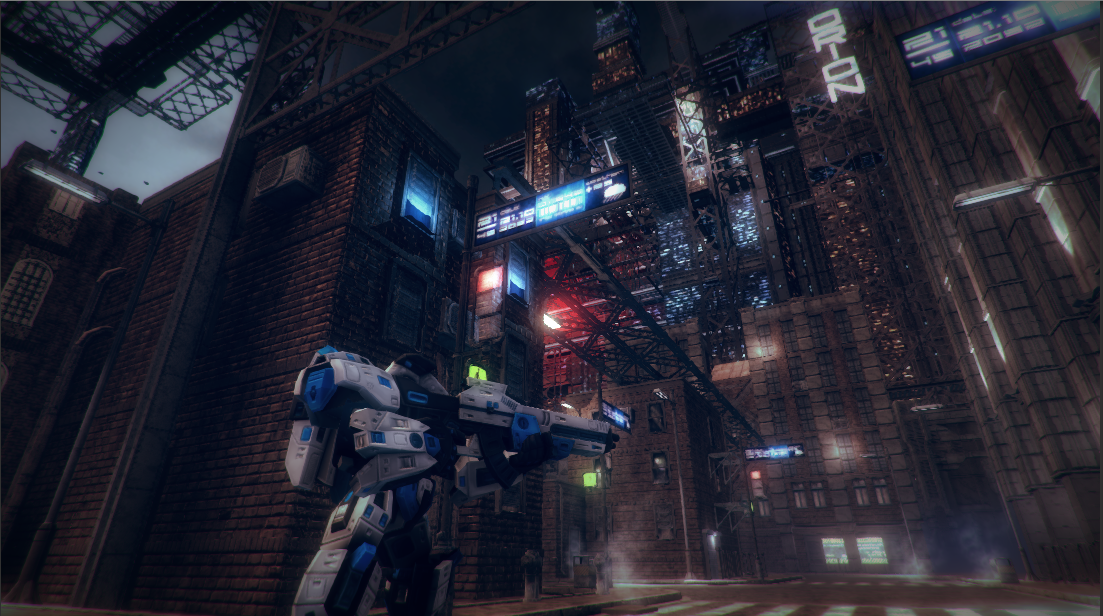
# Creating Your First Multiplayer Game in Unity



Shamim Akhtar

## Introduction

A multiplayer video game is a video game in which more than one person can play in the same game environment at the same time, either locally or online over the internet. In this worksheet, we are going to create our first online multiplayer game using Unity with Photon Unity Networking (PUN2) plugin.

## Photon Unity Networking 2

Photon is a real-time multiplayer game development framework that is fast, lean and flexible. Photon consists of a server and multiple client SDKs for major platforms.

Photon Unity Network (PUN) is a Unity specific, high-level solution based on the Photon engine. PUN2 package is compatible with the managed Photon Cloud service, which runs Photon Servers for you. You can have a free account to create multiplayer games using PUN2.

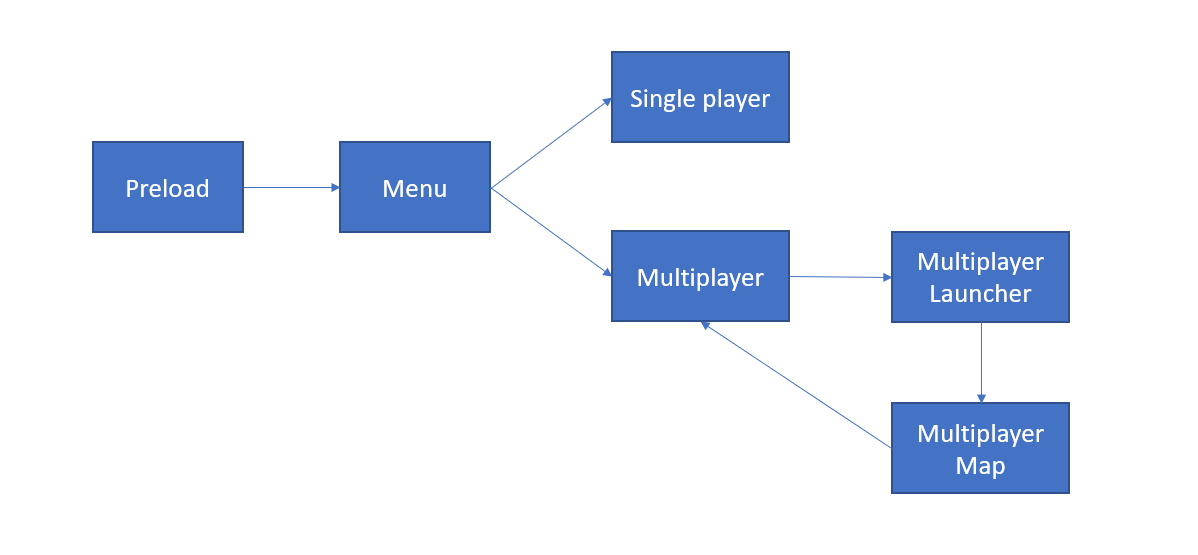
In this worksheet, we will use Photon to create our multiplayer game. We will start with an empty project and build from scratch our first multiplayer game. We will then import our player and the map from Week 1-3 to port our singleplayer third-person shooter game into a multiplayer third-person shooter game. Along the way, you will learn the various concepts and terminologies associated with a multiplayer game. You will also learn about the design considerations for creating networked games.

## Unity Project

Create a new Unity 3D project and name it **Week6\_Multiplayer**.

### The Scene Structure

Before we start with our implementation, let’s give a thought on how to structure our scenes for a multiplayer game. Discuss in class before you proceed with the following sections.



The above picture shows the different scenes that we are going to implement.

#### Preload

In every Unity project, you should have a **Preload** scene. This scene should be the first scene that your game should load. A **Preload** scene is an empty scene that creates a Singleton object associated with your game and keeps a **DontDestroyOnLoad** game object. **DontDestroyonLoad** only affects top-level objects in the scene. You can’t call it on a child where the parent does not have a **DontDestroyonLoad**.

A **Preload** scene is necessary so that you can do game level stuff, like connecting to a database, creating ambient sound, game scoring etc. here and separate from the main game scenes. **Preload** scene is also the scene which should allow game level access to specific objects.

#### Menu

This scene is the main Menu for our game. From this scene, you can either enter your singleplayer or to your multiplayer game.

#### Single Player

Our singleplayer game. Ideally, here you will have your singleplayer game menu from where you will select the different types of singleplayer games (like campaign modes).

#### Multiplayer

Our multiplayer game menu.

