Thang Dang Duc D4988

MULTIPLAYER SOLUTION FOR 3D UNITY GAME PROTOTYPE USING UNITY NETCODE

Bachelor's thesis

Information Technology

T5616SN

2023

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland
University of Applied Sciences

Degree title	Information Technology
Author(s)	Thang Dang Duc
Thesis title	Multiplayer solution for 3D Unity game prototype using Unity Netcode
Commissioned by	
Year	2023
Pages	41 pages
Supervisor(s)	Juutilainen Matti

ABSTRACT

Multiplayer games are an attractive market for game developers, taking up a sizable portion of the market. However, adding multiplayer features to a game increases the scope of the project significantly, which deters potential ideas from being formed.

In 2022, Unity Netcode became production-ready, reintroducing a native solution for multiplayer game development and easing out the process of multiplayer game creation. This thesis aimed to explore the process of prototyping a multiplayer 3D game using Unity Netcode, a high-level networking library built for Unity that enables sending GameObjects and world data across a networking session.

In the thesis, the author implemented the procedure for making a simple game prototype named “Tagteam” and evaluates its feasibility and effectiveness from the author’s working experience and perspective as a game designer.

The study showed that while Unity Netcode eased out the development process, it did not outperform other tools significantly in terms of performance and development cost.

Keywords: documentation, model, thesis, report writing

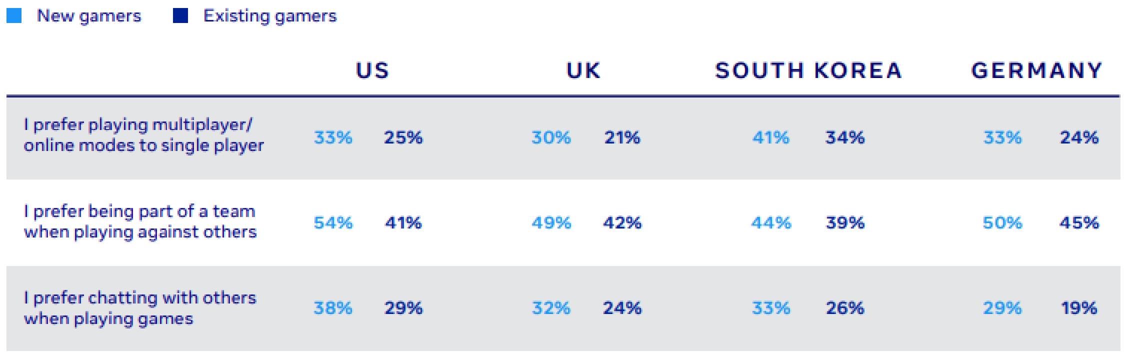
CONTENTS

1 INTRODUCTION

In 2021, the games market generated total revenues of 180.3 billion USD, up +1.4% over the year 2020. Despite the COVID-19 pandemic's seemingly unstoppable surge in players, 3 billion players worldwide were responsible for this revenue, which was a +5.3% increase over 2020. This slight increase in both revenue and player number suggested that while the surge of players could have been accounted for by the pandemic, the players were staying even after the peak of the situation (70% of people report spending more time on mobile devices in 2021). This resulted in the games market no longer being considered the hit-driven business it once was. (Tom 2021.)

Due to quarantine becoming a common practice across the globe, players have been turning to multiplayer games in order to connect with each other. 60% of survey respondents stated that they were playing more multiplayer games during the pandemic in order to replace the face-to-face interaction that was, in many places, severely restricted.

It could be inferred from Facebook's game marketing insights for 2021 (Figure 1) that more than a quarter of players around the world prefer multiplayer or online game modes to their single-player counterparts (Facebook, 2021). The table below shows that new gamers are more likely to cooperate with others in the world of gaming.



■ New gamers ■ Existing gamers

	US		UK		SOUTH KOREA		GERMANY	
I prefer playing multiplayer/ online modes to single player	33%	25%	30%	21%	41%	34%	33%	24%
I prefer being part of a team when playing against others	54%	41%	49%	42%	44%	39%	50%	45%
I prefer chatting with others when playing games	38%	29%	32%	24%	33%	26%	29%	19%

Figure 1. Survey on multiplayer related games

Many multiplayer games have received enormous attention compared to their cost of creation, leading to unprecedented cases of success over the last two years. Some notable games include "Fall Guys,", "Among Us,", "It Takes Two," and "Genshin Impact". In such a rapidly changing industry that is booming in

potential, one is brought back to the ever-going question within the game industry: “How to quickly prototype products, especially multiplayer games?”
Game prototyping is a vital phase in the game development process that involves creating a method to test the concept of the game. The prototyping period usually lasts between 1 and 30 days, with a preferred length of about 2 weeks for rough game ideas. One of the principles of prototyping is minimizing the cost of time and human resources so game studios can test the game’s effectiveness before spending their budget on developing a full game. This means that the principle of prototyping is more about efficiency than the product’s completeness.
In the author’s workplace, a Gameloft studio, despite countless prototypes being made every year, not many of them support multiplayer. One of the reasons for this is that multiplayer games introduce tremendous complexity compared to their single-player counterparts, rendering the prototyping process slow, heavy, and unworthy of delivery. The team would need to take care of both the game’s client mechanics, server logic, and assets. However, if multiplayer games can be prototyped in such a short period of time, they can bring valuable payoffs like opening up potential games to a large chunk of the market or easing the testing process (testers can simulate AI cases).
In this thesis, the author will explore approaches to ease the process of multiplayer game prototyping. The main technologies used include Unity Engine, one of the most commonly used game engines at the time of writing, and Fusion by Photon, a reliable and versatile network engine on the market. The thesis aims to analyze the impact of these tools and discuss their usability through the process of creating a Unity game prototype. The use of some other readily available assets will be discussed. However, these assets are only considered helpers for almost all genres of prototypes.

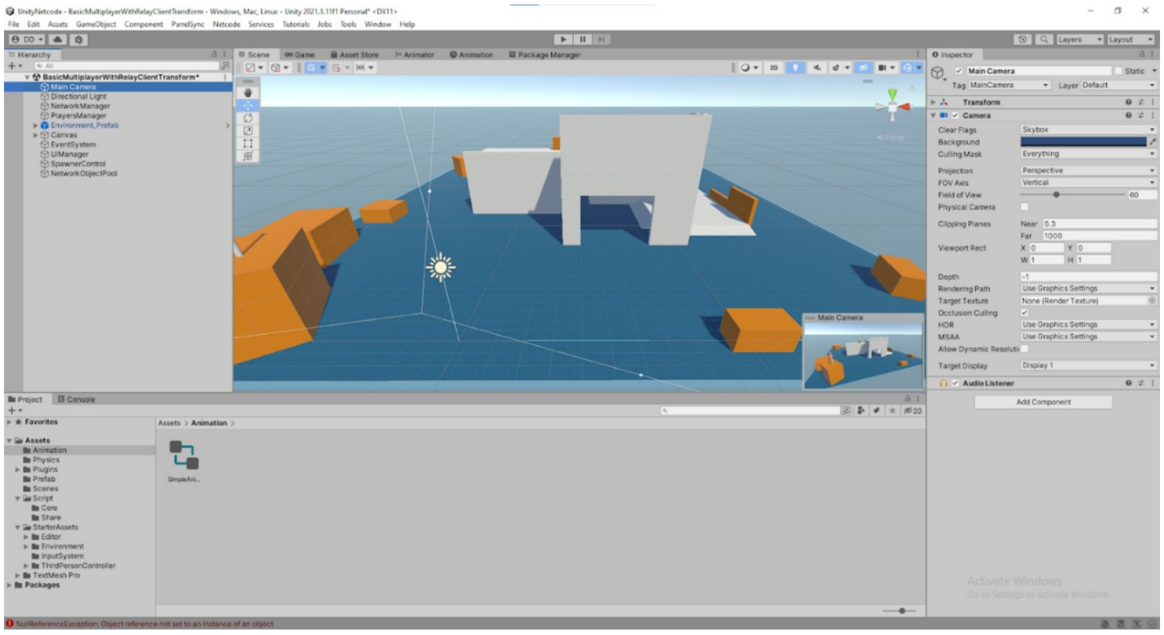
2 UNITY

The Unity engine was initially released in 2005 by the company Unity Technologies from a small Copenhagen apartment, one year after their foundation.
“Unity wants to be the 3D operating system of the world,” says Sylvio Drouin, VP of the Unity Labs R&D team. (Eric 2019.)

With such a vision, Unity has recently been widely used for 3D design and simulation in multiple industries, including video game creation. In 2019, nearly half of the world's games were built with Unity, according to the company's CEO. (Romain 2018.)
Unity's growth is a case study of Clayton Christensen's theory of disruptive innovation. Unity's main rival in the market – Unreal Engine caters for AAA developers who create highly complex and detailed games or 3D representations. While Unity's main target market was smaller companies or independent developers who need a simpler game engine with lower price range. However, Unity has recently appealed to even professional game studios, powering the creation of market leading games like "Angry birds 2", "Hearth stone", "Monument Valley", "Ori and the Blind Forest", etc. (Clayton et al. 2015.)
This success has some of the same reasons why the author chose the engine for this project. Unity supports over 20 different platforms for game deployment. It offers a complete toolkit for building games, from graphics and audio to level-building and teamwork. Unity is relatively easy to use with C#, compared to Unreal Engine with C++. Unity is readily available thanks to its free plan.
A strong community is built around this engine, creating thousands of assets on Unity's asset store at https://assetstore.unity.com/. Supports from the community can be found in abundance on Unity's forum at https://forum.unity.com/. Unity Technology also provides its own support for the community with multiple tutorials, project templates, and, more recently, classes.
However, as of May 2022, Unity does not provide an official networking solution. The community has recommended against and deprecated their initial plan using MLAPI. Some other options include Fusion networking or Mirror networking.

2.1.1 Unity Editor

The Unity editor (Figure 2) is a modular set of interfaces serving multiple purposes and, in turn, shaping the game development workflow. Some default interfaces are: object hierarchy, project explorer, console, inspector, game view, scene view, etc. These interfaces are treated like windows, which can be moved around, docked, grouped, and detached depending on the user's needs.



Main Camera
Tag: MainCamera
Layer: Default
Transform
Camera
Clear Flags: Skybox
Background
Culling Mask: Everything
Projection: Perspective
FOV Axis: Vertical
Field of View: 60
Physical Camera
Clipping Planes: Near 0.3, Far 1000
Viewport Rect: X 0, Y 0, W 1, H 1
Depth: -1
Rendering Path: Use Graphics Settings
Target Texture: None (Render Texture)
Occlusion Culling: ✓
HDR: Use Graphics Settings
MSAA: Use Graphics Settings
Allow Dynamic Resoluti
Target Display: Display 1
Audio Listener

NullReferenceException: Object reference not set to an instance of an object

Figure 2. Unity’s Editor view

Each layout can be customized and later accessed through the “Layout” dropdown menu at the top right of the editor screen. This chapter will go into the method of utilizing said layout for the purpose of prototyping a 3D game.

2.1.2 Scene view

The scene view (Figure 3) is one of the most important views in Unity. It is an interactive visual interface for the game scene, providing features like 3D maneuvering views or drag-and-drop object manipulation. The user can pan the camera view by dragging the mouse wheel, rotate the view by dragging the right mouse button, zoom in and out by scrolling the scroll wheel, or move around the area like they would in a first-person shooter video game using the arrow buttons on the keyboard.

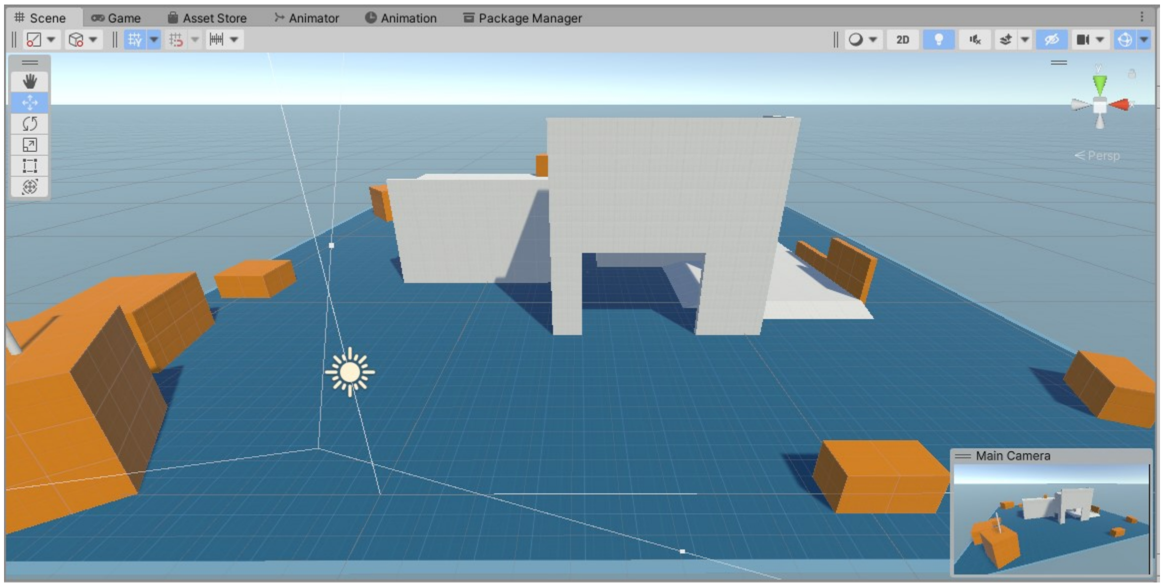


Figure 3. Unity's Scene view

The scene view also supports the option to change the viewing perspective by clicking on the colored handles (Figure 4) of each dimension.

