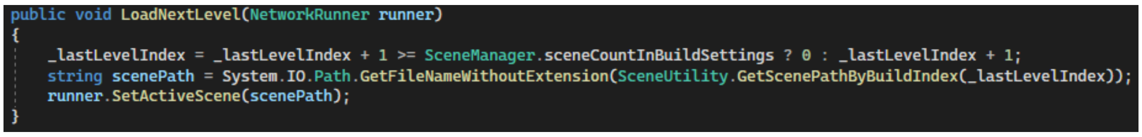
within Fusion. 'HasInputAuthority' determines whether 'Simulation.LocalPlayer' is the designated Input Authority for this exact network entity(see Figure 10).

4.4.4 Collision

For detecting collisions between dynamic and static objects, Fusion offers a specific tool to gather physics scene information, crucial for determining if a player is grounded (resting on the floor) or wall-sliding. Among the various tools within 'GetPhysicsScene2D()', we focus on two: 'OverlapBox()' and 'OverlapCircle()'. Both methods require parameters: the center of the box/circle that denotes the dynamic object, the size or radius of the dynamic object, and a 'layerMask' indicating the layer of static objects with which the dynamic object can collide(see Figure 11).

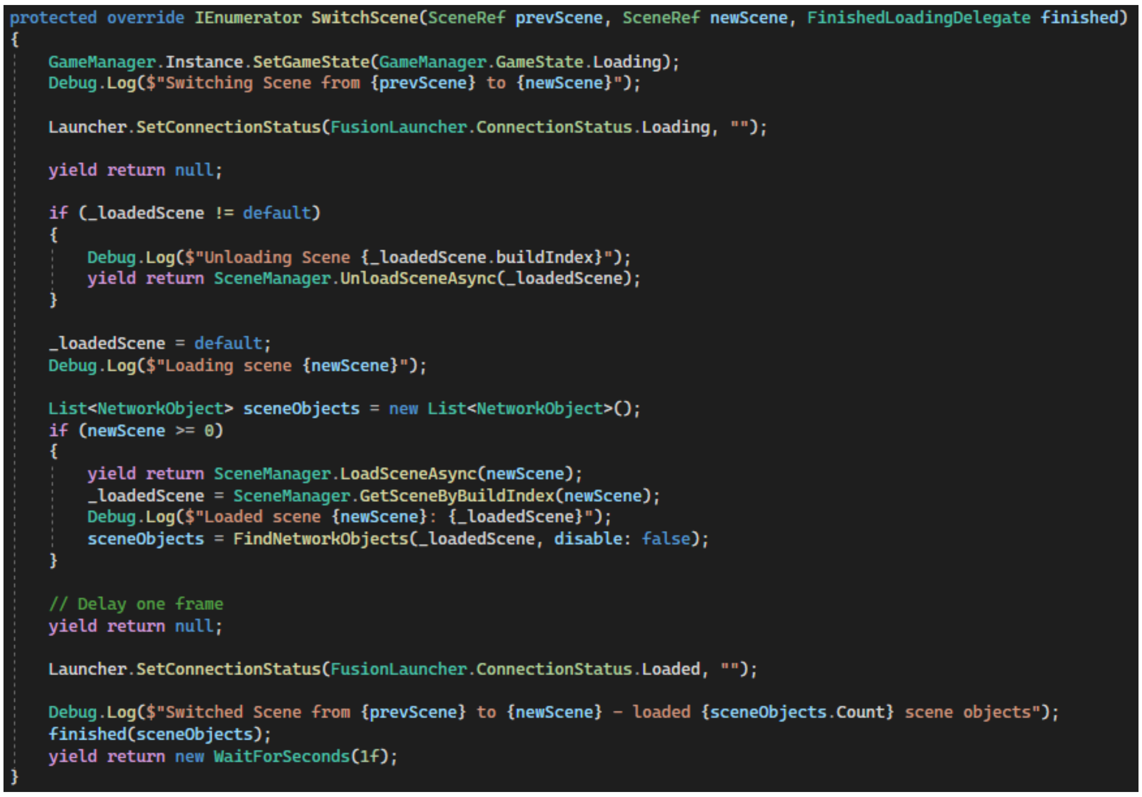
4.4.5 LevelManager

In a scenario similar to 'Instantiate()' and 'Spawn()', issues arose when the author attempted to transition all players from the 'Lobby' scene to the 'Game' scene. However, only the Host was able to change scenes, while the other players remained stuck in the



```
public void LoadNextLevel(NetworkRunner runner)
{
    _lastLevelIndex = _lastLevelIndex + 1 >= SceneManager.sceneCountInBuildSettings ? 0 : _lastLevelIndex + 1;
    string scenePath = System.IO.Path.GetFileNameWithoutExtension(SceneUtility.GetScenePathByBuildIndex(_lastLevelIndex));
