What Will we be Making?

In this course, we're going to create a multiplayer, 3D battle royale game in Unity, using Photon.

- **Photon Unity Networking** is the multiplayer framework we'll be using in Unity.
- We'll also be making a **player controller**, which will allow us to move around, jump, shoot, heal and add ammo through pickups.
- There will be 2 **pickups** that we can place around the map. These can give the player health or ammo.
- Like most battle royale games, we'll also have a **force field.**
- To connect to a game, players will gather together in **lobbies** and find lobbies through a **lobby browser.**
- This will be made using Unity's **UI system.**

Now, let's get started on the project!

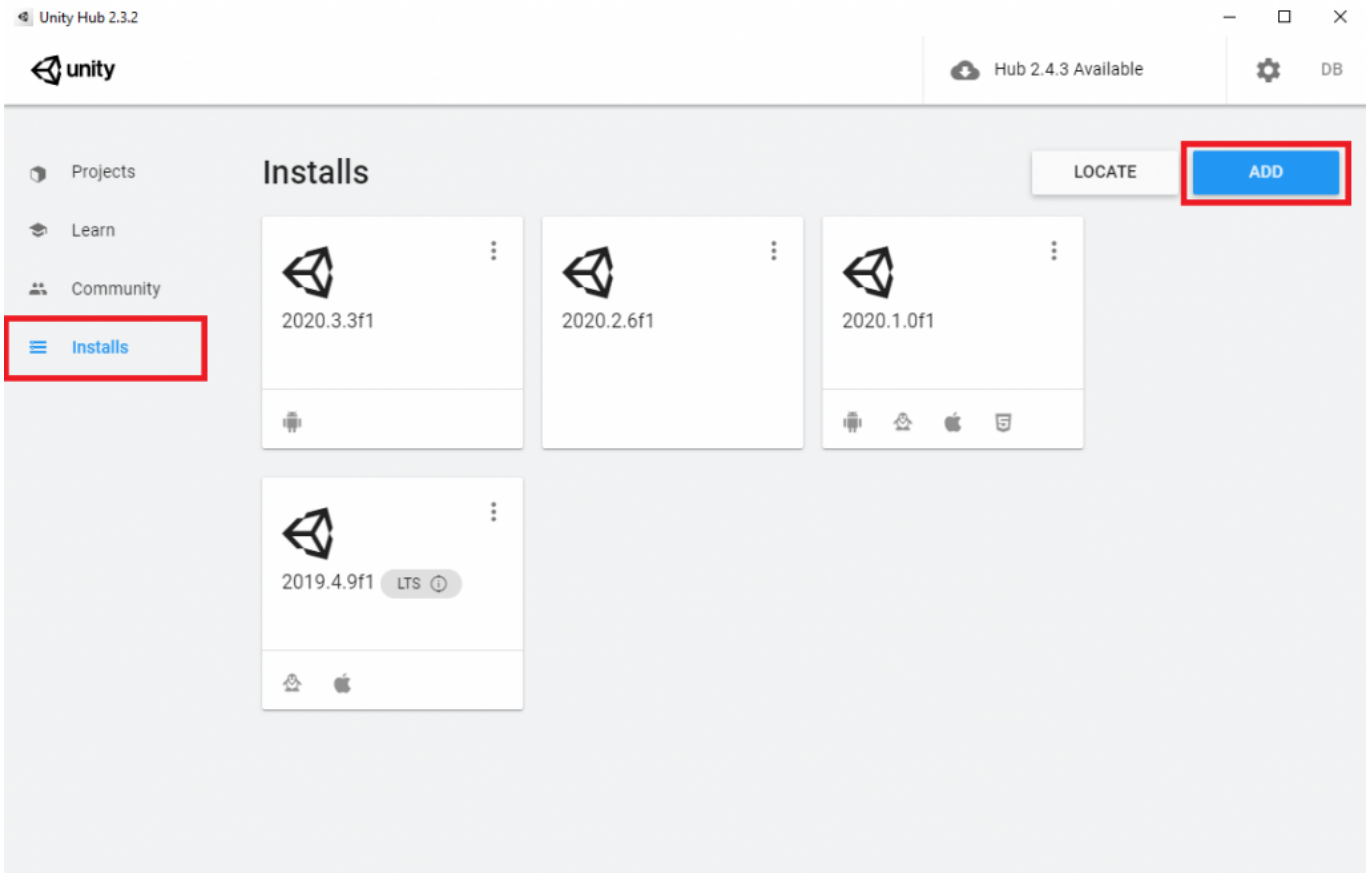
Course Updated to Unity 2020 LTS

For this course, we've updated the project files to Unity version **2020.3 (LTS)**. This is a long term support version, which is recommended by Unity. LTS versions get released each year and are stable foundation for you to build your projects from.