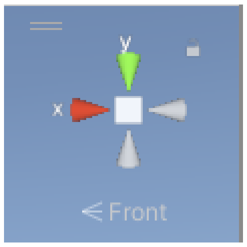
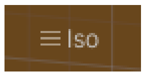
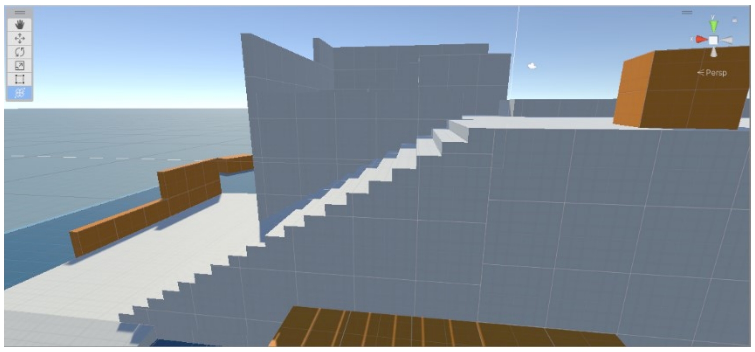
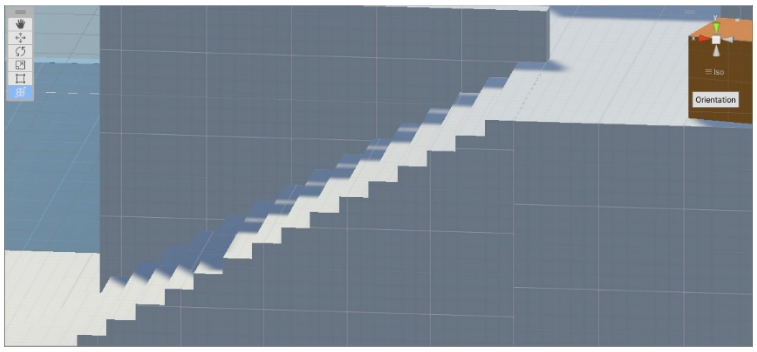


Figure 4. Unity's dimensional view control

Users can switch between the Isometric render mode or Perspective render mode to increase precision in object modification (Table 1).

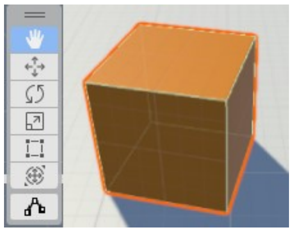
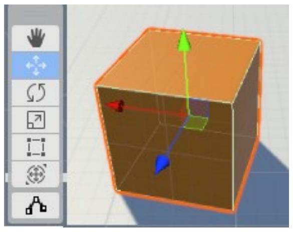
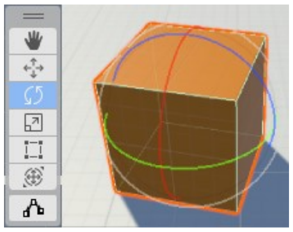
Table 1.Isometric and Perspective render mode

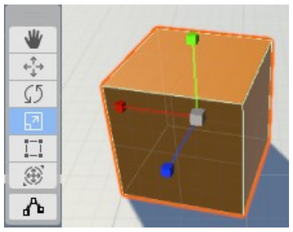
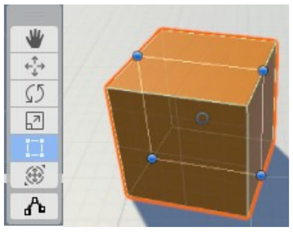
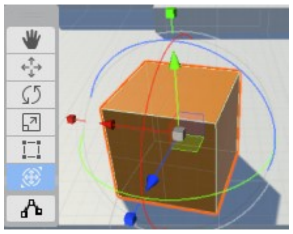
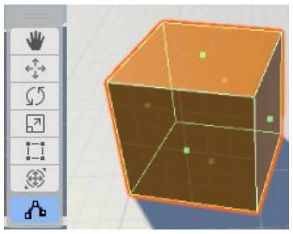
View mode	Button	Result
Perspective	 ≡Iso	

Isometric		

On the left side of the view, a list of object-manipulating actions can be chosen to modify the game scene. These tools are listed in Table 2.

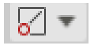
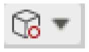
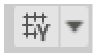
Table 2. Unity object-manipulation tools

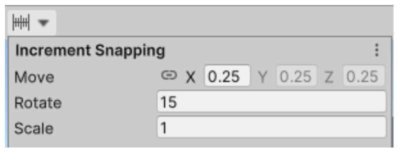
Tool name	Illustration	Interaction	Hotkey
View tool		Drag to move the camera view, click on objects to select them in the hierarchy	Q
Move tool		Each dimension (x, y, and z) is represented by an arrow-shaped handle of different colors. Drag on individual handles to slide the object among one dimension, or drag the squares in the pivot of the object to move them along a plane.	W
Rotate tool		Each dimension (x, y, and z) is represented by a round-shaped handle of different colors. Drag on individual handles to rotate the object along the respective plane.	E

Scale tool		Each dimension (x, y, and z) is represented by a handle of a different color. Drag on individual handles to scale the object along the respective dimension.	R
Rect tool		The object is represented by a rectangle shape. Drag the vertices or edges to resize and reposition the object at the same time.	T
Transform tool		This tool is a combination of moving, rotating, and scaling tools.	Y
Edit bounding volume		The collider of the object is represented by one of Unity's basic shapes. Drag the drops to scale and move the object's collider at the same time.	

Another feature of the scene view is the toolbar, which offers various other tools, as shown in the Table3.

Table 3. Miscellaneous tools

Tool name	Illustration	Interaction
Pivot		Choose how to pivot objects.
Global/local rotation		Pivot objects locally or globally.
Grid visibility		Toggle the visibility of the grid.

Snap to grid		Toggle snapping to grid.
Snap increment	 Increment Snapping Move X 0.25 Y 0.25 Z 0.25 Rotate 15 Scale 1	Choose how the object is snapped into the 3D world.
Shading		Choose the shading mode,
2D/3D view	2D	Toggle 2D/3D view mode
Sound		Toggle sound on/off
Sky box		Choose how skybox is presented
Camera		Change various camera settings
Gizmo		Toggle to show/hide gizmos

If the main camera is selected in the hierarchy view, a small representation of what the camera is currently capturing is shown in a float box at the bottom right of the screen by default (Figure 5).

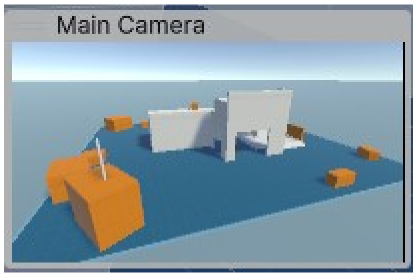


Figure 5. Main camera view

2.1.3 Hierarchy view

The hierarchy in the figure below shows a tree of objects within the scene called “BasicMultiplayerWithRelayClientTransform”. Everything listed in this hierarchy is a GameObject (Figure 6). The order in which objects are listed will be used to determine the render order when the game is built, meaning an object that is higher in the hierarchy will be shown on closest to the player's screen. However, this order should only be used as a reference and not the main way to order objects, as it can lead to some rendering problems.

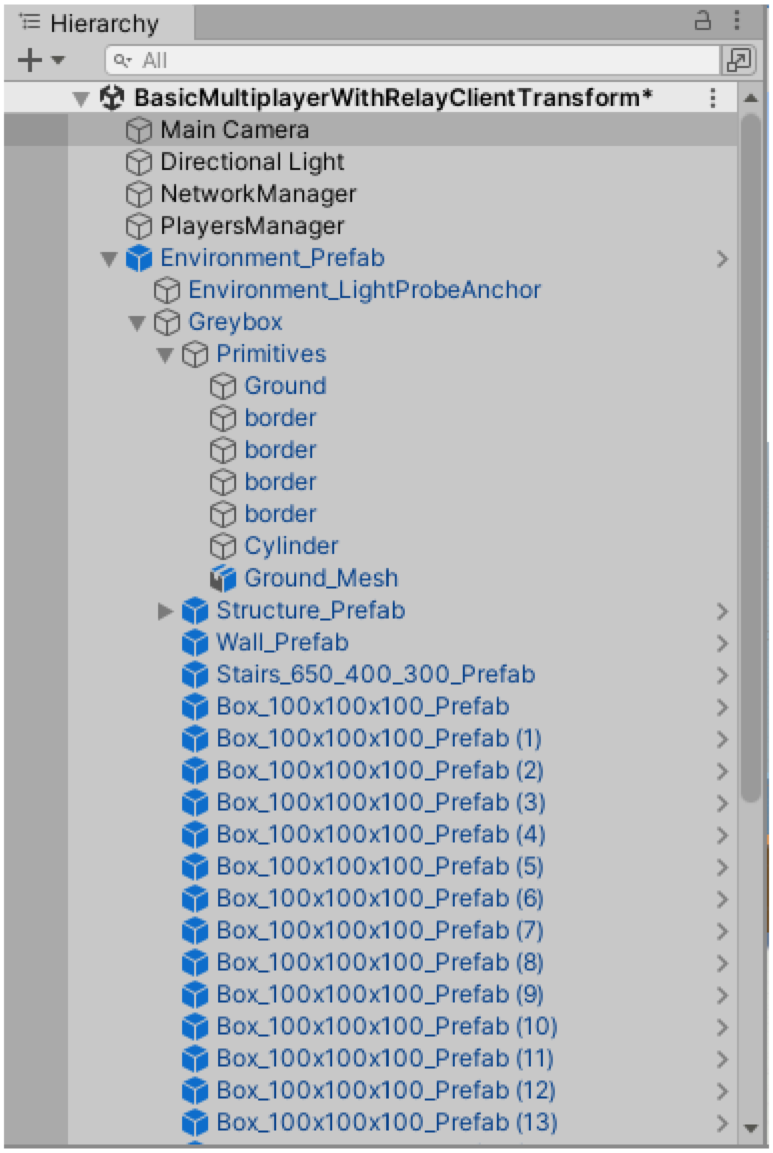

Hierarchy
All
BasicMultiplayerWithRelayClientTransform*
Main Camera
Directional Light
NetworkManager
PlayersManager
Environment_Prefab
Environment_LightProbeAnchor
Greybox
Primitives
Ground
border
border
border
border
Cylinder
Ground_Mesh
Structure_Prefab
Wall_Prefab
Stairs_650_400_300_Prefab
Box_100x100x100_Prefab
Box_100x100x100_Prefab (1)
Box_100x100x100_Prefab (2)
Box_100x100x100_Prefab (3)
Box_100x100x100_Prefab (4)
Box_100x100x100_Prefab (5)
Box_100x100x100_Prefab (6)
Box_100x100x100_Prefab (7)
Box_100x100x100_Prefab (8)
Box_100x100x100_Prefab (9)
Box_100x100x100_Prefab (10)
Box_100x100x100_Prefab (11)
Box_100x100x100_Prefab (12)
Box_100x100x100_Prefab (13)

Figure 6. Hierarchy window

Objects are shown in relation to each other. Some objects are nested in other objects. In Unity, objects can be nested in multiple layers. When an object is nested in another object, it is called a child of that parent object. Parenting objects offer a more organized human view of the hierarchy. In addition, it also introduces dependencies and relationships between different objects. The children's scale and position change in regards to their parents, while the reverse does not hold true.

Here, objects can be created or spawned from the asset folder. However, deleting or moving an object within the hierarchy view does not modify it in the game's asset folder.

2.1.4 Game view

The game view (Figure 7) automatically shows up when the user presses the Play button to test the game. It shows what is actually seen in the final product of the game, in contrast to the free view and edits offered by the scene view.