    runner.SetActiveScene(scenePath);
}
```

Figure 12. LoadNextLevel function



```
protected override IEnumerator SwitchScene(SceneRef prevScene, SceneRef newScene, FinishedLoadingDelegate finished)
{
    GameManager.Instance.SetGameState(GameManager.GameState.Loading);
    Debug.Log($"Switching Scene from {prevScene} to {newScene}");

    Launcher.SetConnectionStatus(FusionLauncher.ConnectionStatus.Loading, "");

    yield return null;

    if (_loadedScene != default)
    {
        Debug.Log($"Unloading Scene {_loadedScene.buildIndex}");
        yield return SceneManager.UnloadSceneAsync(_loadedScene);
    }

    _loadedScene = default;
    Debug.Log($"Loading scene {newScene}");

    List<NetworkObject> sceneObjects = new List<NetworkObject>();
    if (newScene >= 0)
    {
        yield return SceneManager.LoadSceneAsync(newScene);
        _loadedScene = SceneManager.GetSceneByBuildIndex(newScene);
        Debug.Log($"Loaded scene {newScene}: {_loadedScene}");
        sceneObjects = FindNetworkObjects(_loadedScene, disable: false);
    }

    // Delay one frame
    yield return null;

    Launcher.SetConnectionStatus(FusionLauncher.ConnectionStatus.Loaded, "");

    Debug.Log($"Switched Scene from {prevScene} to {newScene} - loaded {sceneObjects.Count} scene objects");
    finished(sceneObjects);
    yield return new WaitForSeconds(1f);
}
```