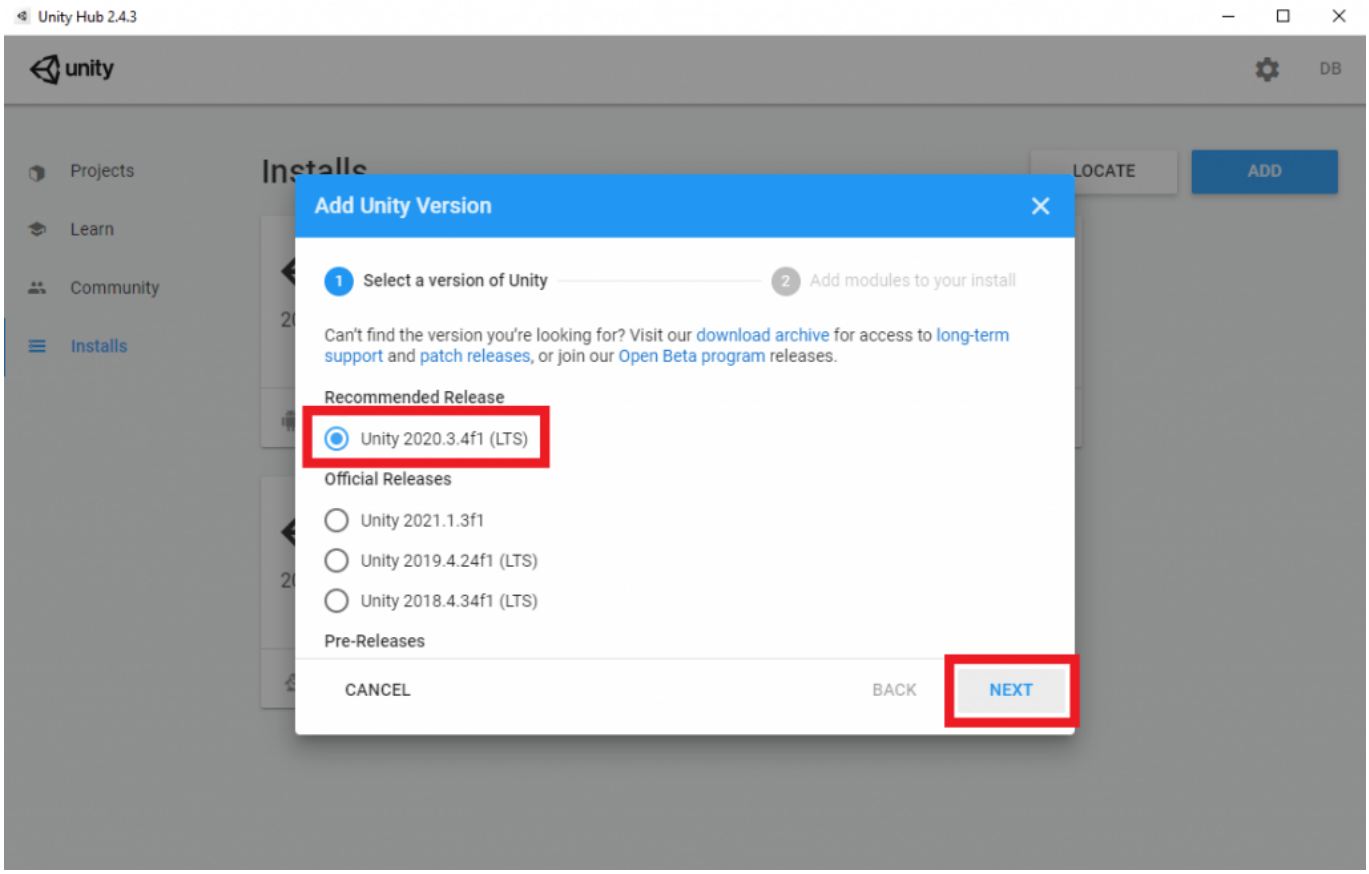
For students, this means consistent project files across all courses – no matter when they were created. No more downloading different Unity versions, or opening old projects filled with errors.