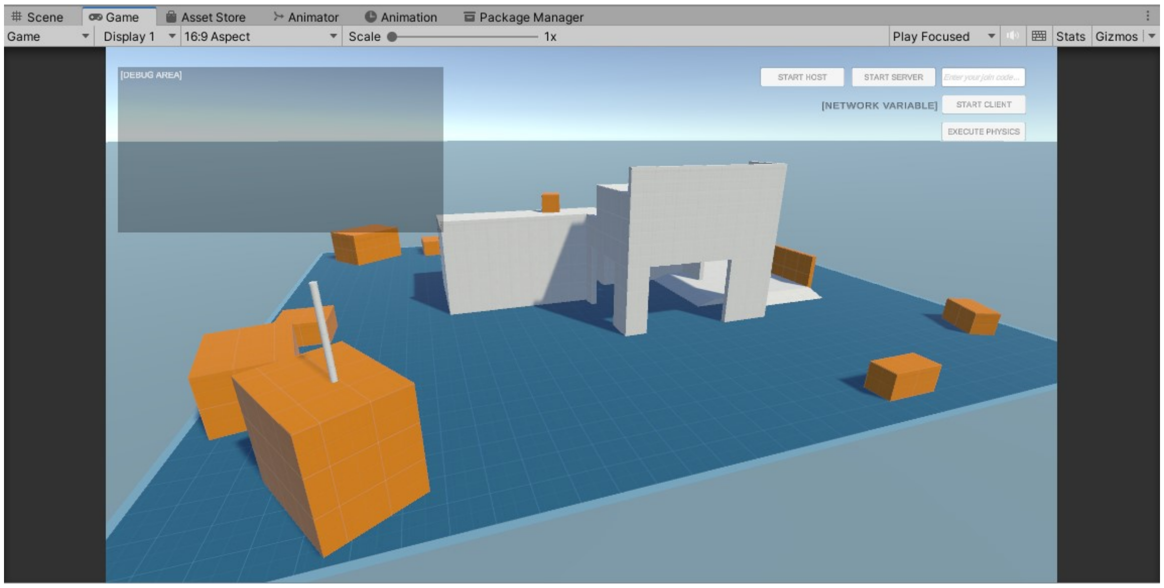


Figure 7. Game view

This view offers a drop-down menu (Figure 8) that shows how the game is shown in various screen sizes or the user's custom screen size. This increases the speed of testing for UI responsiveness on multiple devices or platforms.

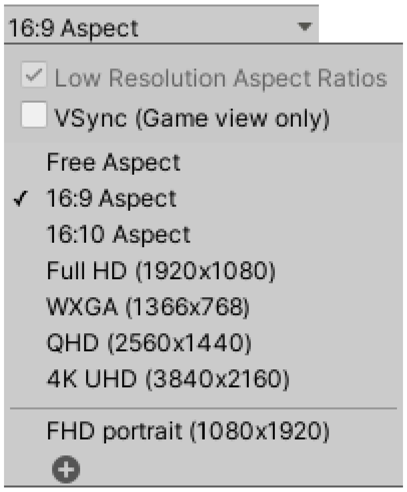


Figure 8. Screen Aspects dropdown menu

2.1.5 Project explorer

The project explorer screen holds the view of every file within the scope of a Unity project. It displays everything in a tree structure, similar to how files are organized on the hard drive. To navigate this interface, the user can click on the hierarchy on the left side and choose the objects on the right side of the interface. Files can be moved, renamed, or deleted in the right-click drop-down menu.

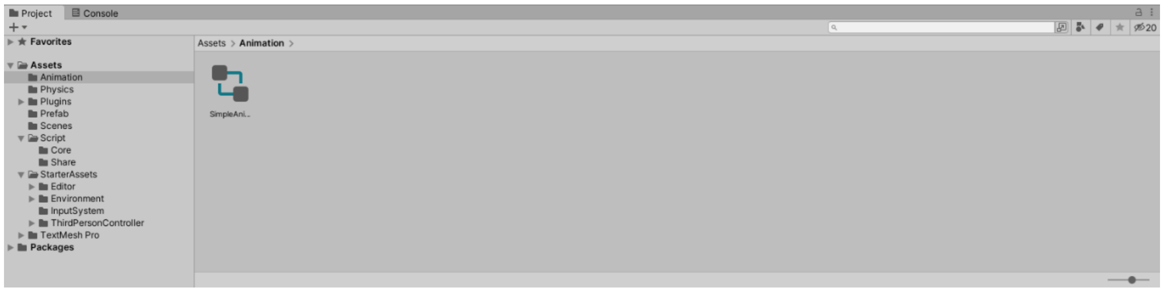


Figure 9. Unity's file explorer

This interface offers more features than the regular Windows file explorer. For example, dragging a GameObject from the hierarchy interface onto the project explorer will automatically create a prefab of that object. And dragging a prefab of an object onto the scene view will instantiate an instance of that prefab. It also has an expanding view for more complex file types like “.fbx” files. Figure 10 shows how the file “Jump–InAir.anim.fbx” is shown in the editor.

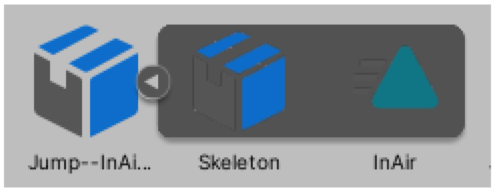
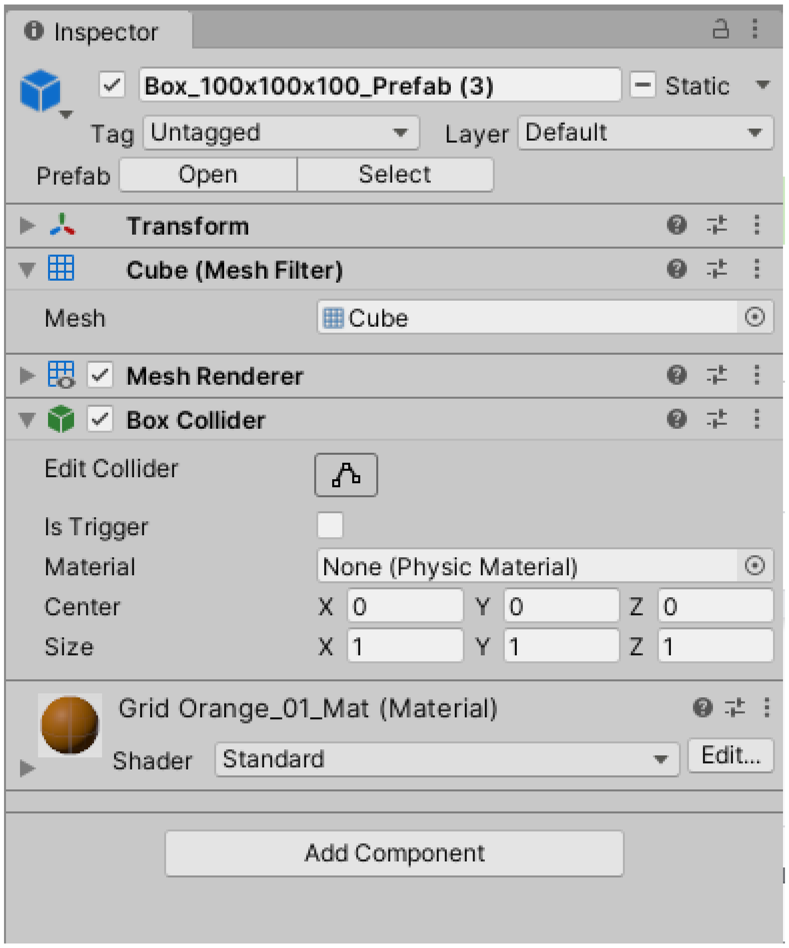


Figure 10. FBX file as shown in Unity's file explorer

2.1.6 Inspector

One of the most important interfaces in Unity is the Inspector view. The inspector window (Figure 11) helps users view and edit properties and settings for almost everything in the Unity Editor, including GameObjects, Unity components, assets, materials, and in-editor settings and preferences.

Each section of the Inspector interface stands for an individual Unity Game component, which can be ready out of the box or represent a class that the user has input into. With the game inspector, developers can add C# classes to an existing GameObject.



Inspector

Box_100x100x100_Prefab (3) — Static

Tag: Untagged — Layer: Default

Prefab: Open | Select

Transform

Cube (Mesh Filter)

Mesh: Cube

Mesh Renderer

Box Collider

Edit Collider

Is Trigger: ☐

Material: None (Physic Material)

Center: X 0 Y 0 Z 0

Size: X 1 Y 1 Z 1

Grid Orange_01_Mat (Material)

Shader: Standard — Edit...

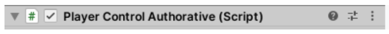
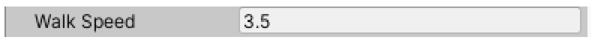
Add Component

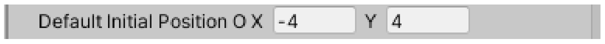
Figure 11. Unity Inspector

These components can be modified in the inspector outside and inside of a game's runtime. Game developers can add or remove components and view or edit their values in real time using the interface. Procedural changes to the game made from code automatically reflect themselves in the inspector view. For example, a moving GameObject's transform component will have its position values of x, y, and z modified every frame during game play.

Users can modify the class of the component by adding or removing custom values, as shown in Table 4.

Table 4. C# scripts in correspondence with the Inspector

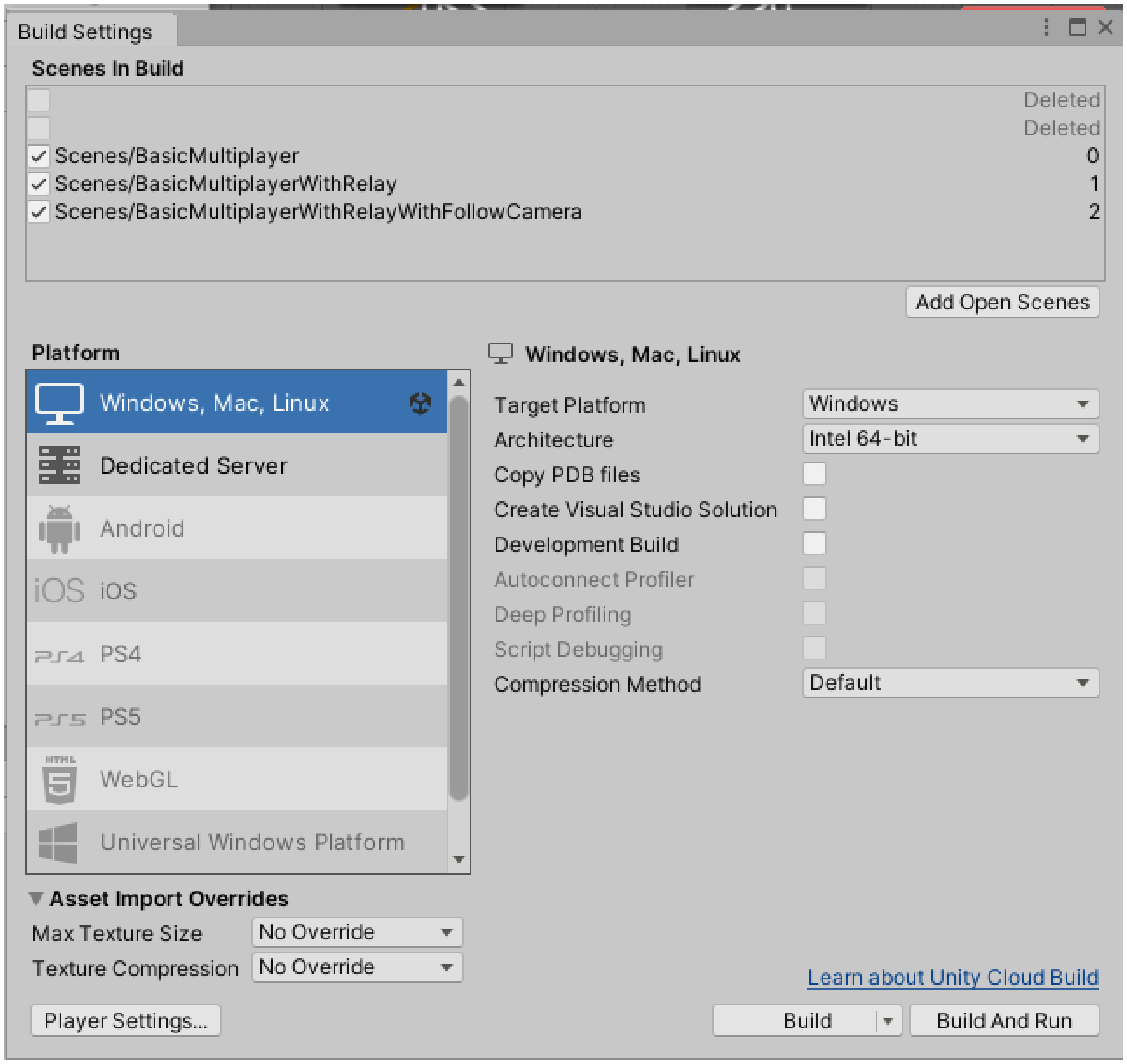
Code	Inspector view
public class PlayerControlAuthorative : NetworkBehaviour	 Player Control Authorative (Script)
[SerializeField]	 Walk Speed 3.5

private float walkSpeed = 3.5f;	
[SerializeField] private Vector2 defaultInitialPositionOnPlane = new Vector2(-4, 4);	 Default Initial Position O X -4 Y 4

It is a good practice in companies that game developers create and expose these fields to the Unity editor so that game designers can modify them in real time, saving testing time and progressing towards the final results as soon as possible.

2.1.7 Build settings

The “Build Settings” screen (Figure 12) takes into account the final phase of game development, which is building the game and targeting different platforms. A scene from the game must be listed in the “Scenes in Build” list to appear in the build.