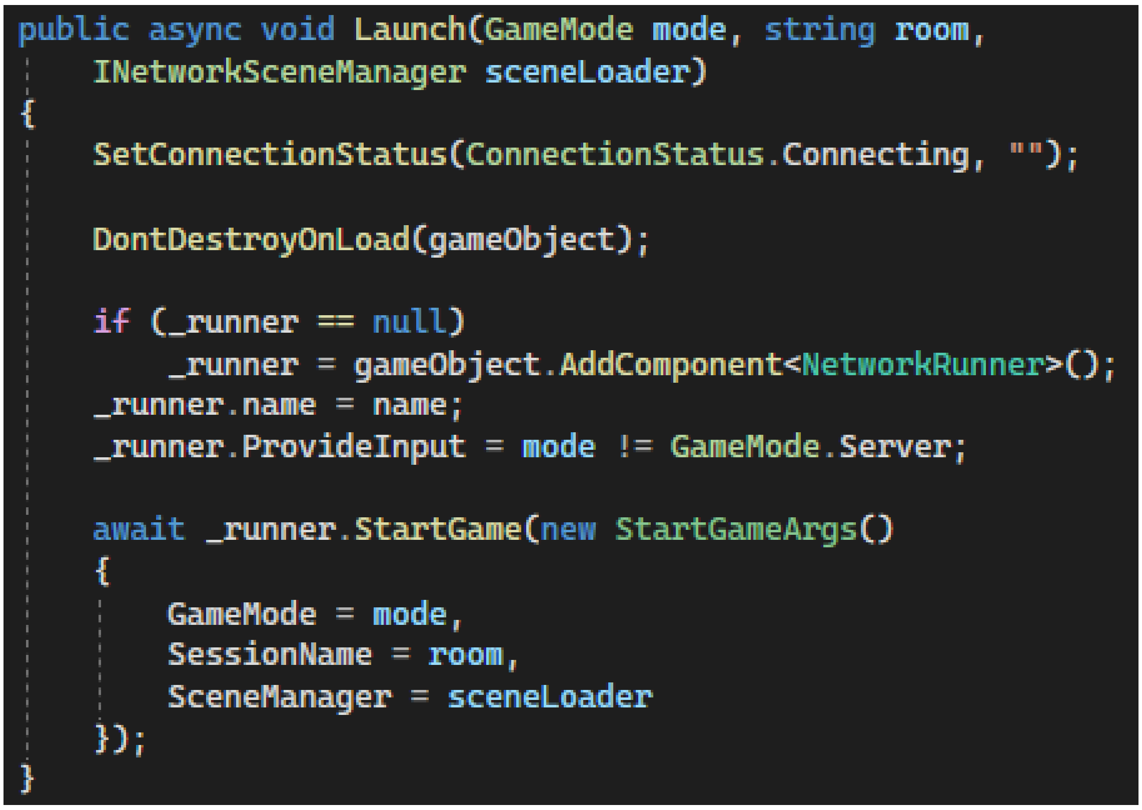
Figure 13. SwitchScene function

lobby. Moreover, no players were spawned or visible in the new scene. This issue was attributed to Fusion's specialized method responsible for scene transitioning across all connected players. The 'SetActiveScene' function within NetworkRunner initiates the scene-changing process for all players(see Figure 12).

To load and synchronize essential data like the Map, ensuring its visibility and interactivity for all players, the NetworkSceneManagerBase interface offers an overridden method designed for two primary purposes. Firstly, it prepares NetworkedObjects to be synchronized in the scene, and secondly, it supports the animation during the scene-switching process(see Figure 13).

4.4.6 Game Launching

In Unity, starting a game often requires no more than just adjusting the UI and activating objects or changing a scene. However, for a multiplayer game, this isn't sufficient. This



```
public async void Launch(GameMode mode, string room,
    INetworkSceneManager sceneLoader)
{
    SetConnectionStatus(ConnectionStatus.Connecting, "");

    DontDestroyOnLoad(gameObject);

    if (_runner == null)
        _runner = gameObject.AddComponent<NetworkRunner>();
    _runner.name = name;
    _runner.ProvideInput = mode != GameMode.Server;

    await _runner.StartGame(new StartGameArgs()
    {
        GameMode = mode,
        SessionName = room,
        SceneManager = sceneLoader
    });
}
```

Figure 14. Launch function that starts a game session

is where Fusion's 'NetworkRunner' proves its worth. When the 'StartGame()' method is invoked from a 'NetworkRunner' instance, it initiates the creation of a Peer, which subsequently joins or generates a room according to the specifications provided in the 'StartGameArgs' argument(see Figure 14). In Multi-Peer Mode, all Scene Objects and Spawned Objects are organized as children of a dedicated 'GameObject' and are included in the associated 'PhysicsScene/PhysicsScene2D' linked to that Runner. It's important to note that a 'NetworkRunner' instance can only be used once. Upon disconnection from a game session or a failed connection, it should be destroyed, necessitating the creation of a new 'NetworkRunner' instance for any subsequent game sessions. During startup, the 'NetworkRunner' component identifies and registers all 'SimulationBehaviour' components among its child elements. These components receive callbacks such as 'FixedUpdateNetwork()' and 'Render()'. Moreover, the 'NetworkRunner' locates all 'INetworkRunnerCallbacks' interfaces within the Components on the game objects and registers them for corresponding callbacks.