#### Multiplayer Launcher

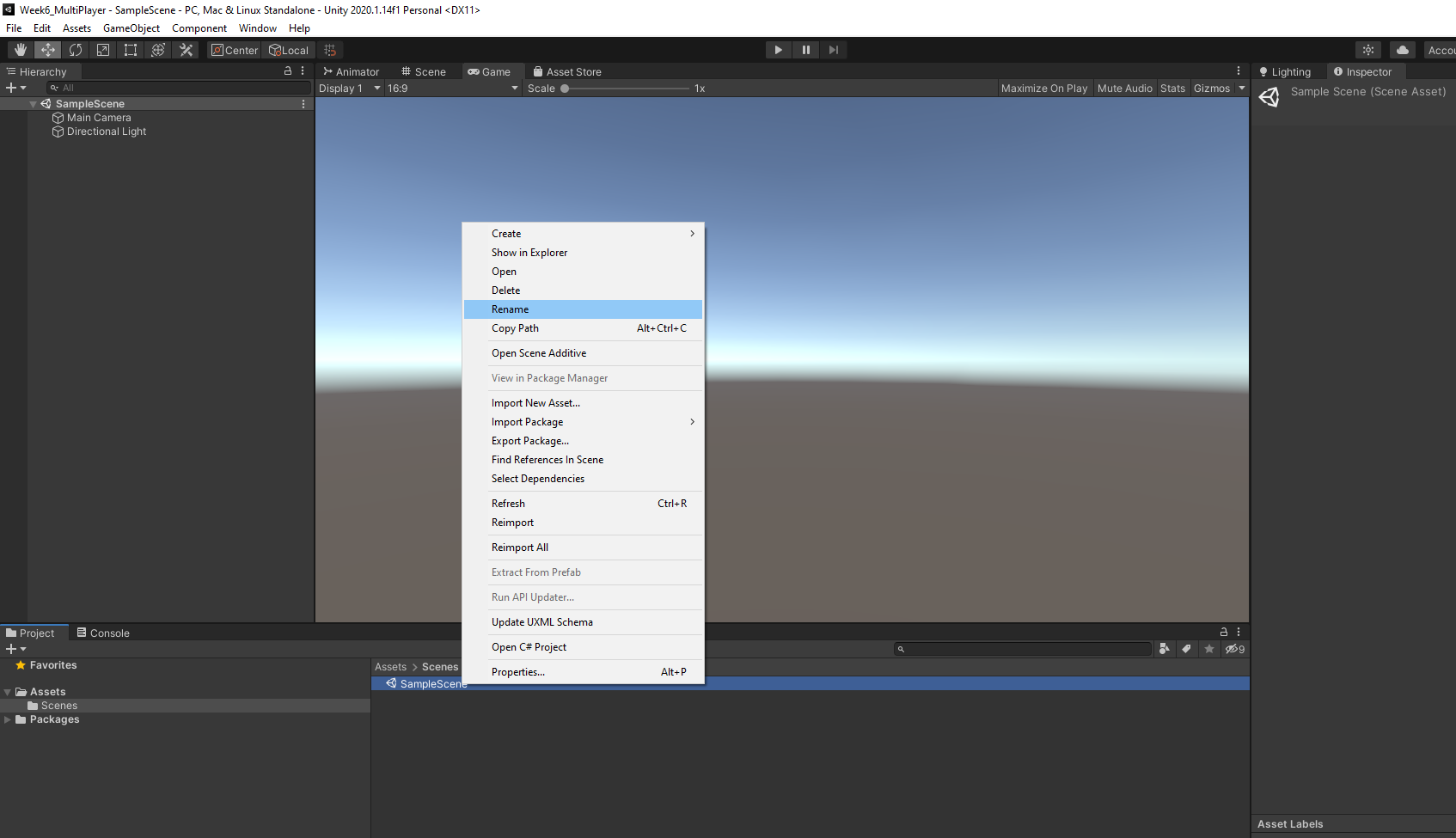
The scene that launches the connection to the server and takes care of other connection-related activities. We will learn about it shortly in the following sections.

#### Multiplayer Map

This scene is going to be our main scene where the multiplayer game happens.

## The Preload Scene

When we created the empty project in Unity 3D, it came with a scene named **SampleScene**. Rename this scene to **0Preload.** The 0 in front is to keep it on top in our list of scenes.



Create the following folders in **Assets**.

**Scripts**

**Resources**

Now, copy the **Singleton.cs** file from your LMS into the Scripts folder.

Create an empty game object and name it **GameApp**.

Attach a new script component called **GameApp** to this empty game object and move the file to the **Scripts** folder. Double click and open the **GameApp.cs** file.

We will make GameApp object a Singleton. For this change the class and derive it from **Singleton<GameApp>** rather than from **MonoBehaviour**.

public class GameApp : Singleton<GameApp>

### Pausing Game

We are now going to implement pausing our game. Usually, we can pause a game by pressing the **Escape** button on your keyboard. So, let’s go ahead and implement the pause game feature in **GameApp**.

Add a property of type boolean and call it **GamePaused**

public bool GamePaused

{

get { return mPause; }

set

{

mPause = value;

mOnPause?.Invoke(GamePaused);

if (GamePaused)

{

Time.timeScale = 0;

}

else

{

Time.timeScale = 1;

}

}

}

Implement the following in **GameApp** **Update** method.

void Update()

{

if (Input.GetKeyDown(KeyCode.Escape))

{

GamePaused = !GamePaused;

}

}

Then by default in **Start,** we make the **GamePaused** false.

void Start()

{

GamePaused = false;

}

For convenience, we add **OnEnable** and **OnDisable** method. **OnEnable** is called when the game starts (at the beginning), and **OnDisable** is called when the game terminates. In these functions, we can add some callbacks by using delegates. For now, we will create a new function called **OnSceneLoaded** and add to **SceneManager.sceneLoaded** delegate.

// called first

void OnEnable()

{

Debug.Log("OnEnable called");

SceneManager.sceneLoaded += OnSceneLoaded;

}

// called when the game terminates

void OnDisable()

{

Debug.Log("OnDisable");

SceneManager.sceneLoaded -= OnSceneLoaded;

}

void OnSceneLoaded(Scene scene, LoadSceneMode mode)

{

Debug.Log("OnSceneLoaded - Scene Index: " + scene.buildIndex + " Scene Name: " + scene.name);

Debug.Log(mode);

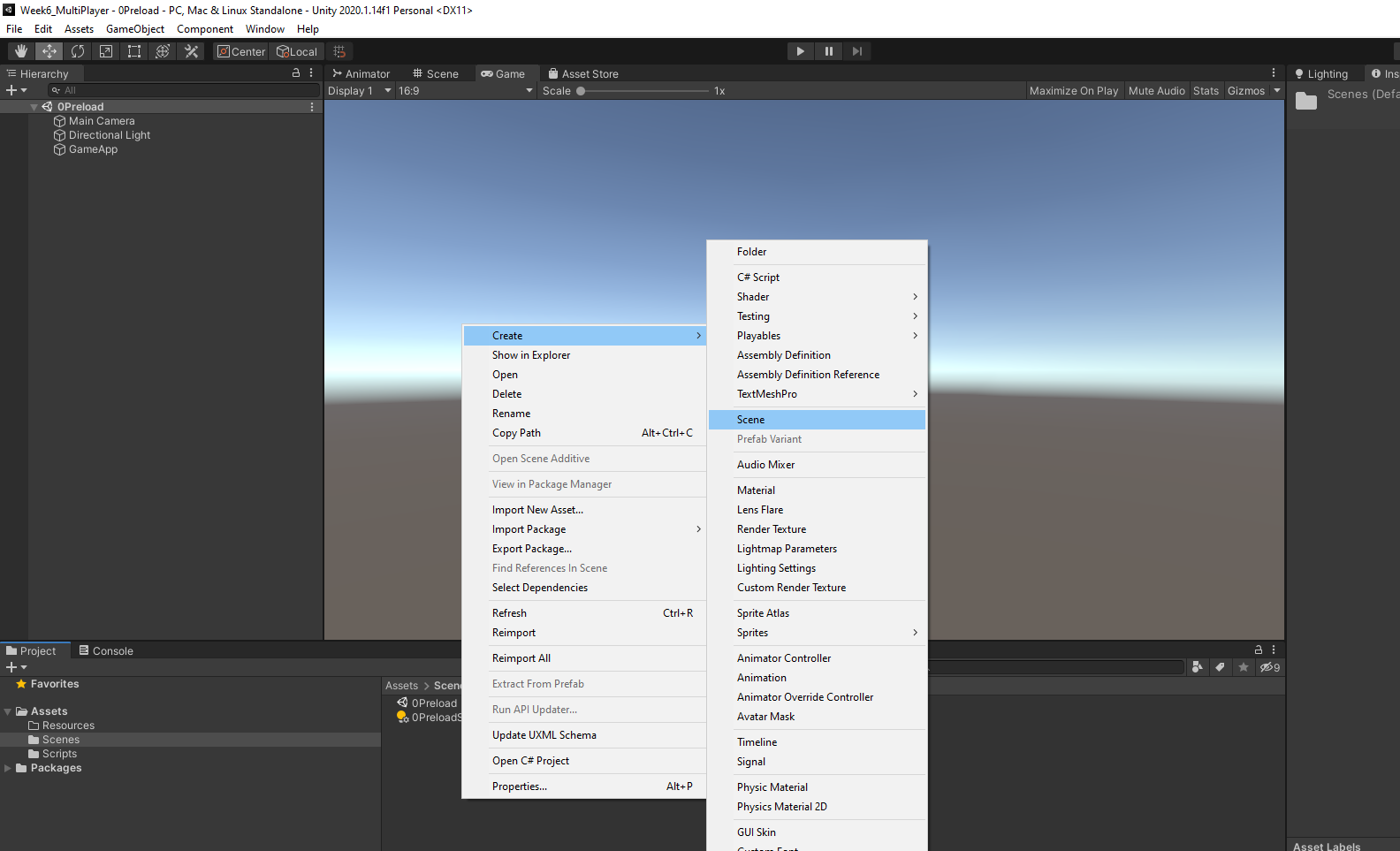
}

The **OnSceneLoaded** method simply outputs the Build Index and the Scene Name into the Debug console.