Build Settings

Scenes In Build

☐		Deleted
☐		Deleted
✓	Scenes/BasicMultiplayer	0
✓	Scenes/BasicMultiplayerWithRelay	1
✓	Scenes/BasicMultiplayerWithRelayWithFollowCamera	2

Platform: Windows, Mac, Linux

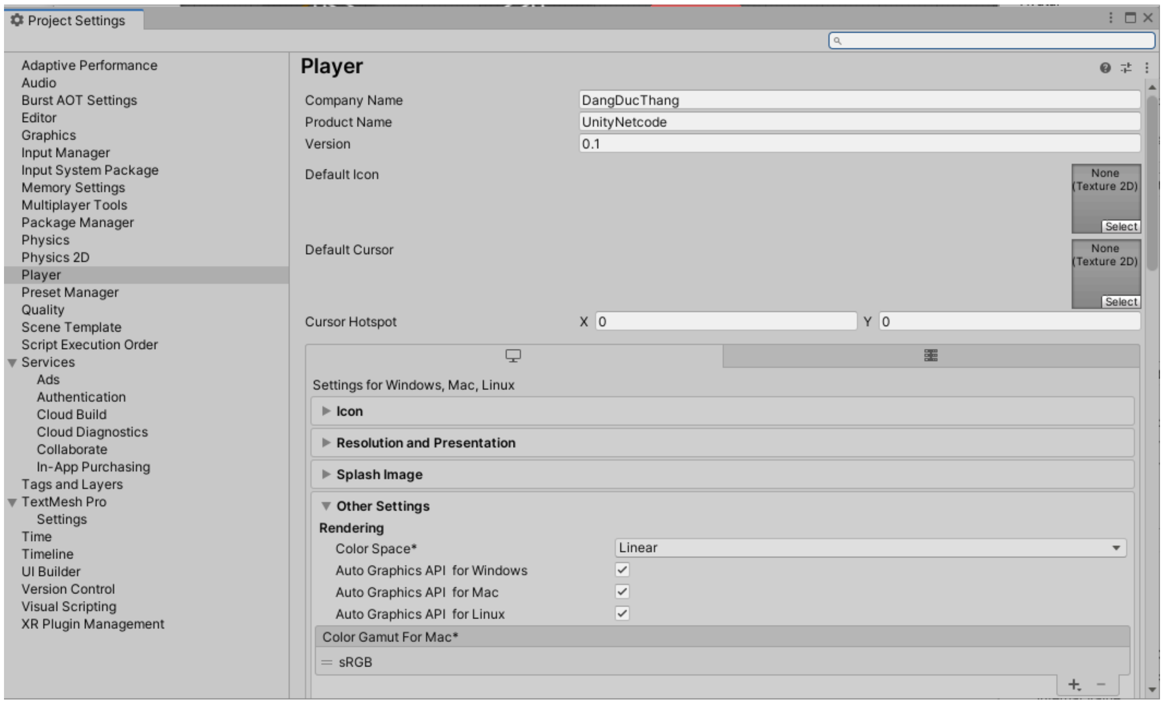
Target Platform	Windows
Architecture	Intel 64-bit
Copy PDB files	☐
Create Visual Studio Solution	☐
Development Build	☐
Autoconnect Profiler	☐
Deep Profiling	☐
Script Debugging	☐
Compression Method	Default

Asset Import Overrides

Max Texture Size	No Override
Texture Compression	No Override

Figure 12. Build settings screen

From here, users can access the Project Settings menu (Figure 13) through the button “Player Settings...”. It is a configuration interface for everything in the game, from game Icon, to gameplay, to services offered by Unity



Project Settings

Adaptive Performance
Audio
Burst AOT Settings
Editor
Graphics
Input Manager
Input System Package
Memory Settings
Multiplayer Tools
Package Manager
Physics
Physics 2D
Player
Preset Manager
Quality
Scene Template
Script Execution Order
Services
  Ads
  Authentication
  Cloud Build
  Cloud Diagnostics
  Collaborate
  In-App Purchasing
Tags and Layers
TextMesh Pro
  Settings
Time
Timeline
UI Builder
Version Control
Visual Scripting
XR Plugin Management

Player

Company Name	DangDucThang
Product Name	UnityNetcode
Version	0.1
Default Icon	None (Texture 2D)
Default Cursor	None (Texture 2D)
Cursor Hotspot	X 0 Y 0

Settings for Windows, Mac, Linux
- Icon
- Resolution and Presentation
- Splash Image
- Other Settings

Rendering

Color Space*	Linear
Auto Graphics API for Windows	✓
Auto Graphics API for Mac	✓
Auto Graphics API for Linux	✓

Color Gamut For Mac*
sRGB

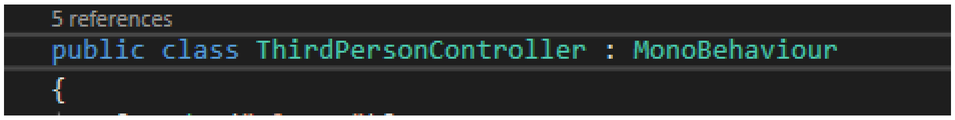
Figure 13. Project settings screen

2.2 Unity mechanics

The Unity Engine is component-based, meaning everything can be expressed through the use of game components. C# classes can be assigned to a GameObject, similar to components, granting them specified characteristics or behaviors.

2.2.1 Components

Components are the base class for everything added to GameObjects. Users' code never directly creates a Component. Instead, users write script code and attach the script to a GameObject. A C# has to be inherited from the base class MonoBehaviour to utilize other parts of Unity like appearing in the editor's Inpector screen, or being assigned to a GameObject (Figure 14).



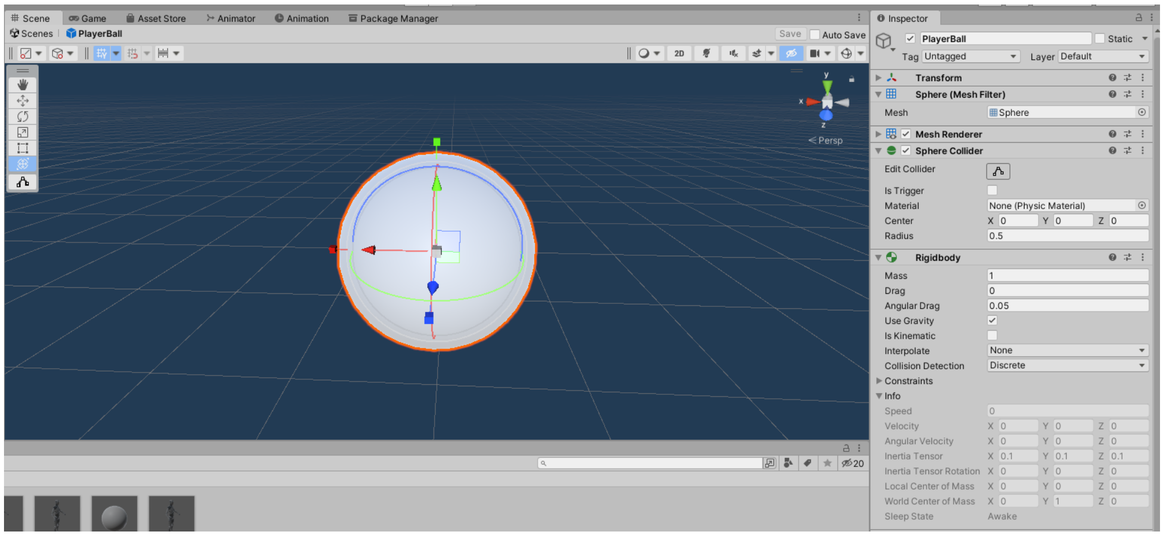
```
5 references
public class ThirdPersonController : MonoBehaviour
{
```

Figure 14. A class called "ThirdPersonController"
An object can have multiple components attached to it, and a component can be attached to many objects. For example, both the player's character and the enemy have the same components that support the running action. However, the

player’s character will also have a component that handles the player's input, while the enemy might have an AI script attached to it. With this structure, Unity heavily supports object-oriented programming.
Components are aware of other components in the same scene, meaning that they can be easily referenced in code to create relationships between different classes. GameObject.Transform is a special component that keeps the data that all objects must have, namely their positions in the world space and a list of their parents and children. It is created by default along with GameObjects, and cannot be removed.

2.2.2 GameObjects

GameObjects are another indispensable building block of Unity games. While these objects do not carry out any actions, they are necessary as carriers for the components. They can be thought of as nouns and verbs. While all objects have transform information, they do not have any shapes, meaning a default empty GameObjects is simply unobservable in the world space. These objects are often used only as containers for actionable scripts like Network Manager or a game manager that takes care of miscellaneous things in the game.
In-game interactable objects, on the other hand, are what players see in the game. The visual appearance of an object is made by adding a mesh renderer component (for visual shaping) along with a collider and rigidbody component (for physical interactions) to it (Figure 15). Unity provides different basic shapes like cubes, spheres, or capsules ready-made for simple applications. In fact, the capsule is one of the most commonly used shapes for a player character’s physical representation in games.



```
Inspector
PlayerBall                         Static
Tag Untagged             Layer Default
Transform
Sphere (Mesh Filter)
  Mesh                   Sphere
Mesh Renderer
Sphere Collider
  Edit Collider
  Is Trigger
  Material               None (Physic Material)
  Center                 X 0    Y 0    Z 0
  Radius                 0.5
Rigidbody
  Mass                   1
  Drag                   0
  Angular Drag           0.05
  Use Gravity            ✓
  Is Kinematic
  Interpolate            None
  Collision Detection    Discrete
  Constraints
  Info
  Speed                  0
  Velocity               X 0    Y 0    Z 0
  Angular Velocity       X 0    Y 0    Z 0
  Inertia Tensor         X 0.1  Y 0.1  Z 0.1
  Inertia Tensor Rotation X 0   Y 0    Z 0
  Local Center of Mass   X 0    Y 0    Z 0
  World Center of Mass   X 0    Y 1    Z 0
  Sleep State            Awake
```

Figure 15. A spherical GameObject

2.2.3 Assets

Unity is noun-based, meaning that the core of logic revolves around assets. Assets can be anything that might be needed in the game. Unity supports the creation and addition of many assets for game development. Some asset types that can be mentioned are: materials, physics materials, animations and animators, scripts, sounds, and prefabs. Even scenes are considered assets. These assets can be utilized through the engine-supported tools, e.g., the Audio Mixer for sounds and the mesh renderer for material.

Every file in Unity is tracked using a metafile of the same name. Any modification of assets in a Unity project will result in a reload of the assets and a modification of the meta files. Unity can detect file modifications that are made outside of the engine; however, it is not made to handle Git smoothly, as changing a position slightly will introduce conflicts in many places within the game in a non-human language.

Assets also contain items that are downloaded from the Unity Asset Store or the Package Manager. Usually they have a default parent folder called “Packages” and kept track in a metadata file called “manifest.json”

2.2.4 Prefab

Unity has a special asset type named "prefabs." Prefabs are like classes for GameObjects; they contain information regarding all components and value settings in an object. Furthermore, when the prefab is modified, all objects that are instantiated from it will update accordingly. This is a key process for increasing the speed and reusability of projects.

Prefabs can be used to handle complicated objects, saving time every time that object is reused. In addition, prefabs are also utilized for generating multiple objects of the same type using scripts. On the scale of thousands of repeating GameObjects, developers have to use prefabs instead of dragging and dropping by hand.

Prefabs will appear in the hierarchy view in blue text as shown in Figure 16. They have their own editor view in the scene view and have the same inspector view as every other game object.