5. Conclusion, Analysis and future development

5.1 Benefits and drawback of using Fusion-SDK

5.1.1 Advantages

Fusion offers several handy tools that this bachelor thesis covers, enabling novices like the author to craft a functional game session within mere hours. Leveraging a range of built-in functionalities such as 'Static Collision Detection', 'Input Handling', 'Game Session Launching', 'Player Spawning' and many others. Fusion eliminates the necessity of manual server configuration or establishing player connections. Moreover, Fusion conveniently provides the most essential callbacks right out of the box

5.1.2 Disadvantages

The Fusion community is relatively small, resulting in a scarcity of tutorials or implementation examples for its features. The extensive documentation poses a challenge for newcomers seeking specific tools. Without guidance from a mentor, achieving similar results could take significantly longer. Moreover, mastering more advanced functionalities such as 'Lag Compensation,' 'Animation Synchronization,' and 'Client-Side Prediction' demands a robust understanding of the Fusion-SDK.

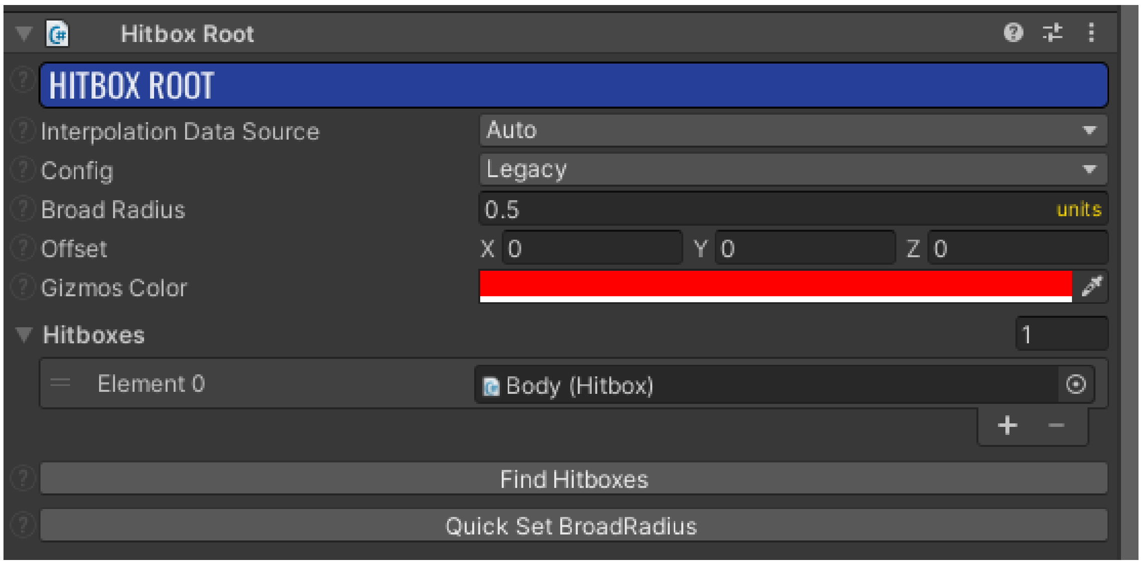
5.2 Future development

5.2.1 Lag compensation

The primary objective to complete the game involves implementing a lag compensation solution for dynamic object collision detection. This requires a shift from detecting collisions among 'Colliders2D' to utilizing the 'Hitboxes'(see Figure 14 and 16) tool provided by Fusion-SDK for effective lag compensation.

5.2.2 Movement update

Further development of the game is supposed to enhance the game's pace. Removing speed limit and updating acceleration system allows seasoned players to challenge their


Hitbox Root

HITBOX ROOT

Interpolation Data Source: Auto

Config: Legacy

Broad Radius: 0.5 units

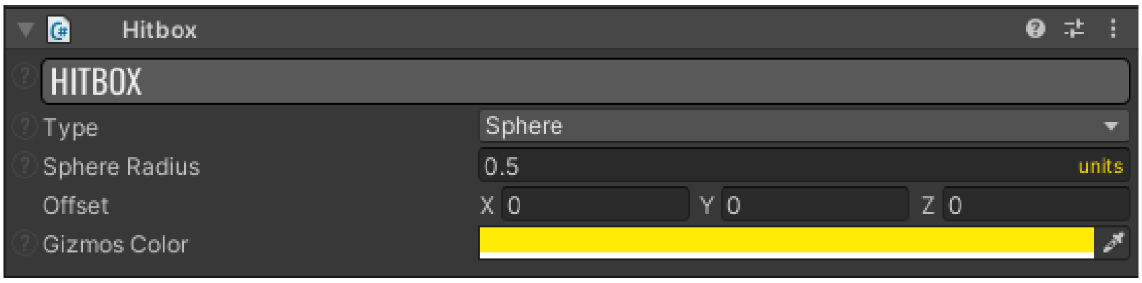
Offset: X 0 Y 0 Z 0

Gizmos Color

Hitboxes: 1

Element 0: Body (Hitbox)