**Discuss in class the above code.**

## The Menu Scene

Right-click on the Scenes folder and create a new Scene called the **Menu**.



Now modify the **GameApp** **Start** method to load the **Menu** scene directly.

void Start()

{

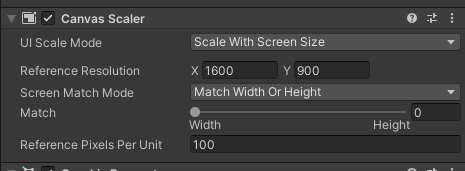
GamePaused = false;

SceneManager.LoadScene("Menu");

}

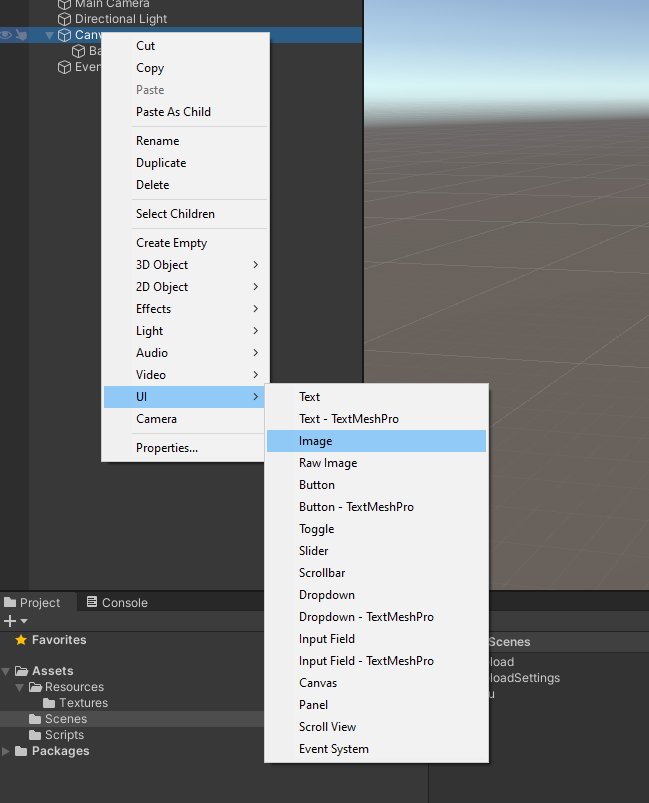
### The Menu

Create a new Canvas. Set the following settings.

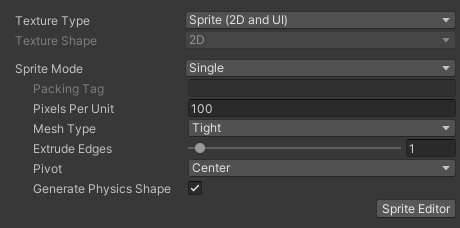


#### The Background

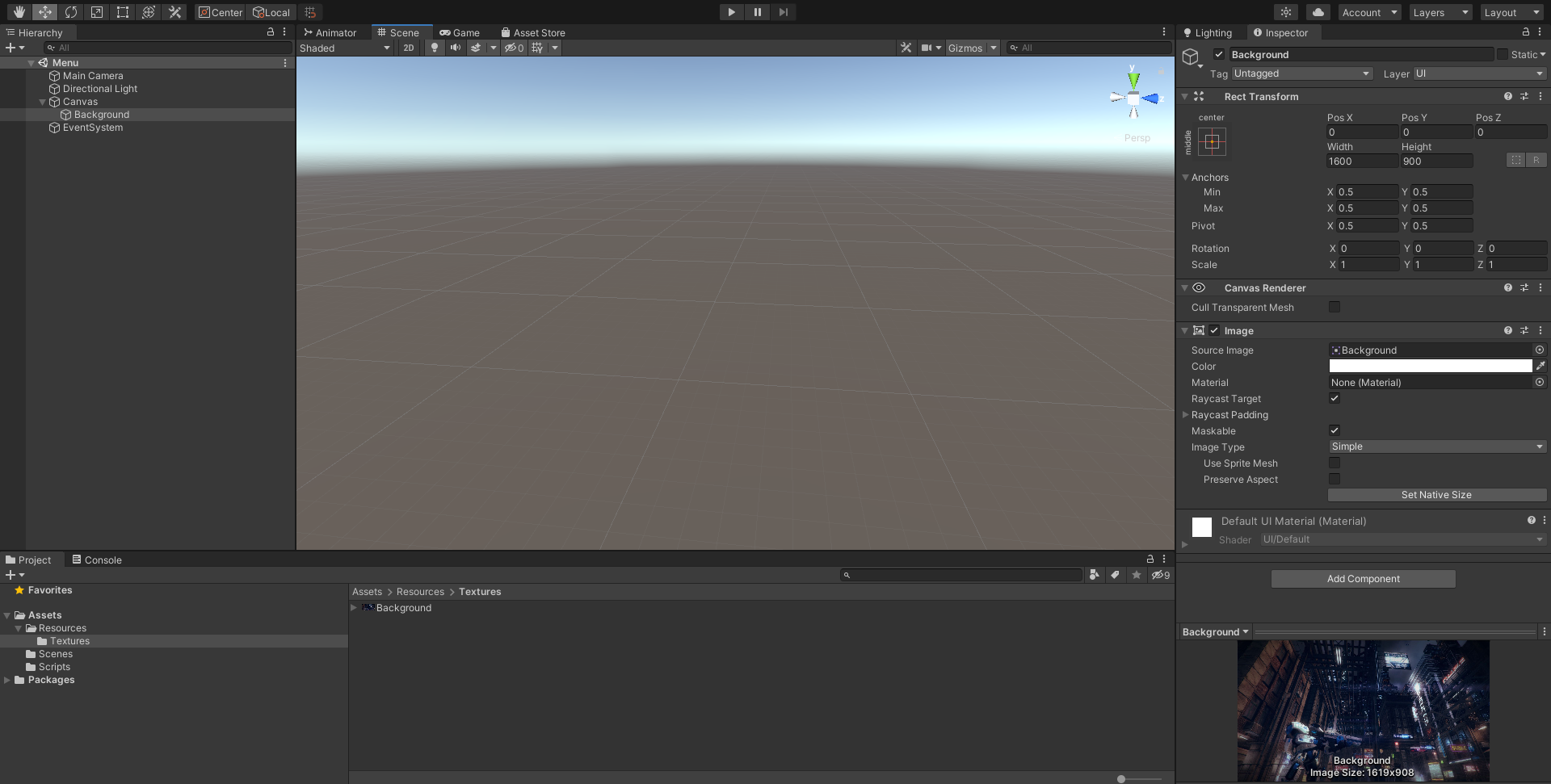
Select canvas, right-click and add a new Image. Name it **Background**.