How to Install 2020 LTS

First, open up your Unity Hub and navigate over to the **Installs** screen. Then, click on the blue **Add** button.



This will open up a smaller window with a list of Unity versions. Under the *Recommended Release* header, we want to select **Unity 2020.3 (LTS)**. From here, click **Next** and follow the install process.



Multiplayer in Unity

For this project, we'll be using **Photon** as our networking framework. If you've ever used UNet (which is now deprecated), Photon is very similar.

So how does multiplayer work? Well for us and all other games, multiplayer involves sending data to other computers. We have players who each have their own PC, and a server which many of the server messages go through. For example, if we shoot a gun, that message is sent out to every other player to make us on their screen, shoot our gun.

Master Server

In Photon, a **master server** is a unique server for your game and even different versions of your game. This is a server where all the players and rooms for that game are located. Players can create or join rooms within the master server.

Photon also has cloud based servers around the world. You can pay to increase the capacity or even host it on your own server.

Rooms

Think of a room as a **match** or **lobby**. This is a group of players who can send messages to each other, e.g. syncing values, positions, rotations, animations, etc.

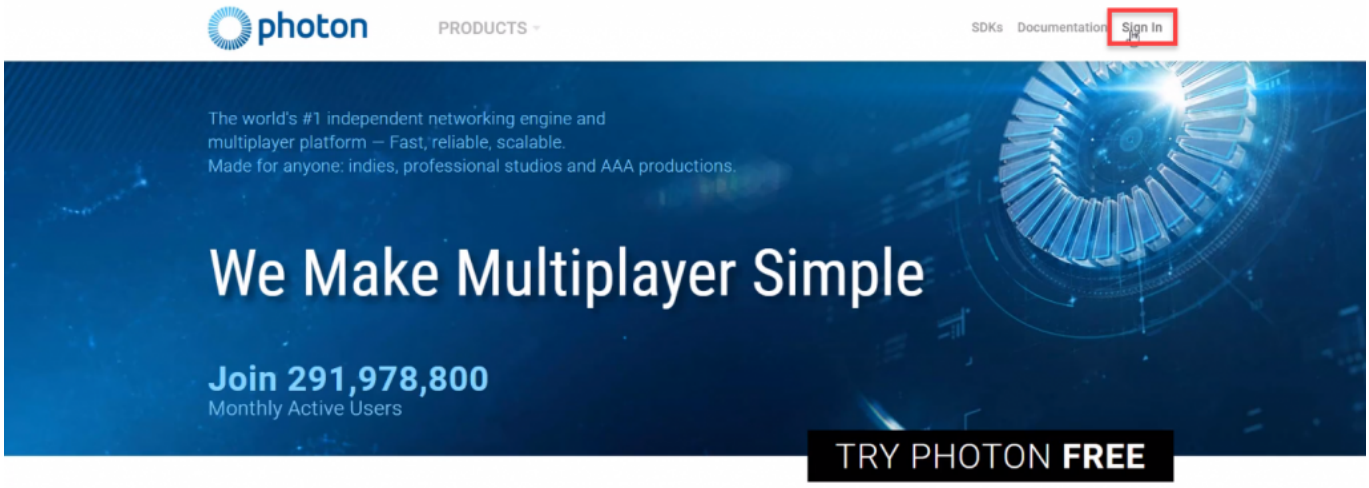
Players

Players can be known as **clients**. Each room also has a **master client** (also known as a host) and by default this is the player who created the room. It's useful for checking and running things server side.