Figure 15. HitboxRoot Component


Hitbox

HITBOX

Type: Sphere

Sphere Radius: 0.5 units

Offset: X 0 Y 0 Z 0

Gizmos Color

Figure 16. Hitbox Component

skills without any limitations. Adding short boost with brief cool-down, can widen the variety of strategies in game play.

5.2.3 Tournament-SDK implementation

As a part of initial plan, once the game itself is considered complete, Tournament-SDK implementation is scheduled to be integrated into the game. Tournament-SDK provides a seamless solution for integrating a tournament system into the game, requiring minor changes or additions to the existing code.

5.2.4 Map update

Further iteration of map update is aiming to expand playground to give player more engaging experience. Current map's enclosed state restricts increased speed and limits maneuverability.

5.2.5 Interest management

Upcoming map expansion in the game is more than just engaging playground, it aims to showcase the usage of Interest Management.

Interest Management involves data culling mechanisms such as: Object Interest and Behaviour Interest, which limit the replication of specific Network Objects and Behaviours from the Server to designated Player Clients. Object Interest settings determine how Player interest in an Object is established, affecting the receipt of updates(such as Networked Properties and RPCs) for that Object from the Server peer.

There are three Object Interest options to choose from:

Area of Interest: Players with an overlapping AOI region with a Networked Object's position in space will be interested in the Object.

Global: All Players receive updates for a Network Object.

Explicit: Only Players explicitly marked using 'SetPlayerAlwaysInterested()' will express interest in a Network Object.

6. Summary

The objective of creating a multiplayer 2D game using Unity and Photon's Engine Fusion-SDK was to offer a comprehensive example of Fusion implementation, showcasing fundamental multiplayer game-building tools. The game effectively demonstrates the usage of Networked Properties, Remote Procedure Calls, Input Authority and other key features.

The utilization of Fusion-SDK for crafting a 2D multiplayer Unity game brought both advantages and challenges to the table. On the positive side, it effectively resolved the intricate issue of server hosting inherent in multiplayer games, thanks to Photon's comprehensive coverage in this area. The diverse set of tools offered by Fusion-SDK facilitated a swift initiation into development, enabling the achievement of initial goals like establishing a lobby and commencing synchronized gameplay.

However, accompanying these benefits were notable drawbacks. Despite the acceleration in the initial development phase afforded by Fusion-SDK, the author's lack of experience in utilizing the SDK and crafting multiplayer games led to increased development complexity and time consumption as more features, notably "Lag Compensation" and "Animation Synchronization" were incorporated into the game.

GimmeBreak company, the project's clients, regard the development progress as satisfactory, aligning with the initial time estimate of 7-9 months. To further enhance development, a closer collaboration between the author and mentor is planned. This entails detailed feedback and guidance for effectively implementing new advanced features like Lag Compensation and Interest Management.

The initial iteration of the game(see Figure 18) lacked a clear goal or objective, leading to a transient interest among players for about an hour. However, based on received feedback, the author improved existing features and introduced new elements. One significant addition, advised by testers, was the "Leap" mechanic, offering a short burst of speed along the X-axis to intensify the gameplay pace.