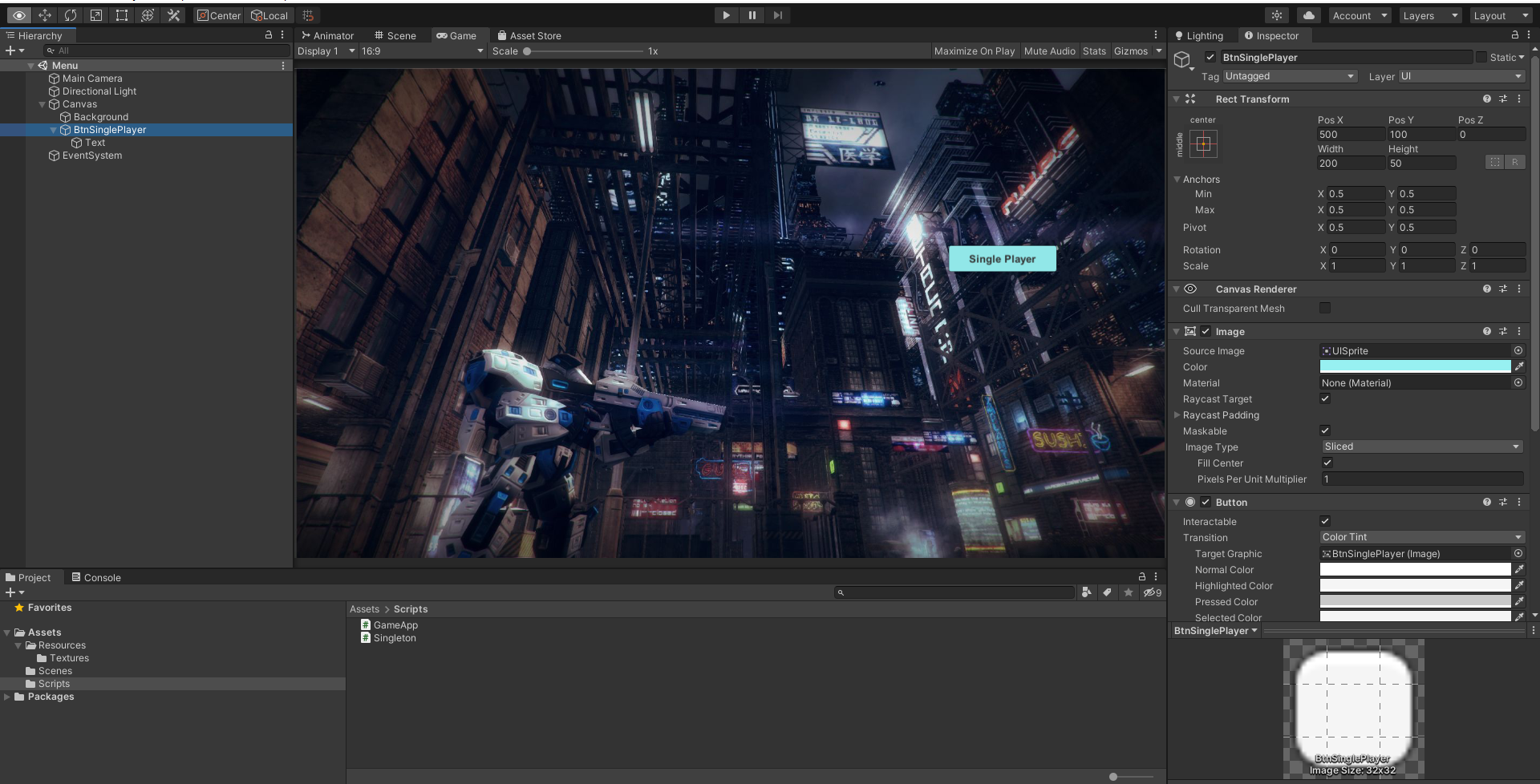
Now, download the **Background.jpg** file from LMS and put it inside a folder named **Textures** in **Resources**. Select the image and convert it to **Sprite (2D and UI).** Click **Apply** to apply the changes.



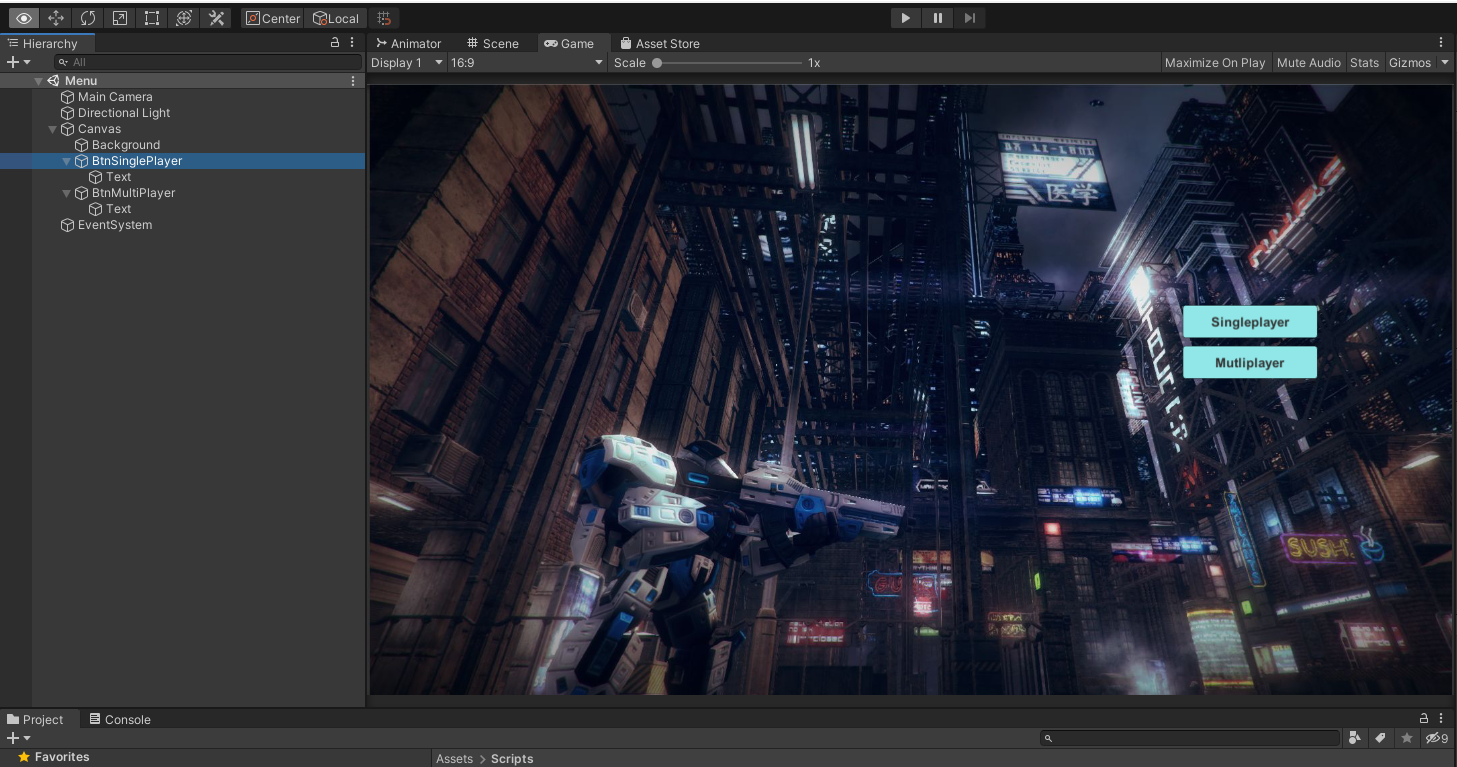
Select **Background** from the Hierarchy and then drag and drop the Sprite to the **Source Image,** as shown below.



#### The Buttons

Right-click on the canvas and add a Button. Name it **BtnSinglePlayer**. Position it as shown in the picture below. Change the text of the button to **Singleplayer**.

Duplicate this button one more time and change the name to **BtnMultiPlayer**. Change the text to **Multiplayer**. You can choose the button colour or image, the font colour, size etc.



Create a new script file called **Menu** and add as a component to the **Canvas**. Double click and open the **Menu.cs** file in Visual Studio.

Create two methods **OnClickBtnSinglePlayer** and **OnClickBtnMultiPlayer**.

public void OnClickSinglePlayer()

{

Debug.Log("Loading singleplayer game");

}

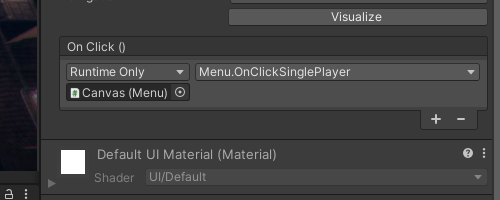
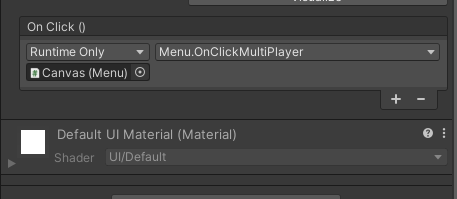
public void OnClickMultiPlayer()

{

Debug.Log("Loading multiplayer game");

}

Associate the two functions to the **On Click** of **BtnSinglePlayer** and **BtnMultiPlayer** respectively.