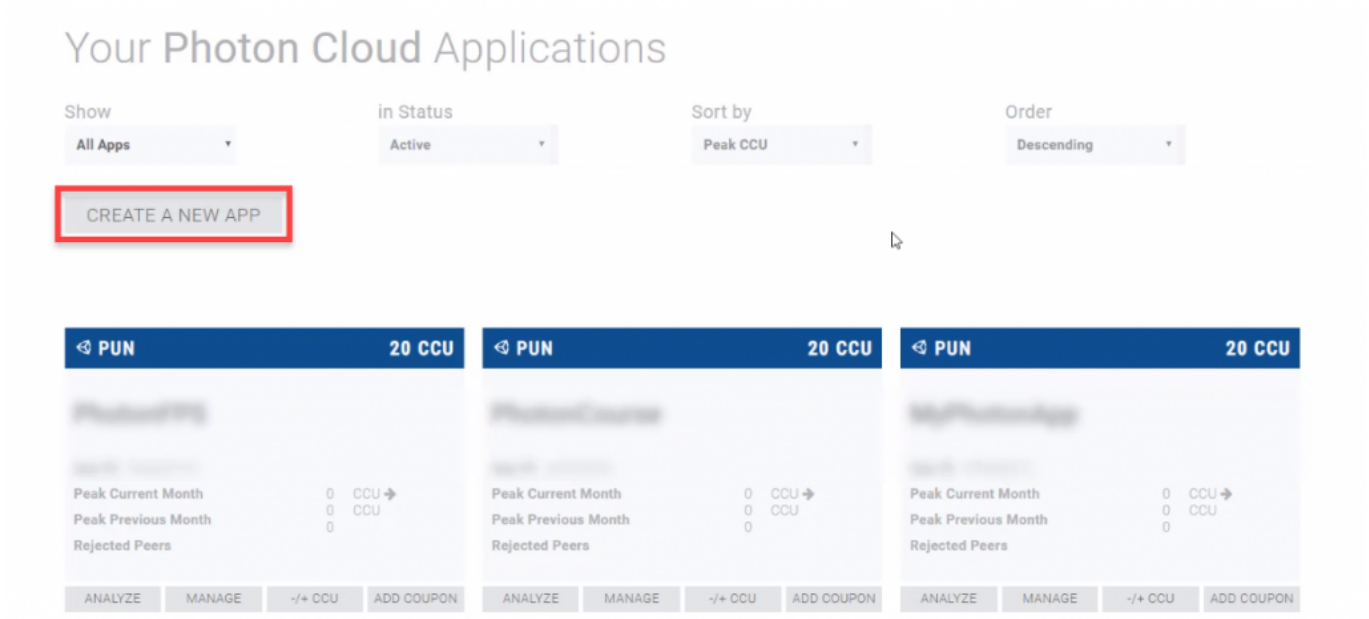
In the next lesson, we'll be setting up Photon.

Creating our Photon App

In order to use Photon, we need to create a new app. Go to http://www.photonengine.com and sign in or create an account.



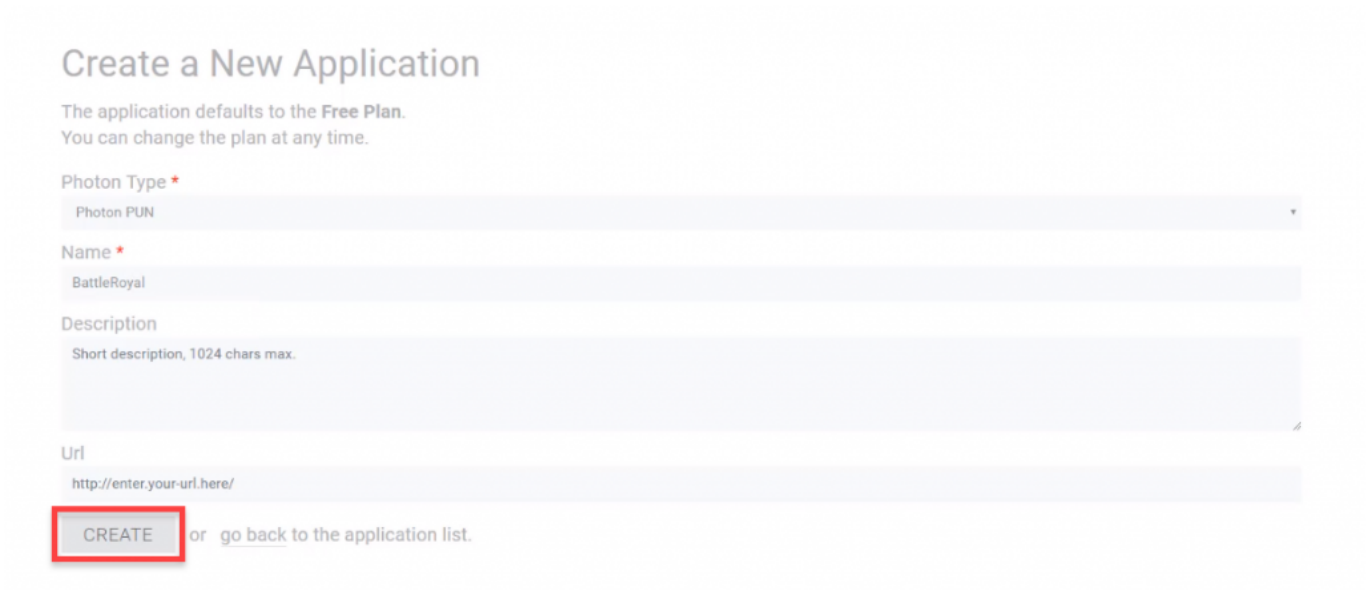
Once you sign in or create an account, you'll be taken to the **Applications** page. Here, we want to create a new app.