The game's focus shifted from randomly pushing players out of the game borders to a new mechanic where time accumulates only when a player stands on the central panel(see Figure 19). Each player can view only their own timer, ensuring engagement even if one player gains a substantial advantage in time. This adjustment aimed to maintain

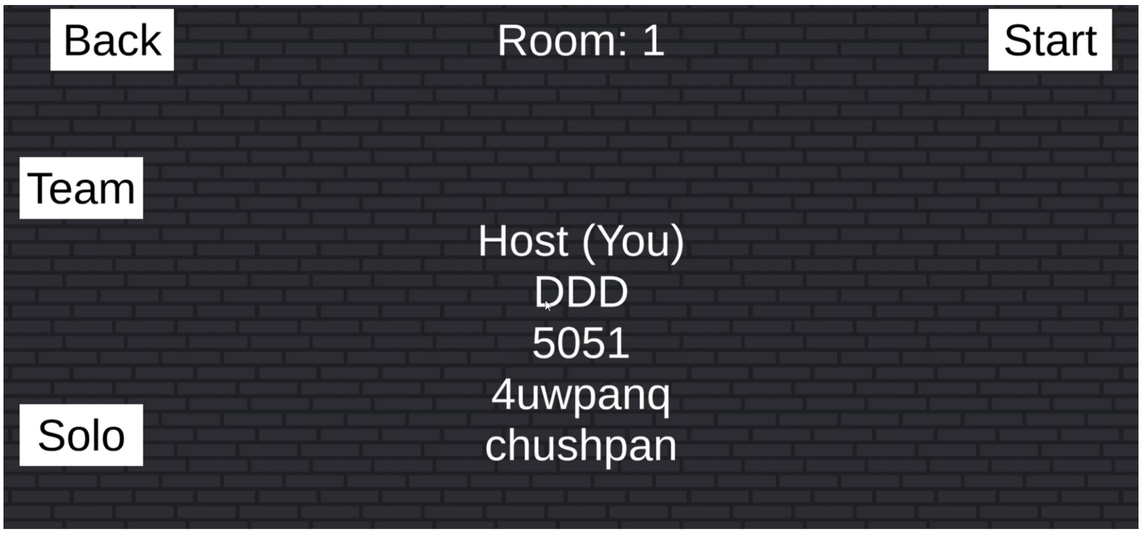


Figure 17. Lobby

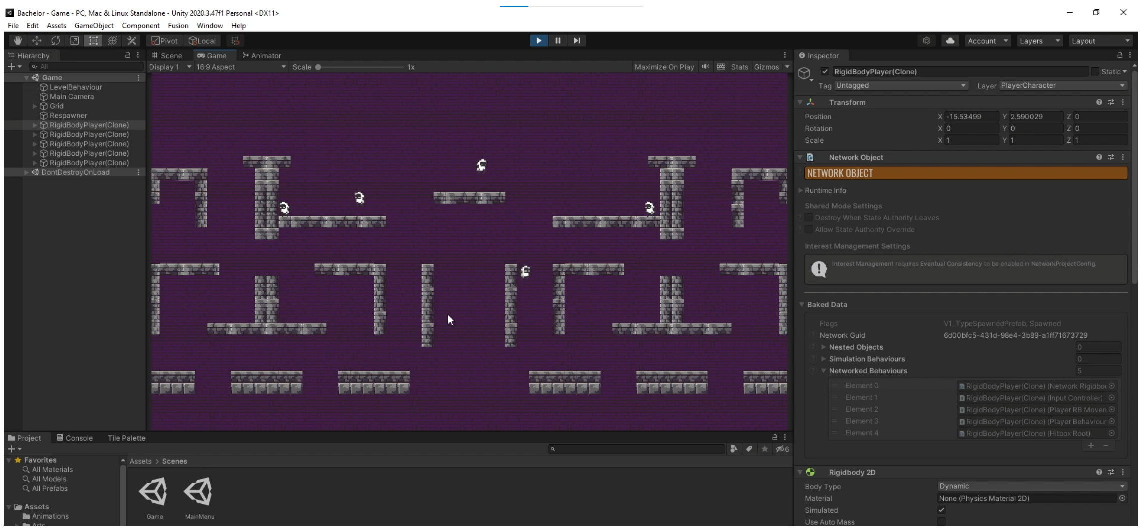


Figure 18. First play test with people

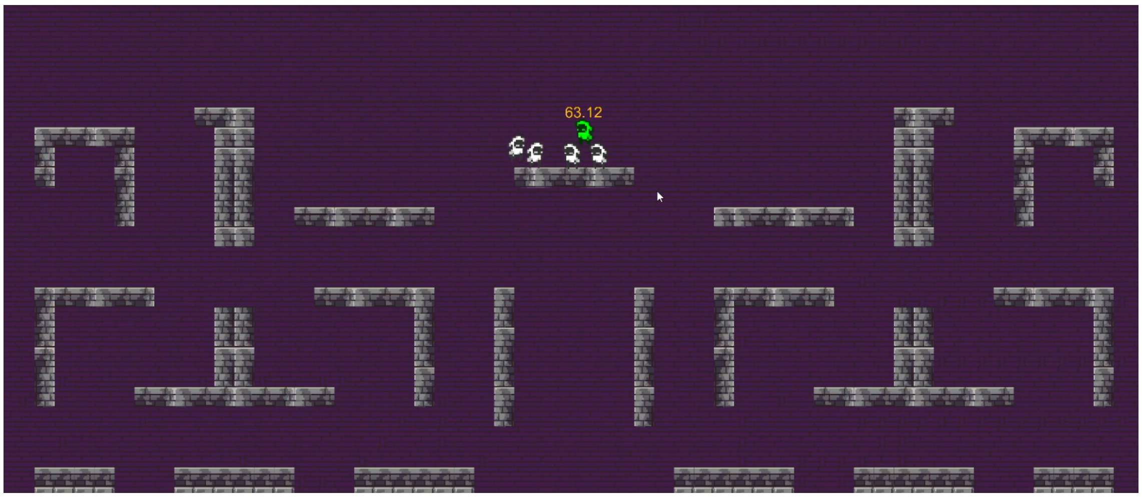


Figure 19. Central Platform

involvement among all players throughout the gameplay.

Despite the hurdles encountered in development, the author firmly believes that investing time in learning Fusion-SDK is a fair endeavor. This belief is based on the SDK's potential for multiplayer game development and its ability to alleviate concerns about such crucial aspects as server maintenance and player connections.

References

[1] Unity. *Unity Multiplayer Networking*. URL: https://docs-multiplayer.unity3d.com/.

[2] Photon. *Photon Engine*. URL: https://www.photonengine.com.

[3] Photon. *Fusion*. URL: https://www.photonengine.com/fusion.

[4] Mirror. *Mirror Networking*. URL: https://github.com/MirrorNetworking/Mirror.