## The Multiplayer Launcher Scene

Create a new scene and name it **Multiplayer\_Launcher**. Go to **Menu** and amend the method **OnClickMultiPlayer** as follows:

public void OnClickMultiPlayer()

{

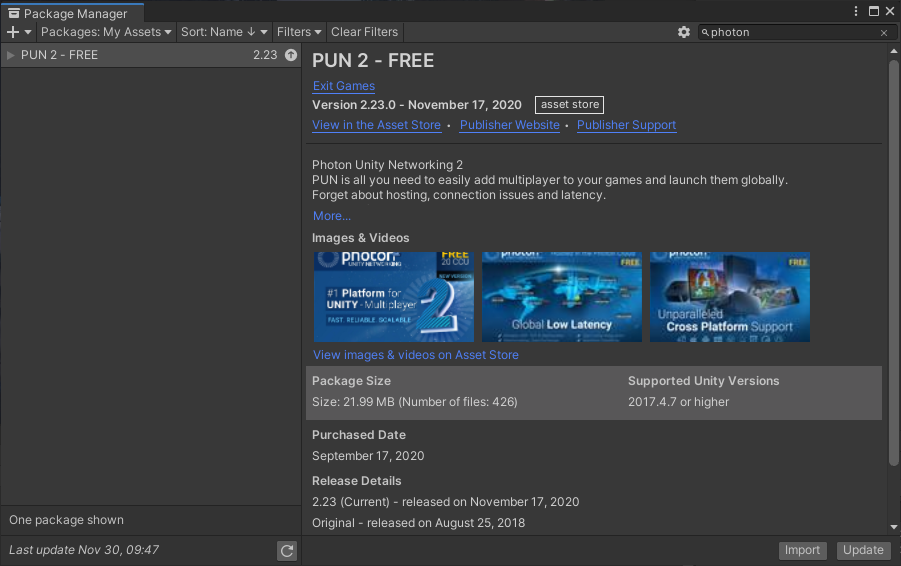
Debug.Log("Loading multiplayer game");

SceneManager.LoadScene("Multiplayer\_Launcher");

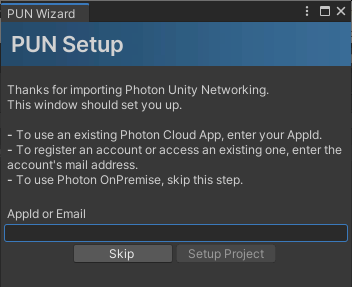
}

### PUN

Now it is time for us to start with the actual coding for our multiplayer networked game. For this, we will now download and install the PUN2 package from Unity Package Manager. Open Unity Package Manager and search for Photon.

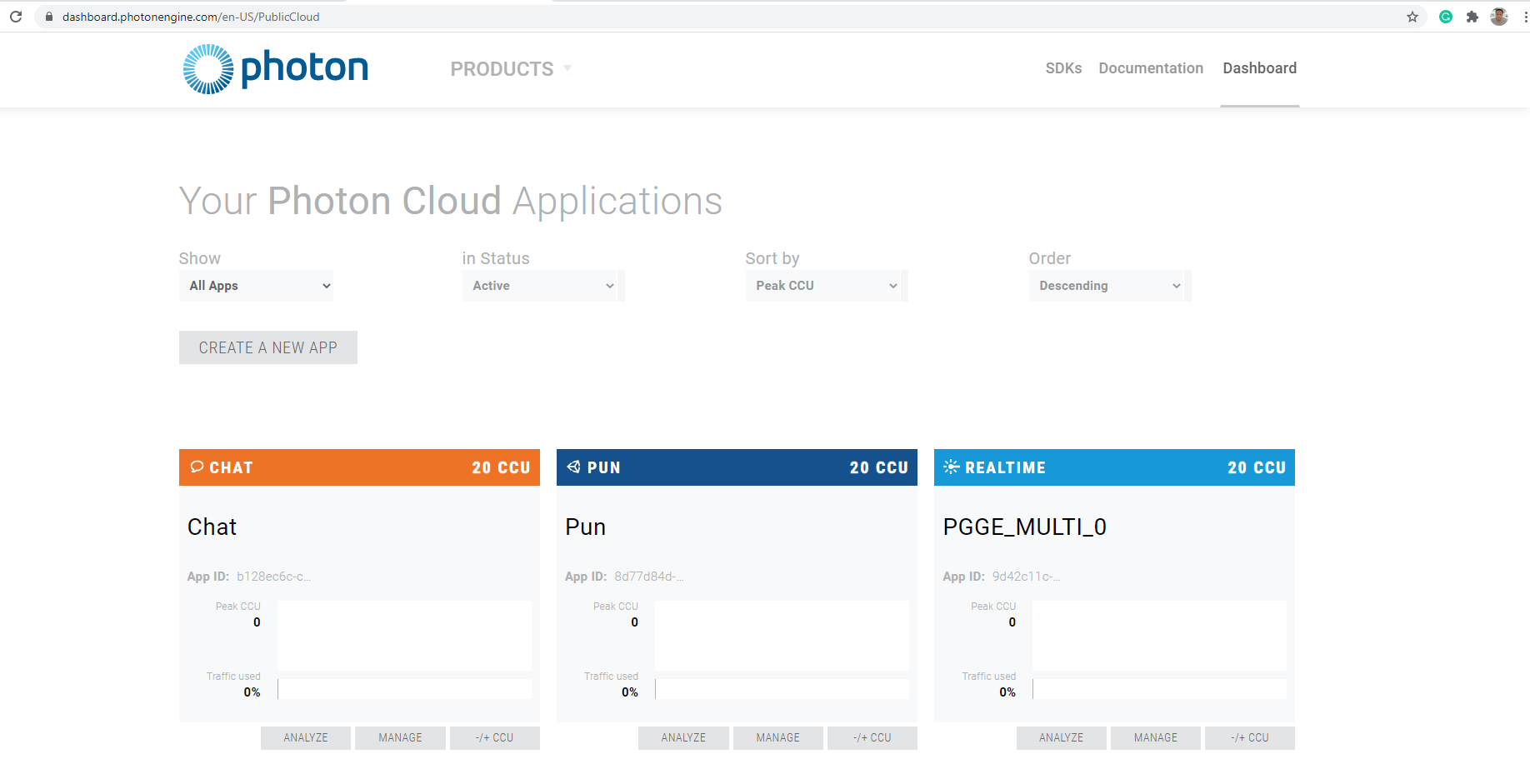


Download and import the package. Let Unity recompile when you have imported all the assets.



The PUN Setup Wizard is there to help you with the network settings and offers a convenient way to get started with our multiplayer game: The Photon Cloud! Using the Cloud with the "Free Plan" is free and without obligation. Enter your email address and move on. New accounts get an "AppId" right away. If you have already registered before then log into your Photon account and find your AppId in the dashboard.

It will look something like this in Photon dashboard webpage. The below picture is from my account.



#### Application IDs & Game Version

As everyone connects to the same servers, there must be a way to separate your players from everyone else's. Each game title gets an "AppId" in the Cloud. Players will only ever meet other players with the same "AppId".

There is also a "Game Version", a string that will separate players with older clients from those with newer ones.

#### Regions

The Photon Cloud is organized in separate Regions across the globe for optimal connections between players.

Each region is separate from all others. You should remember when working with remote teams spread in different areas. Make sure you end up in the same region.

**PUN 2 helps you by figuring out a "Dev Region", which is used for all development builds.**

#### Rooms

The Photon Cloud is "room-based", meaning there is a limited number of players per match separated from anyone else. In a room, everyone receives whatever the others send (unless you send messages to specific players). Outside of a room, players are not able to communicate.

The best way to get into a room is to use Random Matchmaking. We will get to that in a while.

All rooms have a name as an identifier. Unless the room is full or closed, we can join it by name. Conveniently, the Master Server can provide a list of rooms for our app.

#### Lobby

The lobby for your application exists on the Master Server to list rooms for your game. It does not enable players to communicate with one another! The lobby is where all the players gather together to either create their rooms or find rooms to join in.

In this worksheet, we will not make use of the lobby or display the list of rooms. Instead, we will join a random room if one is available, or create a new room if no existing room can be joined (rooms can have a maximum capacity, and so they can potentially be all full).

### PUN Callbacks

PUN is very flexible with callbacks and offers two different implementations.

#### Implementing Callback Interfaces

PUN provides C# interfaces that you can implement in your classes:

* **IConnectionCallbacks**: connection related callbacks.
* **IInRoomCallbacks**: callbacks that happen inside the room.
* **ILobbyCallbacks**: lobby related callbacks.
* **IMatchmakingCallbacks**: matchmaking related callbacks.
* **IOnEventCallback**: a single callback for any received event. This is equivalent to the C# event OnEventReceived.
* **IWebRpcCallback**: a single callback for receiving WebRPC operation response.
* **IPunInstantiateMagicCallback**: a single callback for instantiated PUN prefabs.
* **IPunObservable**: PhotonView serialization callbacks.
* **IPunOwnershipCallbacks**: PUN ownership transfer callbacks.