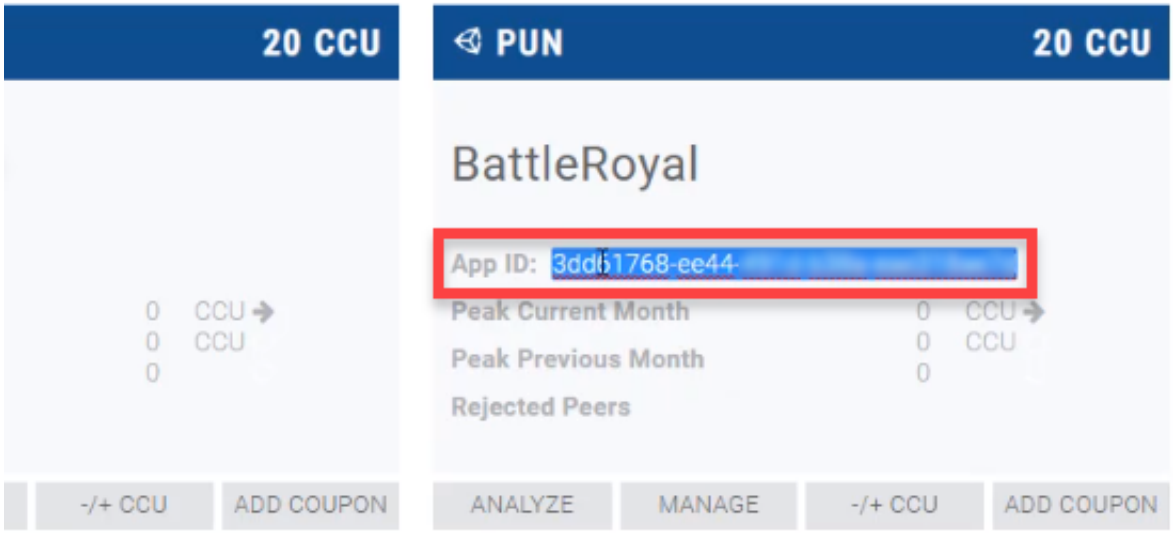
Here, we can fill in the info for our new application.

- Set **Photon Type** to *PUN*
- Enter in a **Name**

When that's done we can click on the **Create** button.