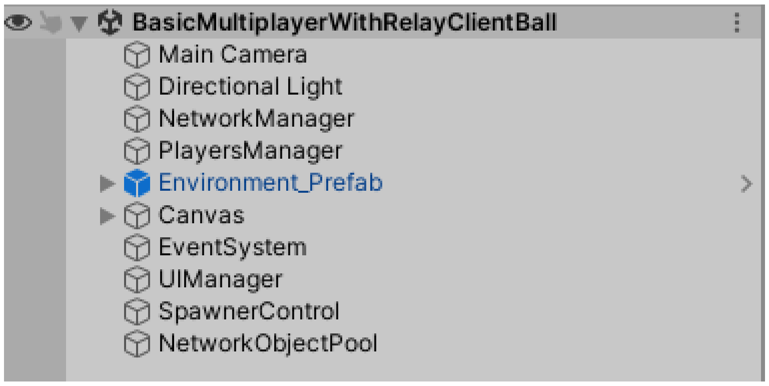


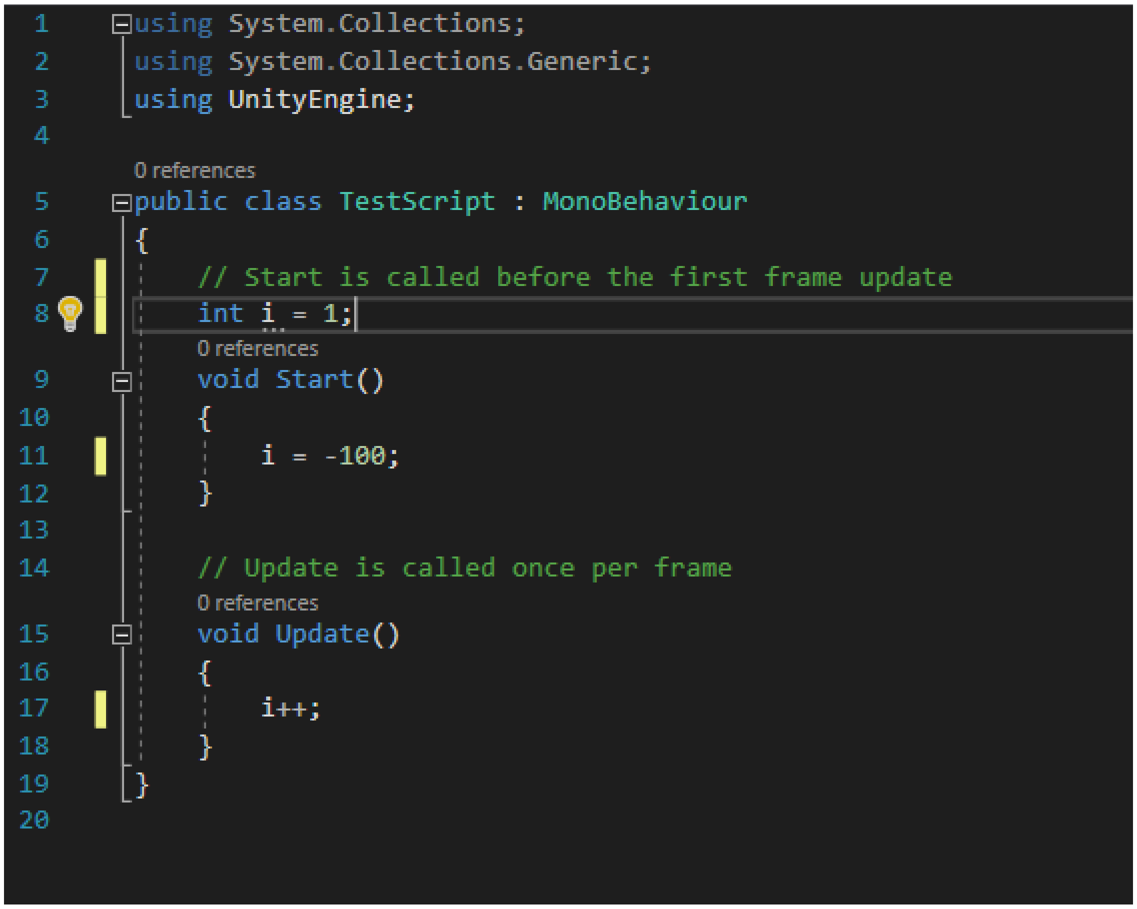
Figure 16. A Prefab in the Hierarchy

2.2.5 Scenes

Scenes are a premade settings in which the game will be played in. A scene can be a main menu UI scene, or a level that players spawn in. The scene holds the information of every GameObjects initiated in it through metadata files. Scene may be edited one at a time in the Unity editor. To instantiate objects, the developer either drags the prefabs onto the scene or generates them using code. Scenes can include the environment, decoration, pre-spawned player characters, and script-carrying objects. By default, all GameObjects in the scene are destroyed when another scene is loaded, unless specified otherwise.

2.2.6 Scripting

Scripts in Unity are written with components in mind. To be assigned to an object, the script has to be inherited from MonoBehavior. Developers have to take into account the engine’s cycle to utilize the functions provided by Unity. For example, the Start function will be called only once after the carrier object is instantiated, while the Update function will run every visual frame of the game. Both functions are automatically generated by Unity at the start of every script. The below script generates a private integer ii, sets it to be -100 when this script’s GameObject is initialized, and adds 1 to i every frame of the game.



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

0 references
public class TestScript : MonoBehaviour
{
    // Start is called before the first frame update
    int i = 1;
    0 references
    void Start()
    {
        i = -100;
    }

    // Update is called once per frame
    0 references
    void Update()
    {
        i++;
    }
}
```

Figure 17. Unity execution cycles demonstration
The simple program shown in Figure 17 plans that, after 60 frame (roughly 2 seconds in a 30fps game), the variable ”i” holds the value of 40. By default, Unity gives 2 comments to help users who are new to the program.

2.3 Asset store and Package manager

The asset store is a key part and needs to be utilized by smaller teams to create games in Unity. Launched in November 2010, the Unity Asset Store is an online marketplace where Unity users and other developers or artists can buy and sell project assets. With over 40,000 asset packages in a variety of categories and prices ranging from free to over $1000, the Asset Store provides a potential opportunity to save money on game development. The publisher of an asset receives a 70 percent cut of the asset's set price in the store, while Unity retains 30 percent of each sale. In addition to free tutorials, sample projects, and standard assets, Unity Technologies also offers free tutorials, sample projects, and standard assets.

From the asset store, developers can purchase artwork and sound sets that are custom-made for games. There are also plugins or more Unity windows to increase ease of use. Many Unity-made components that are not included in the engine build because they are not commercially stable yet can be found and downloaded here.

The third-person starter pack is an example of Unity’s assets. It is a ready-made project that can be run right after downloading, giving developers a framework to kickstart a third-person game or act as research material for best practices in Unity. This project will utilize said asset for its map.

3 NETCODE

Netcode for GameObjects (Netcode) is a high-level networking library for Unity that allows developers to abstract network logic. It allows you to send GameObjects and world data to multiple players simultaneously over a networking session. With Netcode, developers can concentrate on building their games rather than on low-level networking protocols and frameworks.

3.1 Network

3.1.1 NetworkObject

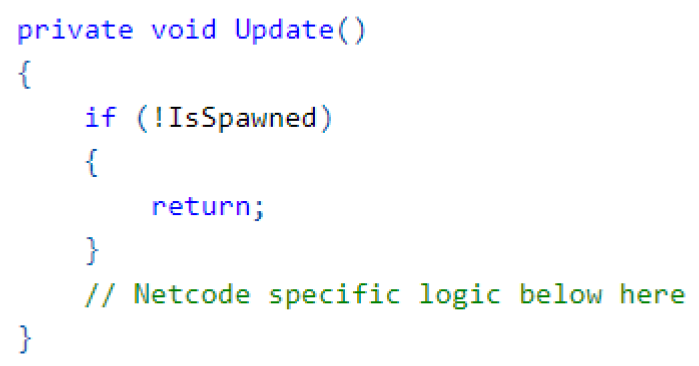
NetworkObject is the component that enables the most important features of Netcode like RPCs, NetworkVariables, and the object spawning system. It essentially is an ID that is assigned to a GameObject to make it awared by the whole network (clients, servers, hosts).
For example, a game world usually has the ground, that is the same across all clients, and does not need customized interaction among different players. This ground object can be stored locally and does not need to be recognized over the network. On the other hand, a player character needs to be spawned on all clients when he joins the game, and when it is moved by the owner, that movement has to be reflected on other machines as well. In this case, when that player object is spawned, it will be assigned an NetworkObjectID of, for example, 1. After said assignment, the server or host can tell the clients to spawn the object with the ID of 1, and move it to the position of (0,1,3) and the request is understood accordingly.
Not only does NetworkObject hold the ID of its parent object, it also stores certain values that are necessary in a multiplayer game. A NetworkObject always has a value for an owner – the machine that is authorized to manipulate it. By default, this owner is set to server or host, depending on the structure of the network. Only the owner of the object can spawn it or change its ownership status of the object itself.
In Netcode, there is a special NetworkObject which is the PlayerObject. It is unique to each client and clients cannot have more than one player object. When set up, Netcode will automatically spawn the player object for each player when they join the game. In most action games, players control a single character that can move or shoot, the logic of which can be effortlessly implemented using this feature.
NetworkObject also holds some configuration information like transform synchronization, which decides whether that object needs to be considered as something that moves in the scene, or a purely logical object (which many

GameObjects are). Another choice is to whether keep that object when changing scenes on the clients or not.
As explained above, NetworkObject is the core of the Netcode system, and it is made with the Unity's code flow in mind. To spawn a NetworkObject in the game, the following code is used:

```
GetComponent<NetworkObject>().Spawn();
```

3.1.2 NetworkBehaviour

Requires NetworkObject on the same GO, or parent, auto add. Derives from MonoBehaviour.OnNetworkSpawn
NetworkBehaviour component, as suggested from the name, is a network-aware replacement for MonoBehaviour. It is in fact derived from the class MonoBehaviour and behaves like its parent class. NetworkBehaviour requires a NetworkObject component on the same GameObject as itself, or on a parenting GameObject. The reason for this requirement is that Netcode needs to identify the GameObject that is making the request, as explained from chapter 3.1.1. If the assigned object does not have a NetworkObject component ready, Netcode automatically adds one.
NetworkBehaviour enables the usage of NetworkVariables and RPCs, which are the network versions of common C# variables and functions. NetworkBehaviour acts like a common MonoBehaviour class with network features added. It adds custom logic to the GameObject, much like offline game scripting. However, there are network specific differences that the user has to be aware of. For example, other than the common Start() function that is common between MonoBehaviour and NetworkBehaviour, the network version also has a method called OnNetworkSpawn() that is called before or after Start(), depending on whether the object was created dynamically or placed in the scene before game start. The interaction is well explained on Unity's Netcode documents, about when to use which version of these slightly different functions.