Callback interfaces must be registered and unregistered.

Call PhotonNetwork.AddCallbackTarget and PhotonNetwork.RemoveCallbackTarget (likely within OnEnable() and OnDisable() respectivly).

#### Extending MonoBehaviourPunCallbacks

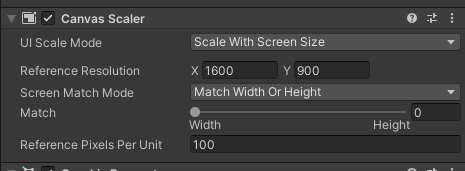
The other technique is to extend the MonoBehaviourPunCallback. This technique is a more convenient way where, instead of creating a class that derives from MonoBehaviour, derive the class from MonoBehaviourPunCallbacks. Then we override the specific properties and virtual methods for us to use at our convenience. It is convenient because we can be sure that we don't have any typos, and we don't need to implement all methods.

Read more about Photon Callbacks and Interfaces from Photon Official documentation.

### The Lobby UI

Now we will focus on creating a UI for the lobby. We will keep it simple and only have one **Input Field for the Player name**, a **Button to join the room** and load the multiplayer map, and a **Text that shows the Connection Progress**.

Create a new Canvas. Set the following settings.



#### The Background

Follow the steps from section The Background from the Menu scene. You can choose a different background image for this if you wish to.

#### The Player Name

An essential requirement for a multiplayer game is to let the user input their name so that other players know who they are. We will use **PlayerPrefs** here to remember the word so that when the user opens the game again, we can autofill the name.

1. Select the Canvas, right-click and add a new Input Field UI item. Position it accordingly.
2. Add a new script component and name it **PlayerNameInput**. Move it to the **Scripts** folder.
3. Double click and open the **PlayerNameInput** file in Visual Studio.
4. We want the input name from the input field to be associated with the nickname for PUN. We will need to add **Photon.Pun** namespace into our PlayerNameInput script file. Go ahead and add the using Photon.Pun namespace in your file header.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using Photon.Pun;

1. Now. Add two variables. One constant variable for storing the key to the PlayerPrefs. Set the value to “PlayerName” and the other the InputField that captures the name entered by the user.

const string playerNamePrefKey = "PlayerName";

private InputField mInputField;

1. In the **Start** method, try to get the value of the InputField text from the PlayerPrefs first and fill up. If the user keys in a name then replace the existing name with the newly keyed in name and store in the PlayerPrefs.

void Start()

{

string defaultName = string.Empty;

mInputField = this.GetComponent<InputField>();

if (mInputField != null)

{

if (PlayerPrefs.HasKey(playerNamePrefKey))

{

defaultName = PlayerPrefs.GetString(playerNamePrefKey);

mInputField.text = defaultName;

}

}

PhotonNetwork.NickName = defaultName;

}

1. Create a **SetPlayerName** method that we will use as a callback from the InputField’s **OnValueChanged**. In this method, we will take the text value from the InputField and set to the **PhotonNetwork.Nickname** as well as to the **PlayerPrefs** dictionary.

public void SetPlayerName()

{

string value = mInputField.text;

if (string.IsNullOrEmpty(value))

{

Debug.LogError("Player Name is null or empty");

return;

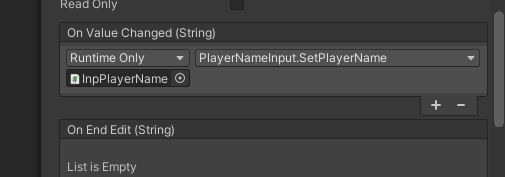
}

PhotonNetwork.NickName = value;

PlayerPrefs.SetString(playerNamePrefKey, value);

}

1. Associate the **SetPlayerName** function to **OnValueChanged.**



#### The Join Button

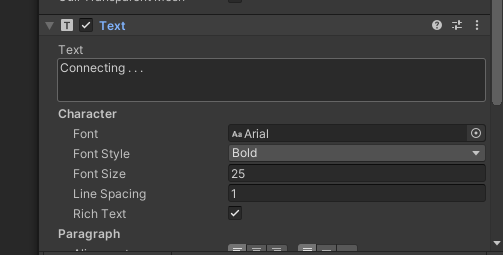
In this section, we will create the Join button that will allow us to connect to a server and join a room. Later on, I will discuss the logic of joining a room.

1. Select the Canvas, right-click and add a new Button UI item. Position it accordingly and name the button **BtnJoinRoom**.
2. We will associate the OnClick function in a while. Before that, we will have to create the necessary functionality for creating a multiplayer game using PUN.

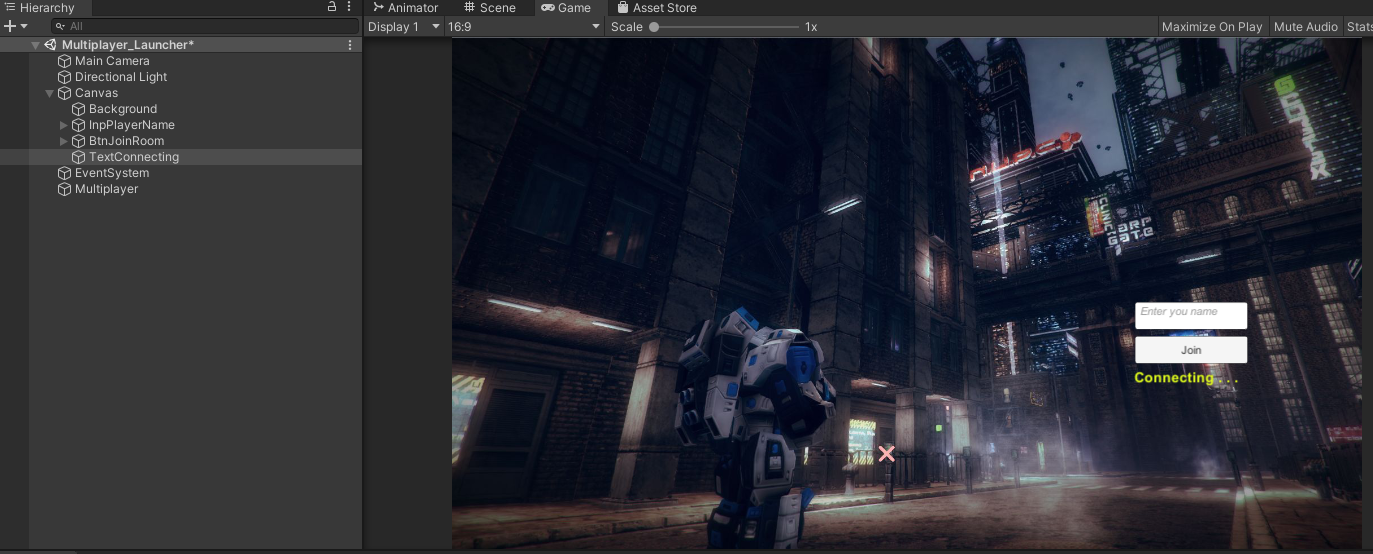
#### The Connecting text

In this section, we will create the **Connecting ...** text that will allow us to show the status to the user while the connection is in progress.

1. Select the Canvas, right-click and add a new Text UI item. Position it accordingly and name the text to **TextConnecting**.
2. Go to the Text field of **TextConnecting** and set the value to **Connecting . . ..** You can change the Font and the Font Size to something that suits you.



After creating all these UI fields, the screen should look something like below.



### The ConnectionController Script

In this section, we will create the script that creates and maintains a connection, the callbacks when it disconnects and the callbacks associated with joining a room. We will make these scripts in a dummy (empty) game object.