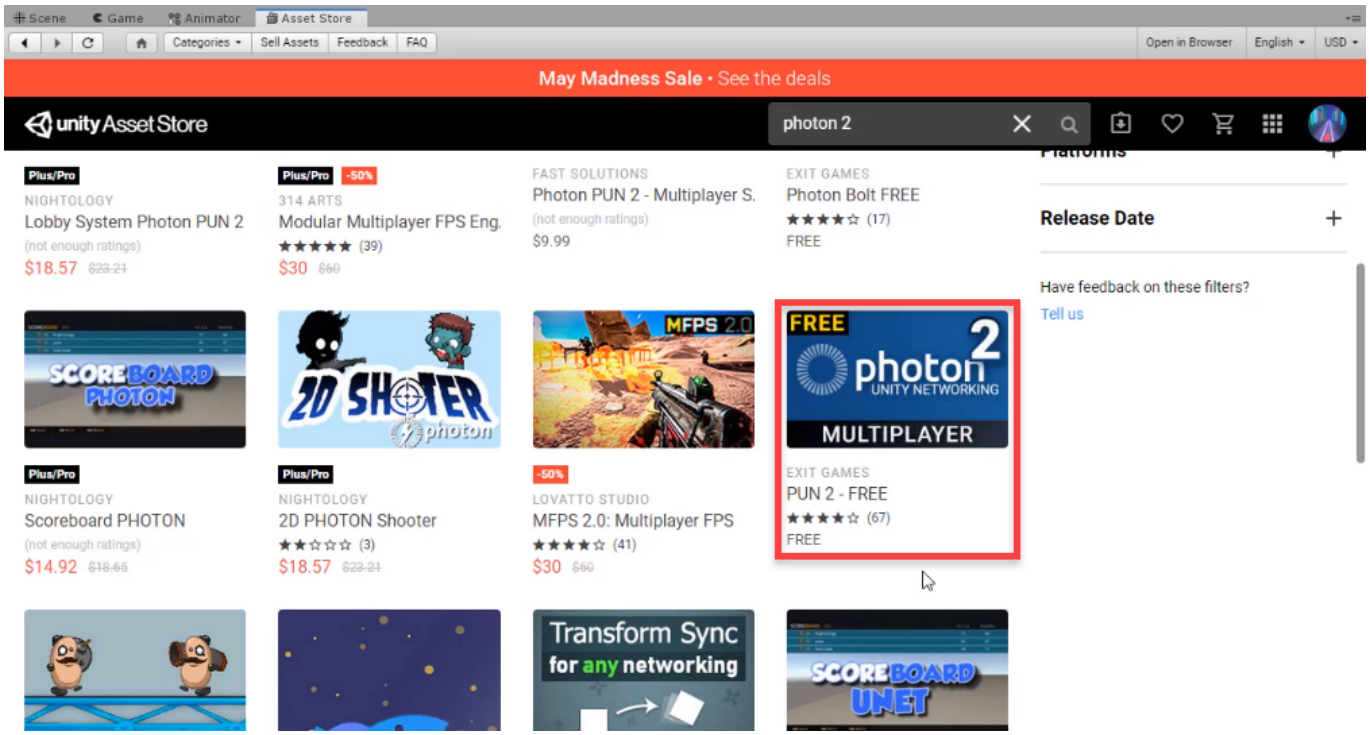


Now you should be taken back to the applications page. Your app should now be in the list. All we need from here is the **App ID**.



Importing Photon

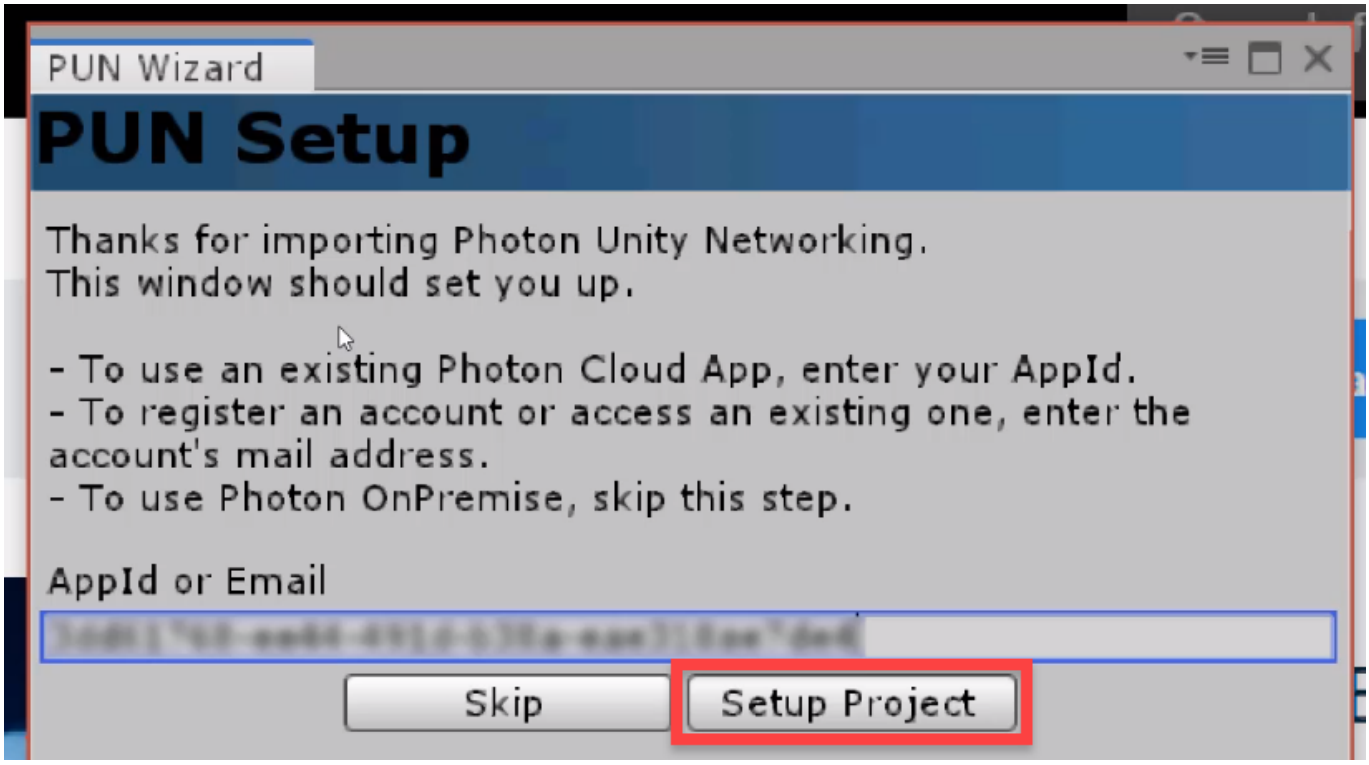
In Unity, we can now import the Photon asset. Open the **Asset Store** window (*Window > Asset Store*) and search for “photon 2”. Import and download the **PUN 2 – FREE** asset.