```
private void Update()
{
    if (!IsSpawned)
    {
        return;
    }
    // Netcode specific logic below here
}
```

Figure 18. Netcode "IsSpawned" usage

The field IsSpawned is also a network-aware version of IsActive, which is implemented in the Update loot demonstrated in Figure 18. In the mentioned code, the local object will check if it is spawned over the network, then return specific codes.

3.1.3 NetworkVariables

Much like common variables in C#, NetworkVariable are nouns that are used in scripting logics. To make the distinction between the two mentioned nouns, the following example can be referred to. A door in the game can have two states, either opened or closed, which can be stored in a Boolean variable named IsOpen with the value of either true or false. If said variable is changed, the door state is also reflected in the game scene. However, the other clients in the network also need to be aware of the variable in the original player's computer. To make that happen, the developer can write custom logic to broadcast to the network everytime IsOpen is changed, or make use of Netcode's NetworkVariable. NetworkVariable is automatically handled by the Netcode backend system. Utilizing this feature speeds up the development process and improve the code's readability. NetworkVariables can be customized to work the same way any common variable does, while also being aware of network permissions and other important information.

3.2 Remote Procedure Calls

Remote Procedure Calls, or RPC, are the main way for developers to send custom messages over the network. It is the Netcode-awared version of functions. RPCs can either be called from clients or the server. To utilize RPCs, developers have to follow a naming requirement of adding the tag [ClientRpc] (or [ServerRpc]) before a function, and putting ClientRpc (or ServerRpc) suffix in that function's name. RPCs have parameters much like common C# functions.

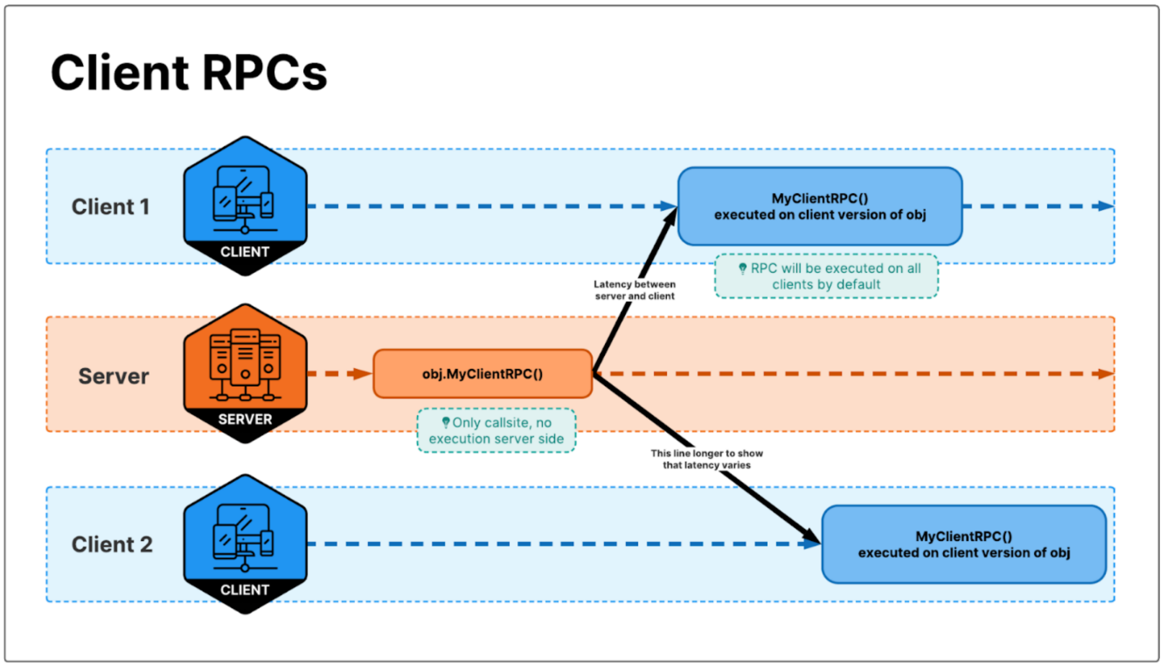

Client RPCs
Client 1
CLIENT
MyClientRPC()
executed on client version of obj
RPC will be executed on all clients by default
Latency between server and client
Server
SERVER
obj.MyClientRPC()
Only callsite, no execution server side
This line longer to show that latency varies
Client 2
CLIENT
MyClientRPC()
executed on client version of obj

Figure 19. Client RPCs

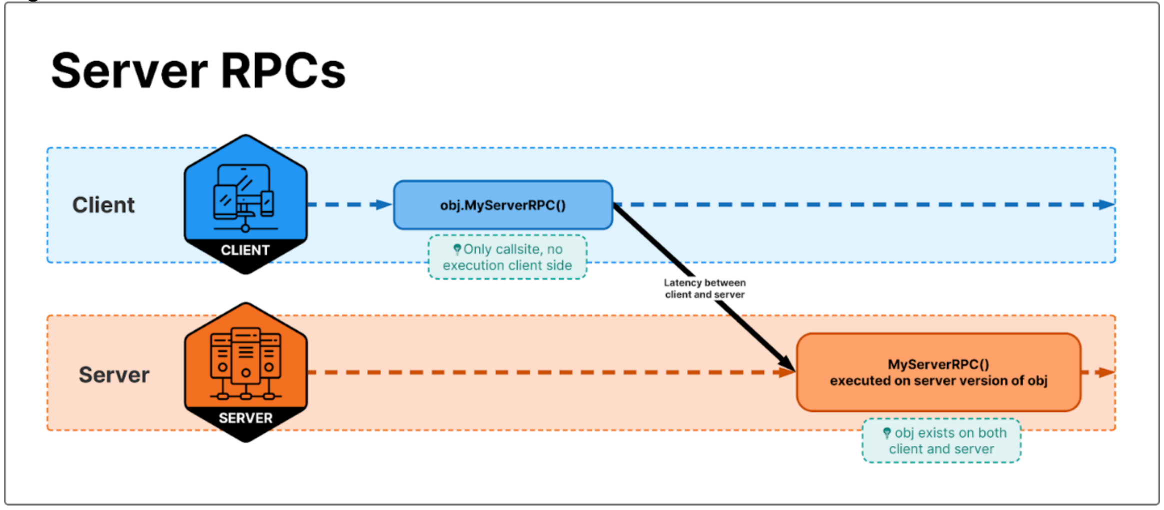

Server RPCs
Client
CLIENT
obj.MyServerRPC()
Only callsite, no execution client side
Latency between client and server
Server
SERVER
MyServerRPC()
executed on server version of obj
obj exists on both client and server

Figure 20. Server RPCs

Figures 19 and 20 shows the process of client and server RPCs. ClientRPCs are called by the server and executed on the client,and the vice versa is applied to ServerRPCs.

3.3 Components

Well embedded in Unity, Netcode adds to the natural workflow of Unity developers by providing ready-to-use components that can be used to replace their traditional counterparts to add network related logic to the game. Some noticeable components are NetworkManager, NetworkTransform and NetworkAnimator.

3.3.1 NetworkManager

NetworkManager is the core component of Netcode. It handles network logics in games similar to a GameManager object in offline matters.
NetworkManager comes with a readily available connection system that developer can test right out of the box. It provides three options for roles that a machine in the network can take (depending on Netcode’s network topology). A computer can assume the role of a server that runs game logic or other machines, a client that listens to said server, or a host that is both a server and a client. The three connecting functions are also conveniently placed in the Inspector view of this component.
NetworkManager requires a Transport or Unity Transport component, that handles the connecting details like IP and Port. It is a primitive way of handling connection however, and such tasks should be delegated to Relay which is mentioned in chapter 3.4.
To disconnect from the network, the following code can be used:

```
public void Disconnect()
{
    NetworkManager.Singleton.Shutdown();
    // At this point we must use the UnityEngine's SceneManager to
switch back to the MainMenu
    UnityEngine.SceneManagement.SceneManager.LoadScene("MainMenu");
}
```

The NetworkManager class also have a wide range of crucial network configurations and data that is explained in Table 5.

Table 5. Important NetworkManager fields

Field name	Description
Log level	Detail level of debug information
PlayerPrefab	A special prefab that is unique to each player. Auto matically spawned and assigned when a client connects to the game
NetworkPrefab	A list of all prefabs that need to be network-aware in the game
Protocol Version	Made for matching build versions so as not to create conflicts over the network
Network Transport	A set of configurations for the transport method
Tick Rate	Number of times the network updates per second (like frame rate)
Connection Approval	Used along with NetworkManager.ConnectionApprovalCallback. Helps implementing custom connection logic.
Load Scene Time Out	The amount of time the NetworkSceneManager wait before considering the loading a failed connection.

3.3.2 NetworkTransform

NetworkTransform, as suggested from the name, is a network-aware version of the component Transform. It handles all of the positioning logic of an online object automatically when put on an object (that already has a NetworkObject component attached to it). The high-level logic of this feature includes:

- Being aware of the local object’s position.
- Serializing that position object to optimize bandwidth.
- Broadcasting the serialized position over the network (usually through the help of the server).
- Deserialize the position object locally on other machines.
- Other machines apply the new value to their local GameObjects.

Other than these steps, it also must take care of the details:

- The owner/authority of the position change (whether it is a client or a server).
- Tick rate/Interpolation: The trade off between smoothness and accuracy.

- Quaternion or Euler rotation: Euler is more efficient in terms of data load, while Quaternion gives more flexibility to the rotation.
- Which information need to be sent over the network: In most first-person shooter games, the character does not turn sideways. Therefore, the rotation on the horizontal axis can be left out for optimization.

3.3.3 NetworkAnimator

NetworkAnimator is the network version of Unity’s Animator. It handles animation related information over the network, including date like animation states, transitions, and properties. Naturally due to the way animation states are synchronized, if a player joins the game in the middle of an animation, they will not receive that animation on their machine. This is because animator use a trigger property to mark the start of an animation. Animations are stored locally on each machine, and only the trigger and variables are sent over the network to ensure the optimized usage of bandwidth.

There are two modes for the NetworkAnimator, each with its own advantages and drawbacks: Server Authoritative (default mode), and Client Authoritative. These modes decide the machine that has the authority to initiate a state change.

The Server Authoritative mode reduces the synchronization latency between all client animations and ensures that players see what other players are doing in roughly the same time. However, because the input from the client must be sent as a request to the server before coming back to the screen of the user, it causes a delay for the owner of the action itself, hurting the gaming experience of players.

The Owner Authoritative mode fixes the above problem by showing the animation on the user’s screen first, then broadcasting to the network. However, this also means that players will see the Owner’s animation after a full round trip time. This might cause unfair situations like a player feels like they have hit another player with a bullet, while in fact they are a few frames behind.

3.4 Relay

To connect two machines over different networks (over the Internet) Unity provide a service outside of Netcode named Relay. Relay is a third party server on the Internet with a public-facing IP that both computers can reach. Instead of connecting directly using IP and port information, the clients send game data over the Relay server.

Like a common online game experience, Relay provide online “lobbies” that players can create and join. To enable Relay in a project, the developer needs to install its SDK, and enable it in their Unity Dashboard. The “lobbies” are created using such code:

```
//Ask Unity Services to allocate a Relay server that will handle up
to eight players: seven peers and the host.
Allocation allocation = await
Unity.Services.Relay.RelayService.Instance.CreateAllocationAsync(7);
```

Here, Allocation is a class that holds all the needed information to create such lobby. Developers can specify the capacity of the lobby by changing the parameter of the function CreateAllocationAsync().

It is worth noting that Relay is not a part of Netcode but belongs to Unity’s gaming service which acts like a game publishing platform. It has a price model which starts with free usage, and scales up in price according to the number of players using the game.

4 GAME PROTOTYPE

Prototyping is a common video game development technique which aims to test out a game’s vision – with minimal time and effort invested (UXPIN 2023). Game prototypes are a minimal version of the game, focusing on a rather narrow scope that aims to gauge certain metrics of said game. Key areas can include execution feasibility, player interest, procedure effectiveness, etc.

Prototypes can be made in form of physical assets like paper or figures, or they can be made in wireframe to demonstrate the implementation of UI/UX. In this research, the auther utilizes a method called ”Greybox prototyping”, meaning creating the game in Unity in while stripping it of visual elements as well as other uninteresting parts. The prototype will focus on exploring the networking side of the game and only have minimal visualization and gameplay mechanics.

4.1 Pillars of Tagteam

A game's core pillars are a set of parameters that the game team set out at the start of the development process and follow through as there can be various questions that emerge during the developement process where they need a reference for decision making.

Core pillars help the development team understand the overall picture of the game, ensuring a smoother development process. Core pillars empowers the team to make more informed design decisions. They act as guidelines to help make the game team's life easier. (Max 2017.)

In this thesis, a game's core pillars will not only carry the meaning in the sense of game design, but also in fulfilling the project's goal and requirements. The main purpose of the project is to demonstrate how a 3D multiplayer game can be made in a short time period in order to test the gameplay and functions of the program, accounting for the fast production cycle of modern game development.

With the knowledge above, Tagteam's core pillars can be stated as bellow:

- **Playability:** Users can use features such as multiplayer input and output, and interact with other players.
- **Reusability:** The components of the game are created in a way that can be copied over to later versions or to other games. This can also come in a form of a full creation process for other areas other than code.

4.2 Game Design

The game is played using the same controls as third-person shooter games.

WS keys: moving forwards and backwards

Space key: jump

AD keys: turn the character.

As understood from the name of the game, it involves players chasing each other, if the chaser manages to touch the runner, the game ends in victory for the chasing side. On the other hand, if the runners can survive for a long enough time period (3 minutes), they win the game.

4.3 Implementation

The author planned to carry out the game in the span of 2 weeks. To make the most out of the short development period, the author used Unity starter assets for the base game, and built the game utilizing its models, pre-built map, and some animations. Then, the author rewrote the game’s logic and updated Unity components to transform it into a multiplayer game.

4.3.1 Starter Assets

Unity provides a starter asset for kickstarting prototyping projects that is totally free. The asset pack has a prebuilt map with various objects and a prefab for the player character with ready-made animations and meshes. This is a very good starting point for any project. The starter asset can be found in Unity's online asset store (Figure 21) by typing the name in the search bar.

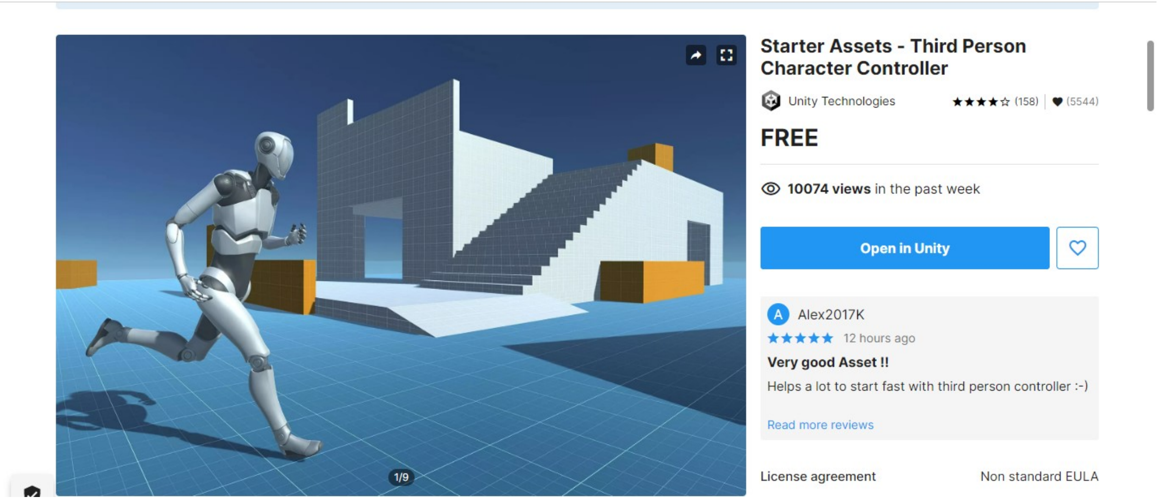


Figure 21. Starter Assets store page

After purchasing and opening the asset in Unity, the next step is to immediately load the asset. By going to StarterAssets>ThirdPersonController>Scenes>Playground. Unity, the user can see the ready-made playground and placed characters (Figure 22).

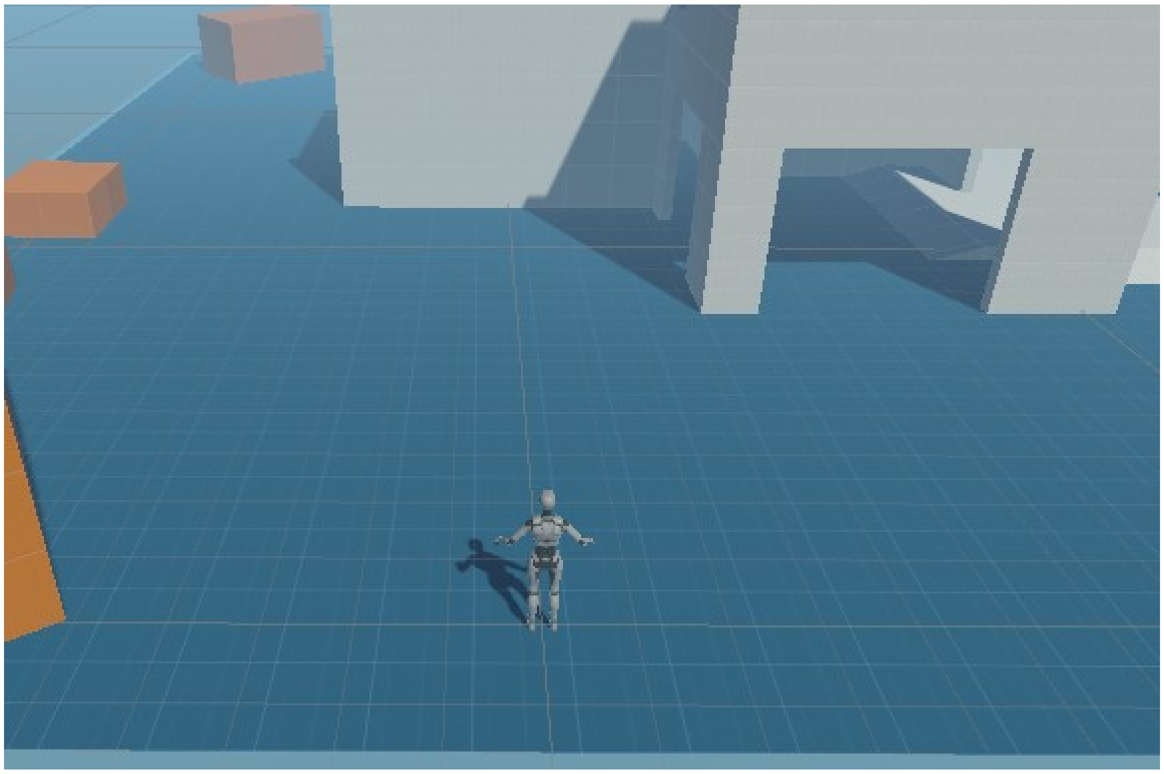


Figure 22.Starter Asset default view in Unity

The player can move around and jump off objects. However, this scene lacks multiplayer content and features.

4.3.2 Netcode implementation

It is important to note that Netcode only supports later Unity versions. According to Unity’s documentation, here are the requirements for Netcode:

- Unity 2020.3, 2021.1, 2021.2, and 2021.3 LTS
- Mono and IL2CPP Scripting Backends

Since Netcode is not yet an entry in Unity’s registry of packages in the Package Manager in its 2020 versions, the user has to take some extra steps to install Netcode.

1. Open Unity Hub and select the current project/New project.
2. Click Window then Package Manager.
3. For users of versions 2021.3+, the package has to be added by name (Figure 23), accessed throught he plus sign in the Package Manager’s status bar.
4. For Unity editor versions lower than 2020.3LTS, adding package by name is not an option. In this case, users need to choose the option : “Add package from git URL”.
5. Next the link of the package needs to be inserted in the pop-up window: “com.unity.netcode.gameobjects”.

6. After the user clicking Add, the package appears as Netcode for GameOjects in the Package Manager.

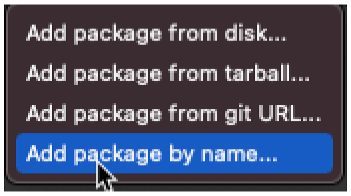


Figure 23. Adding package by name

4.3.3 NetworkManager

The NetworkManager component is required for every project using Netcode. A good practice is to create an empty GameObject in a used scene and assign the component to it.
From the dropdown bar of the Hierarchy window (opened by right-clicking anywhere in the Hierarchy window), a new empty object can be created as the first choice (Figure 24). That empty object should be called “NetworkManager” for ease of object management (Figure 25).

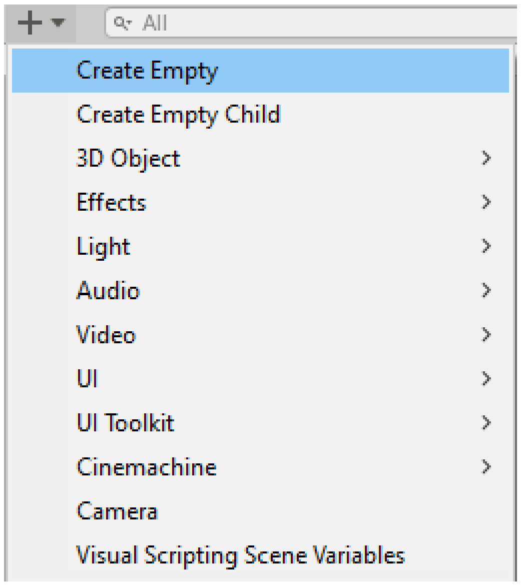


Figure 24. Creating an empty GameObject

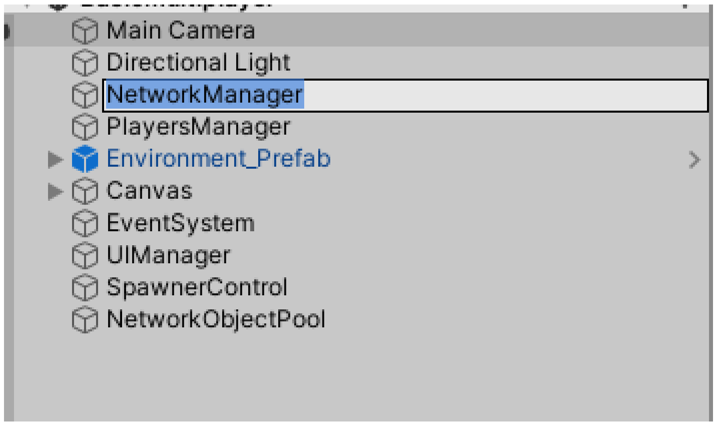
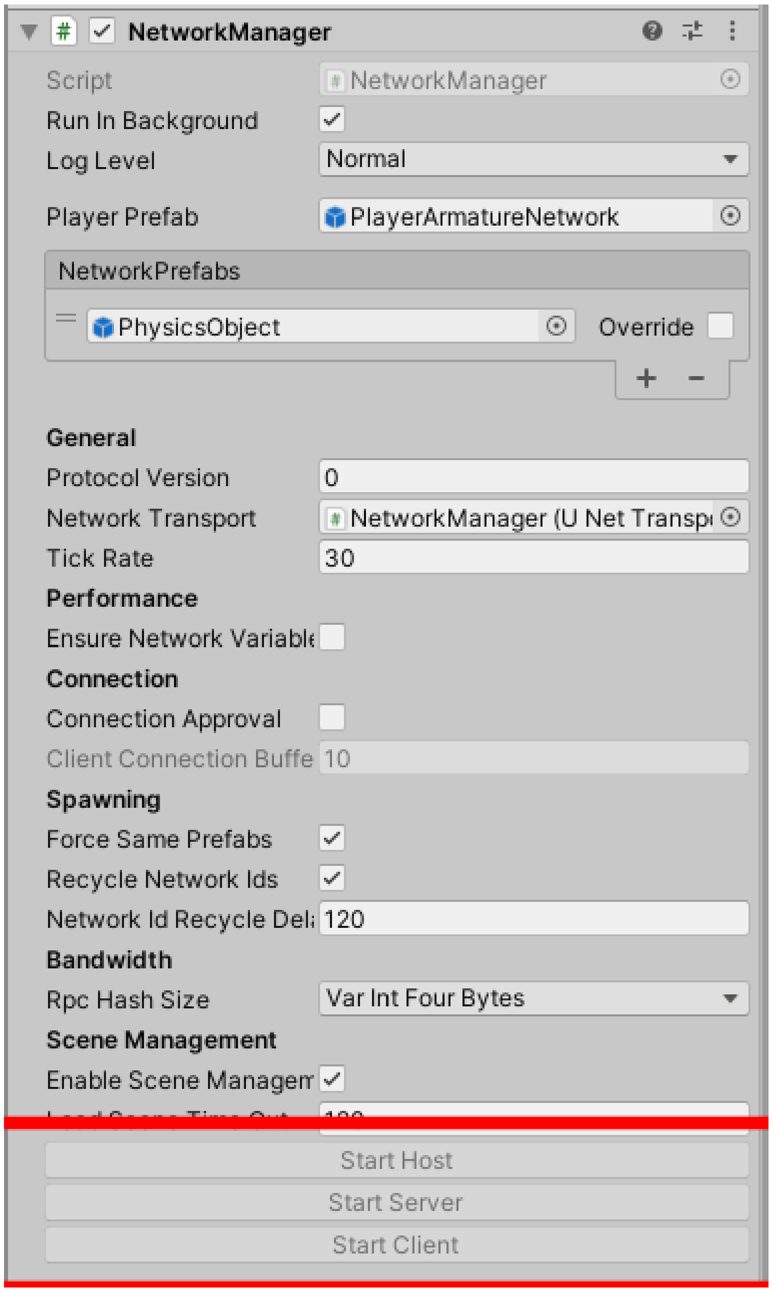


Figure 25. Renaming newly created GameObject to NetworkManager

The Network component details have been listed in Chapter 3.2.1. The most noticeable thing is that it has 3 functions exposed to the user: StartHost, StartServer, and StartClient in the bottom area as shown in Figure 26. These functions can later be called within the game to initialize the game automatically.

- Server: the machine where the game is run on, server assumes all authority unless specified otherwise.
- Client: clients are the machines that by default receive information from the server and mirror on their own build.
- Host: A host is both a server and a client.



NetworkManager

Script	NetworkManager
Run In Background	✓
Log Level	Normal
Player Prefab	PlayerArmatureNetwork

NetworkPrefabs

PhysicsObject	Override	☐

General

Protocol Version	0
Network Transport	NetworkManager (U Net Transp
Tick Rate	30

Performance

Ensure Network Variabl	☐

Connection

Connection Approval	☐
Client Connection Buffe	10

Spawning

Force Same Prefabs	✓
Recycle Network Ids	✓
Network Id Recycle Del	120

Bandwidth

Rpc Hash Size	Var Int Four Bytes

Scene Management

Enable Scene Managem	✓

Figure 26. The three network options

4.3.4 Prefab

The Player prefab needs to be created and assigned to the NetworkManager field called Player Prefab. It will then be generated automatically upon players joining the game. The player prefab needs to at least have the NetworkObject and NetworkTransform components (Figure 27).

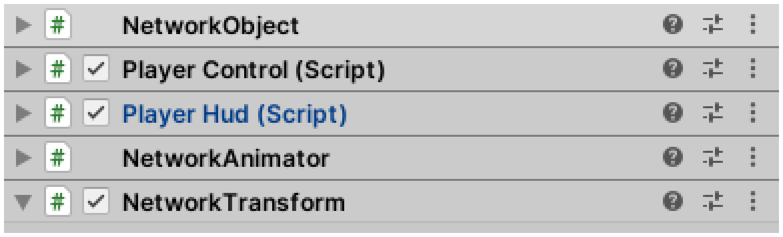


Figure 27. Common components of a Player Prefab

4.3.5 UIManager

Although the start-game functions are exposed to the game developer, they are unreachable for players, rendering them only functional for playtesting. To start a lobby within the game, the player needs a UI button that calls those three functions. A new UI canvas is created in the game along with some script to connect it to the function (Figure 28).

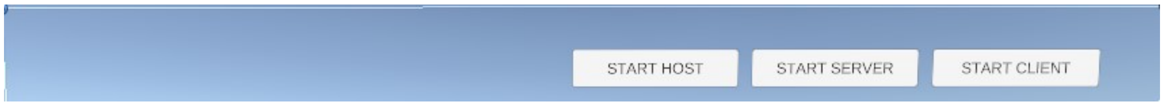
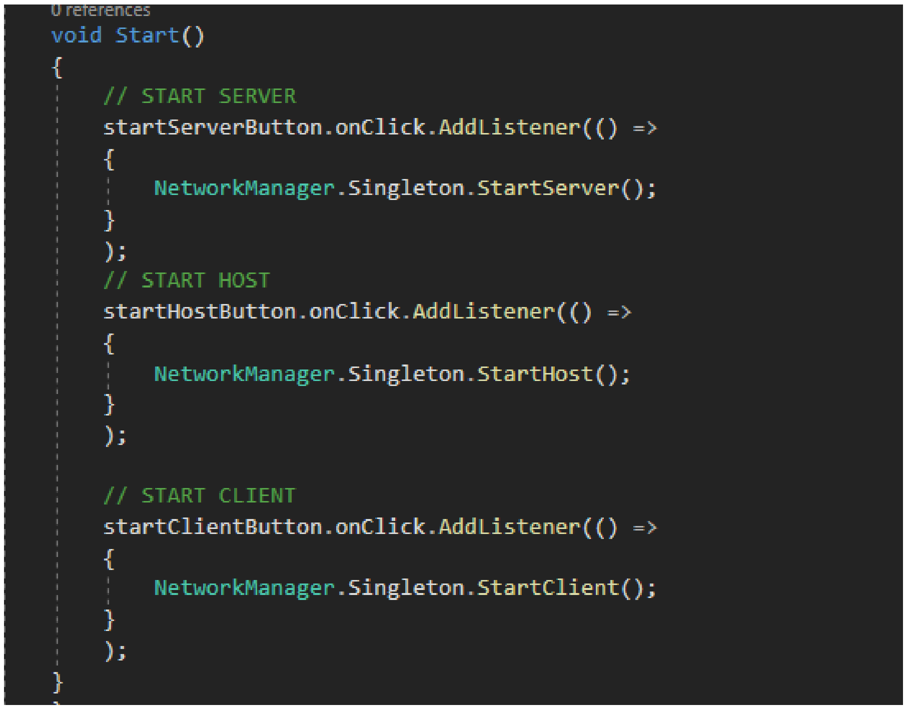


Figure 28. Implementation of network options in the user interface

In the script, these buttons need to be referenced and listened to trigger the function StartHost(), StartClient() and StartServer() (Figure 29).



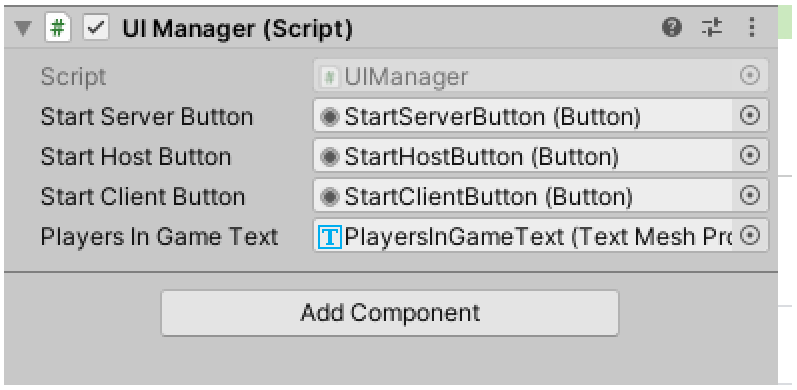
```
0 references
void Start()
{
    // START SERVER
    startServerButton.onClick.AddListener(() =>
    {
        NetworkManager.Singleton.StartServer();
    }
    );
    // START HOST
    startHostButton.onClick.AddListener(() =>
    {
        NetworkManager.Singleton.StartHost();
    }
    );