1. Right-click on the Hierarchy and add a new empty game object. Name it **Multiplayer**. This game object is a dummy game object. We will use this dummy game object to associate the scripts required for the multiplayer game.
2. Select **Multiplayer** game object from the Hierarchy and add a new Script Component. Name this script **ConnectionController**. Move the file to the **Scripts** folder. Double click and open the file in Visual Studio.
3. The required namespaces

using UnityEngine;

using Photon.Pun;

using Photon.Realtime;

1. For clarity and to separate our scripts for the multiplayer implementation, we will put our codes in namespace **PGGE.Multiplayer**.

namespace PGGE.Multiplayer

{

public class ConnectionController : MonoBehaviourPunCallbacks

{

\*\*\*

}

}

1. Add the following variables for the **ConnectionController** script.

const string gameVersion = "1";

public byte maxPlayersPerRoom = 4;

public GameObject mConnectionProgress;

public GameObject mBtnJoinRoom;

public GameObject mInpPlayerName;

bool isConnecting = false;

1. **AutomaticallySyncScene** in **Awake** method. We will set the value of to true. It means that the currently loaded scene is the same across all clients. If the MasterClient used PhotonNetwork.LoadLevel to load another scene then all other clients will load this new scene as well.

void Awake()

{

// this makes sure we can use PhotonNetwork.LoadLevel() on

// the master client and all clients in the same

// room sync their level automatically

PhotonNetwork.AutomaticallySyncScene = true;

}

1. In the Start method, we set the **mConnectionProgress** text to inactive so that it is not displayed. We will only set this to active when we are connecting to the server.

void Start()

{

mConnectionProgress.SetActive(false);

}

1. In this step, we will implement the **Connect** method. In this method, we will make a connection to a game server. If we are already connected to a server, then we will just join a random room.

public void Connect()

{

mBtnJoinRoom.SetActive(false);

mInpPlayerName.SetActive(false);

mConnectionProgress.SetActive(true);

// we check if we are connected or not, we join if we are,

// else we initiate the connection to the server.

if (PhotonNetwork.IsConnected)

{

// Attempt joining a random Room.

// If it fails, we'll get notified in

// OnJoinRandomFailed() and we'll create one.

PhotonNetwork.JoinRandomRoom();

}

else

{

// Connect to Photon Online Server.

isConnecting = PhotonNetwork.ConnectUsingSettings();

PhotonNetwork.GameVersion = gameVersion;

}

}

1. In this step, we will implement the OnConnectedToMaster and OnDisconnected methods. Create two methods that override the OnConnectedToMaster and OnDisconnected.

public override void OnConnectedToMaster()

{

if (isConnecting)

{

Debug.Log("OnConnectedToMaster() was called by PUN");

PhotonNetwork.JoinRandomRoom();

}

}

We keep our logic very simple In the above method, on connected, we just join a random room. If you want a more complicated and better logic, then you can create a sophisticated matchmaking mechanism. You can refer to Matchmaking Guide in Photo documentation for a better understanding of matchmaking capabilities.

public override void OnDisconnected(DisconnectCause cause)

{

Debug.LogWarningFormat("OnDisconnected() was called by PUN with reason {0}", cause);

isConnecting = false;

}

1. In this step, we will implement **OnJoinRandom** fail. This callback is called when Photon fails to join a random room. It can happen either because all the rooms are full or because there are no rooms. In both situations, we are okay because we can create a new room.

public override void OnJoinRandomFailed(short returnCode, string message)

{

Debug.Log("OnJoinRandomFailed() was called by PUN. " +

"No random room available" +

", so we create one by Calling: " +

"PhotonNetwork.CreateRoom");

// Failed to join a random room.

// This may happen if no room exists or

// they are all full. In either case, we create a new room.

PhotonNetwork.CreateRoom(null,

new RoomOptions

{

MaxPlayers = maxPlayersPerRoom

});

}

1. Finally, we implement the callback for **OnJoinedRoom**. This method is called when a player joined a room successfully. Our implementation loads the level if it is a master client.

public override void OnJoinedRoom()

{

Debug.Log("OnJoinedRoom() called by PUN. Client is in a room.");

if (PhotonNetwork.IsMasterClient)

{

Debug.Log("We load the default room for multiplayer");

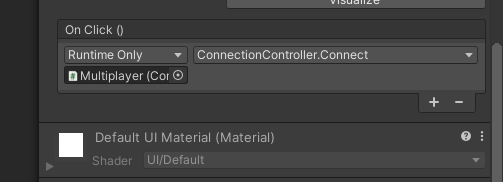
PhotonNetwork.LoadLevel("MultiplayerMap00");

}

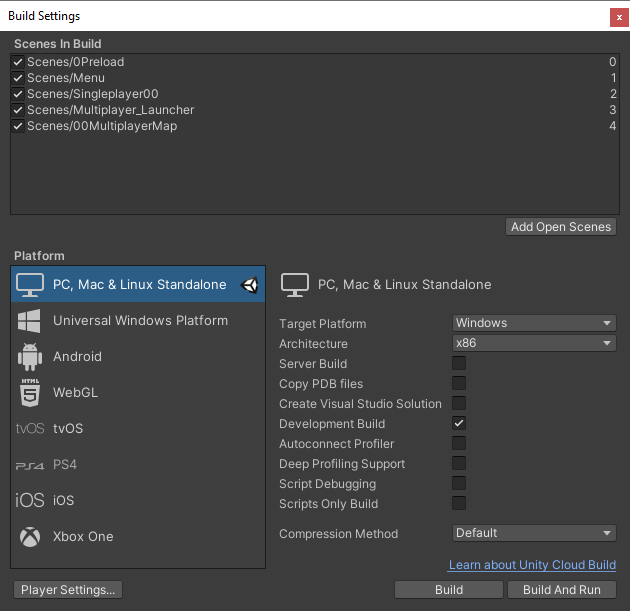
}

1. Associate the Join Button to Connect method. You may recall that we have created a Join button named **BtnJoinRoom**. The idea is that when the player clicks on this button, the game tries to make a connection to the game server. To do so, we will need to associate the **OnClick** callback of the button to the **Connect** method of the **ConnectionController** script.

Select the BthJoinRoom from the Hierarchy and then drag and drop the Multiplayer game object to its OnClick method. Then click on the dropdown and select **ConnectionController -> Connect**.



1. Before we click **Play** to test out implementation, create a new scene and name it **MultiplayerMap00**. Don’t forget to add this new scene to your Build Settings.



Now double click and select **0Preload** scene and click **Play**.

## Create Your First Multiplayer Scene

In this section, we will create our first multiplayer scene named **MultiplayerMap00**. For this, we will (a) port our scene, (b) port our third-person player and (c) port our third-person camera.

Open up your Week 3 project, go to **Assets** and click **Export Package**. Export All and select dependencies.



In case you are not able to export, or you did not complete your Week 3 worksheet, I have prepared the package for you. Go to LMS, and under Week 6 you will find the Week3\_Package. Download and import it into your project.

### Port Scene

Open **MultiplayerMap00** scene. We will use this scene for our multiplayer map. To load our map from Week 3, go to **Resources->Prefabs** and drag and drop **Map\_v1**.

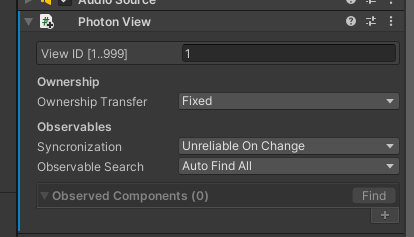
### Port Player

We have configured a SciFi player in our Week 1 to 3 worksheets. We are going to use the same player. To make the player compatible with multiplayer playing, we will need to make changes to its components and scripts.

Go to **Prefabs** folder and find the **SciFiPlayer** prefab. Drag and drop this into the scene and rename it to **SciFiPlayer\_Networked**. In this section, we will modify the **SciFiPlayer** prefab to make it adapt to multiplayer capabilities. This player prefab works as is, but now we are going to change it so that it works and complies when we use it within PUN environment. The modifications are very light; however, the concepts are critical. So this section is vital for your understanding.

#### PhotonView Component

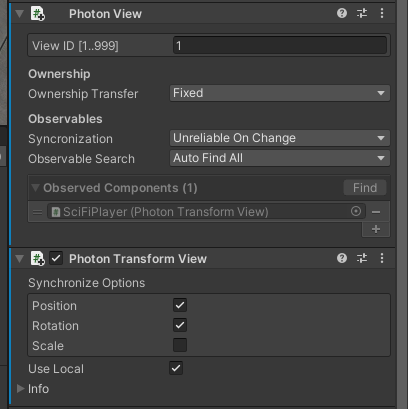
Select the **SciFiPlayer\_Networked** from the Scene Hierarchy. Go to the Inspector and add a new Component called the **PhotonView**. **PhotonView** connects the various instances on each computer and defines what components to observe and how to observe these components.



#### Transform Synchronization

We want to synchronize is the Position and Rotation of the player so that when a player is moving around, it behaves similarly on other clients. Photon provides an easy way to observe these transform changes across the network by using the PhotonTransformView component.

Add a PhotonTransformView to **SciFi\_Networked** game object.