[5] FishNet. *FishNet vs Mirror*. URL: https://fish-networking.gitbook.io/docs/manual/general/performance/fish-networking-vs-mirror.

[6] Unity. *Free Networking Solutions Comparison*. URL: https://forum.unity.com/threads/updated-free-networking-solution-comparison-chart.1359775/.

[7] Unity. *Which Networking Solutions Do You Use And Why*. URL: https://forum.unity.com/threads/which-networking-solutions-do-you-use-and-why.1345262/.

[8] Placeholder-software. *Choosing A Network*. URL: https://placeholder-software.co.uk/dissonance/docs/Basics/Choosing-A-Network.html.

[9] Mirror. *Mirror Networking*. URL: https://mirror-networking.gitbook.io/docs/.

[10] JesusLuvsYooh. *TransportResults1Feb2021*. URL: https://docs.google.com/spreadsheets/d/12PGl9cgaH52VKPnkTA5yt9contgnD6zYMdJOjj4r9So/edit#gid=838356447.

[11] Unity community. *Unity Networking Solutions*. URL: https://docs.google.com/spreadsheets/d/1Bj5uLdnxZYlJykBg3Qd9BNOtvE8sp1ZQ4EgX1sI0RFA/edit#gid=233715429.

[12] Author. *ProjectGitlab*. URL: https://gitlab.cs.ttu.ee/jegkuz/bachelor.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student’s application for restriction on access to the graduation thesis that has been signed by the school’s dean, except in case of the university’s right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Gitlab

Code of the project is available in gitlab[12].