    // START CLIENT
    startClientButton.onClick.AddListener(() =>
    {
        NetworkManager.Singleton.StartClient();
    }
    );
}
```

Figure 29. Scripts to trigger Netcode's function

To define to Unity which button is connected to each function, the UI element has to be dragged and dropped into the exposed field in the custom UI Manager script (Figure 30)

. Fields that are marked as public or with the tag [SerializableField] are automatically exposed in the Inspector window and are modifiable in-game.



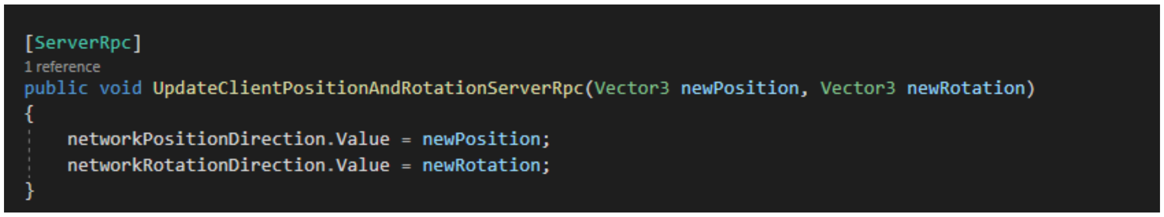
UI Manager (Script)

Script	UIManager
Start Server Button	StartServerButton (Button)
Start Host Button	StartHostButton (Button)
Start Client Button	StartClientButton (Button)
Players In Game Text	PlayersInGameText (Text Mesh Pr

Figure 30. Assigning buttons to the C# script variables

4.3.6 Player’s movement

To add player movement, there needs to be a script taking in the controls of players and then send that information to the server to be broadcast to the network. If the player object is going to be moved and rotated in one client, it’s new position and rotation is sent as two 3D vectors over the server with the function shown in Figure 31, while the object is not actually moved in the physics engine.

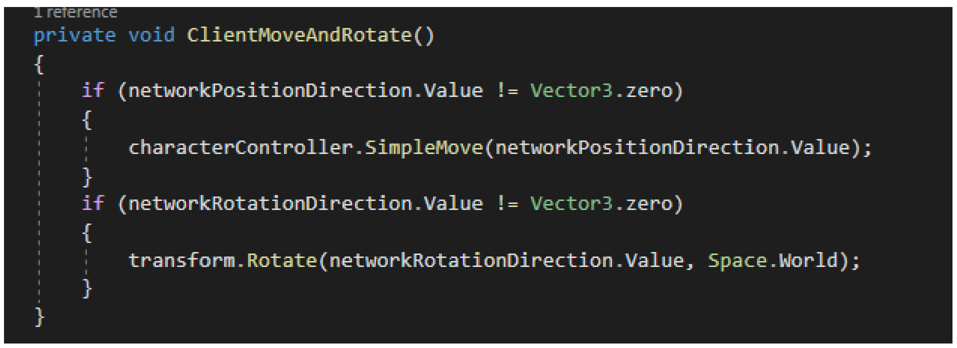


```
[ServerRpc]
1 reference
public void UpdateClientPositionAndRotationServerRpc(Vector3 newPosition, Vector3 newRotation)
{
    networkPositionDirection.Value = newPosition;
    networkRotationDirection.Value = newRotation;
}
```

Figure 31. A function to help with updating position and rotation

Then the client itself fetches that information back from the server and run it in the same update frame (Figure 32). This means that according to Netcode’s logic, the client is not allowed to move the object itself. It has to, however, send a request to the server, where the physics is calculated, and authority is verified before updating what the player sees. This procedure ensures not only antihacking, but also the integrity of what all players see.

Normally in a multiplayer action games, fairness is of utmost importance for players, which is why this method of networking is implemented. The disadvantage of this process is that it depends enormously on the quality of internet services. Lagged payloads will affect player’s experience directly, causing the character to move too far, too short, or back in time in between frames.



```
1 reference
private void ClientMoveAndRotate()
{
    if (networkPositionDirection.Value != Vector3.zero)
    {
        characterController.SimpleMove(networkPositionDirection.Value);
    }
    if (networkRotationDirection.Value != Vector3.zero)
    {
        transform.Rotate(networkRotationDirection.Value, Space.World);
    }
}
```

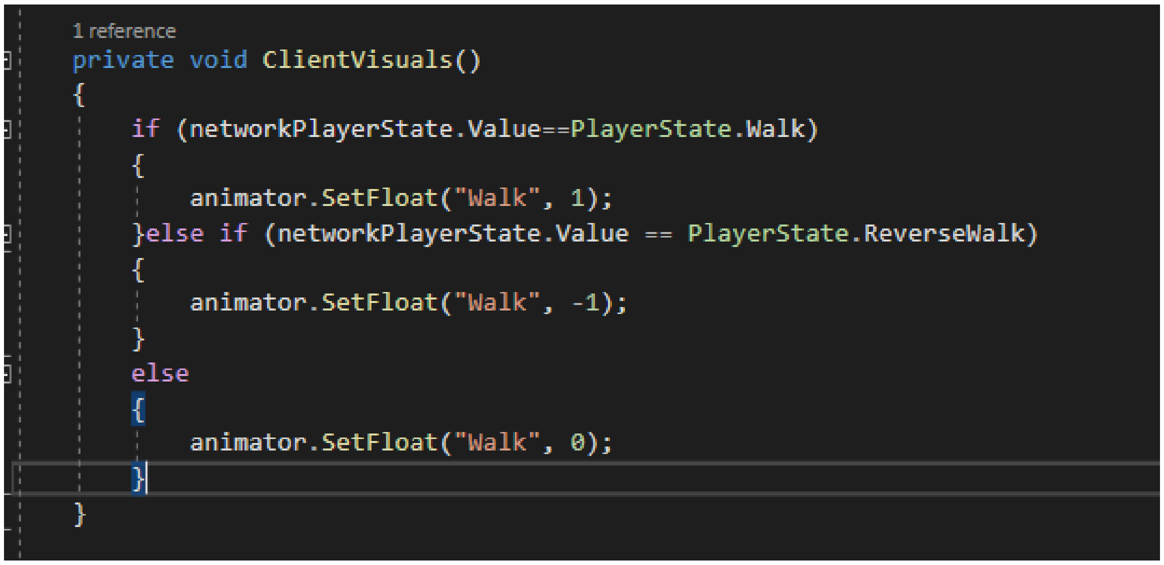
Figure 32. Receiving the information from server to move the object

Another implementation approach is having the server trust the client, where players see exactly their own movement, but not neccessarily of other players.This method, on the other hand, does not ensure smooth multiplayer interactions.

Most advanced action multiplayer games address this by implementing a prediction system where the server guesses before hands where the player will move, bringing the best of both worlds: smooth control and fairness.

4.3.7 Player’s Animation

For this, the Prefab of the player needs to have the component NetworkAnimator that is connected to the offline version of the Animator. To do that, the client has to take the state of the online player in a similar fashion as the movement script. Then it refers to the animator and sets the animation state to the right type. In the script in Figure 33, it can be seen that the server-client communication system will not send the whole animation information like the movement system. Only a simple int is sent over the network to trigger a pre-loaded animation on the client-side. This means that the client and server always have to be the same version to function correctly.



```
1 reference
private void ClientVisuals()
{
    if (networkPlayerState.Value==PlayerState.Walk)
    {
        animator.SetFloat("Walk", 1);
    }else if (networkPlayerState.Value == PlayerState.ReverseWalk)
    {
        animator.SetFloat("Walk", -1);
    }
    else
    {
        animator.SetFloat("Walk", 0);
    }
}
```

Figure 33. Client-side animation script

5 CONCLUSION

This thesis project was aimed to explore the feasibility of a multiplayer project with the assistance of the newly released Unity Netcode for GameObject. Throughout the thesis, the game development tools and process used in the project have been described.

Within 2 weeks of development, the game was able to be played online, with the player's ability to move in the game world, and seen by others players while doing so. Twenty clients are able to connect in and out of the game easily without crashes or lag (local networking might have played a major role). Players could interact with each other with Unity's physics engine.

However, the result of the project fell short of the set of expectation in chapter 4.1 where the game's core pillars is discussed. The custom logic of dividing a chasing and running side, and the winning condition along with common user interface update could not fit in the scope. The base project is not entirely reusable because many heavy networking logics are individual to the game itself. The project scope was cut down as many unresolved bugs and technological roadblocks emerge during the development.

The author believes this solution has potentials, being the officially supported and commercially advertised networking solution of Unity. During the time the project occurred, there was not much community support surrounding Netcode which was bound to grow. Considering that the project was powered by one person, a small development team has good possibility of finishing said project (with previous knowledge of Netcode and a refined working process.

REFERENCES

[1] Tom, W. 2021. The Games Market and Beyond in 2021: The Year in Numbers. Web page. Available at: https://newzoo.com/insights/articles/the-games-market-in-2021-the-year-in-numbers-esports-cloud-gaming/ [Accessed 25 May 2023].

[2] Facebook. 2021. Games Marketing Insights for 2021. Pdf report. Available at: https://www.facebook.com/fbgaminghome/marketers/find-new-players/advertising-hub/gaming-marketing-insights-2021 [Accessed 25 May 2023].

[3] Eric, C.P. 2019. How Unity built the world's most popular game engine. Available at: https://techcrunch.com/2019/10/17/how-unity-built-the-worlds-most-popular-game-engine/#:~:text=Unity%20was%20founded%20in%20Copenhagen,based%20game%20developers%20like%20himself [Accessed 25 May 2023].

[4]Romain, D. 2018. Unity CEO says half of all games are built on Unity. Website. https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/ [Accessed 25 May 2023].

[5]Clayton, M.C, Micheal, E.R & Rory, M. 2015. What Is Disruptive Innovation? Website. Available at: https://hbr.org/2015/12/what-is-disruptive-innovation [Accessed 25 May 2023].

[6]UXPIN. n.d. Video Game Prototyping – How To Do It and Why You Should! Available at: https://www.uxpin.com/studio/blog/why-and-how-to-use-video-game-prototyping/ [Accessed 25 May 2023].

[7]Max, P. 2017. Design Pillars – The Core of Your Game. Available at: http://www.maxpears.com/2017/09/02/design-pillars-the-core-of-your-game/ [Accessed 25 May 2023].