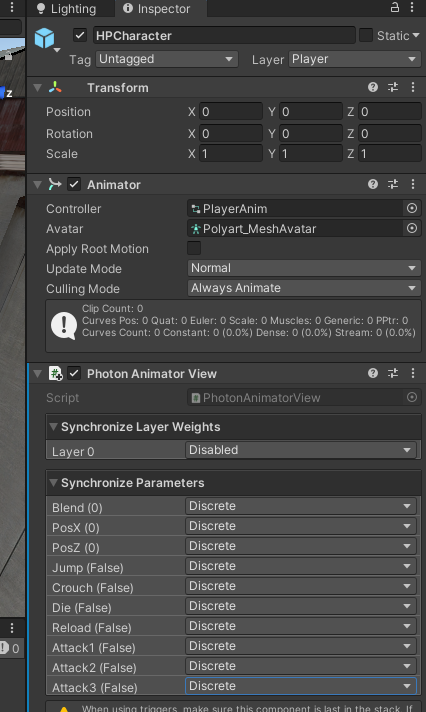
Note that once you add the PhotonTransformView component it becomes an observable in PhotonView. If it does not autopopulate then drag and drop the PhotonTransformView component into this box.

Check Synchronize Position and Synchronize Rotation checkboxes, as shown in the above picture.

#### Animator Synchronization

Next step for us would be to synchronize our player animation across the network. For this, we will use the **PhotonAnimatorView** component. The **PhotonAnimatorView** component makes networking setup easy and saves a lot of time and trouble. It allows us to configure which layer weights and which parameters we want to synchronize across the network. Layer weights only need to be synchronized if they change during the game, and it might be possible to get away with not synchronizing them at all. The same is valid for parameters. Sometimes it is possible to derive animator values from other factors. A speed value is an excellent example for this, you don't necessarily need to have this value synchronized, but you can use the synchronized position updates to estimate its value.

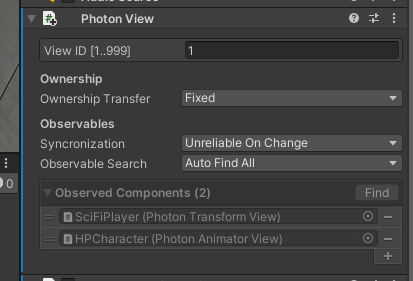
Note that for our case we have the Animator component, not in **SciFiPlayer\_Networked** game object but its child **HPCharacter** game object. So select **HPCharacter** and add the **PhotonAnimationView** component.



Set all to Discrete

Each value can be disabled or synchronized, either discretely or continuously. Discrete synchronization means that a parameter gets sent ten times a second (in OnPhotonSerializeView) by default. The receiving clients pass the value on to their local Animator. Continuous synchronization means that the PhotonAnimatorView runs every frame. For animation, it is probably not required to send data across the network continuously.

Select the SciFiPlayer\_Networked, and you will now find that the PhotonView component now observes two components, as shown below in the picture.



Now you should have two observablecomponents as shown on the left. If it does not autopopulate then drag and drop the PhotonTransformView and the PhotonAnimatorView components into this box.

#### The Player Script

You need to understand that in a multiplayer game you will have the same prefab instantiated across all the clients. However, only the actual player controls the character, and all other instances represent other users, playing on different computers. So the first hurdle with this in mind is "Input Management". How can we enable input on one instance and not on others and how to know which one is the right one? Enter the **IsMine** concept.

Double click the **Player** script and open in Visual Studio.

1. Add a new variable to cache the PhotonView component.

private PhotonView mPhotonView;

Then in the **Start** method, we get the PhotonView component and set it to mPhotonView.

mPhotonView = gameObject.GetComponent<PhotonView>();

1. In the **Update** method and the **Move** method, we check if **IsMine** is true and only then we go through the rest of the Update functionalities and the Move functionalities. If false, then we simply return from the function.

if (!mPhotonView.IsMine) return;

We have completed creating our multiplayer Player. Now we drag the **SciFiPlayer\_Multiplayer** to Prefabs folder and create a new **SciFiPlayer\_Multiplayer** prefab. Then we remove it from the scene.

### Port Third Person Camera

We do not have to do anything special with the third-person camera. However, we will not attach the **ThirdPersonCamera** script to the Main Camera in Unity Editor. Instead, we will attach this script to the Main Camera at runtime, after we Instantiate the **SciFiPlayer\_Multiplayer** prefab.

### Player Spawn Points

Before we instantiate the player lets create some spawn points where we can spawn our player randomly.

Right-click on the Scene Hierarchy and add a new empty game object. Name it **PlayerSpawnPoints**. Add a new script component to this game object called **PlayerSpawnPoints**. Double click and open the file in Visual Studio. This script will contain a List of transforms that comprised the possible player spawn positions and rotations.

public class PlayerSpawnPoints : MonoBehaviour

{

public List<Transform> mSpawnPoints = new List<Transform>();

public Transform GetSpawnPoint()

{

if (mSpawnPoints.Count == 0) return this.transform;

return mSpawnPoints[Random.Range(0, mSpawnPoints.Count)].transform;

}

}

### Player Instantiation

Create a new empty game object and name it **PlayerManager**. Reset the transform. Go to the Inspector and add a new script component called **PlayerManager**. Double click and open the file in Visual Studio.

1. Add the following variables to the file.

public string mPlayerPrefabName;

public PlayerSpawnPoints mSpawnPoints;

[HideInInspector]

public GameObject mPlayerGameObject;

[HideInInspector]

private ThirdPersonCamera mThirdPersonCamera;

1. We will instantiate our player in the **Start** method.

private void Start()

{

mPlayerGameObject = PhotonNetwork.Instantiate(mPlayerPrefabName,

mSpawnPoints.GetSpawnPoint().position,

mSpawnPoints.GetSpawnPoint().rotation,

0);

mPlayerGameObject.GetComponent<PlayerMovement>().mFollowCameraForward = false;

mThirdPersonCamera = Camera.main.gameObject.AddComponent<ThirdPersonCamera>();

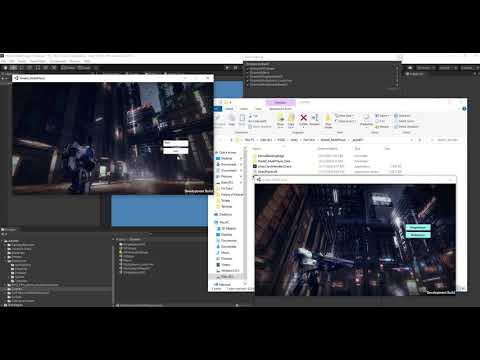
mThirdPersonCamera.mPlayer = mPlayerGameObject.transform;

mThirdPersonCamera.mDamping = 20.0f;

mThirdPersonCamera.mCameraType = ThirdPersonCamera.CameraType.Follow\_Track\_Pos\_Rot;

}

We have completed creating our first multiplayer scene for our third-person shooter game. Double click and select the **0Preload** scene. Go to **File->Build Settings** and click **Build**. Play the game. See the video below.

[](https://www.youtube.com/embed/FaHNNKUL6Q4?feature=oembed)

### Leaving a Room

Once a player in a room, we should allow the flexibility for the Player to leave the room. For this, we will create a button that when clicked, will let the player leave the room.

1. Open up the PlayerManager script file and add the **LeaveRoom** method as below.

public void LeaveRoom()

{

Debug.LogFormat("LeaveRoom");

PhotonNetwork.LeaveRoom();

}

1. Override and Implement the **OnLeftRoom** callback as follows.

public override void OnLeftRoom()

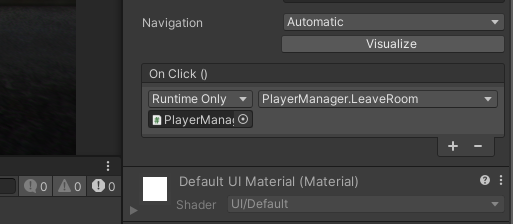
{

Debug.LogFormat("OnLeftRoom()");

SceneManager.LoadScene("Menu");

}

1. Create a Canvas and set the Reference Resolution to 1600 by 900.
2. Right-click and create a button. Name it BtnLeaveRoom. Position it appropriately at the top right corner of the screen.
3. Associate the **OnClick** method to **PlayerManager’s** **LeaveRoom** method.



Double click and select the **0Preload** scene. Go to **File->Build Settings** and click **Build.** Once the game is built, you can double click and play the game.