In the import package window, we want to *not* import some assets. Disable these:

- PhotonChat
- PhotonRealtime > Demos
- PhotonUnityNetworking > Demos

After you import the asset, the **PUN Wizard** window will popup. Here, we just want to paste in our app id, then click on **Setup Project**.



In the next lesson, we'll begin to setup our project files and map.

Folder Structure

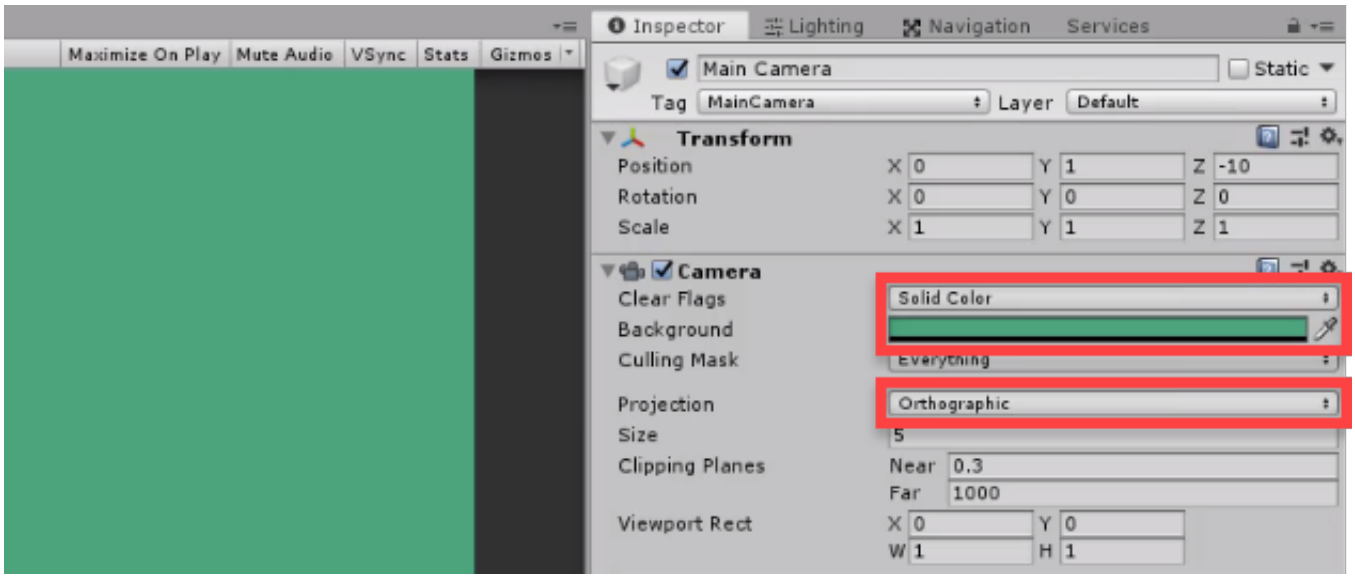
Let's start by setting up out folder structure and 2 scenes.

- Materials
- Models
- Photon
- Prefabs
- Resources
- Scenes
    - *Game scene*
    - *Menu scene*
- Scripts
- Textures

Menu Scene

In our **menu** scene, let's setup the camera for when we do the menu UI in a later lesson.

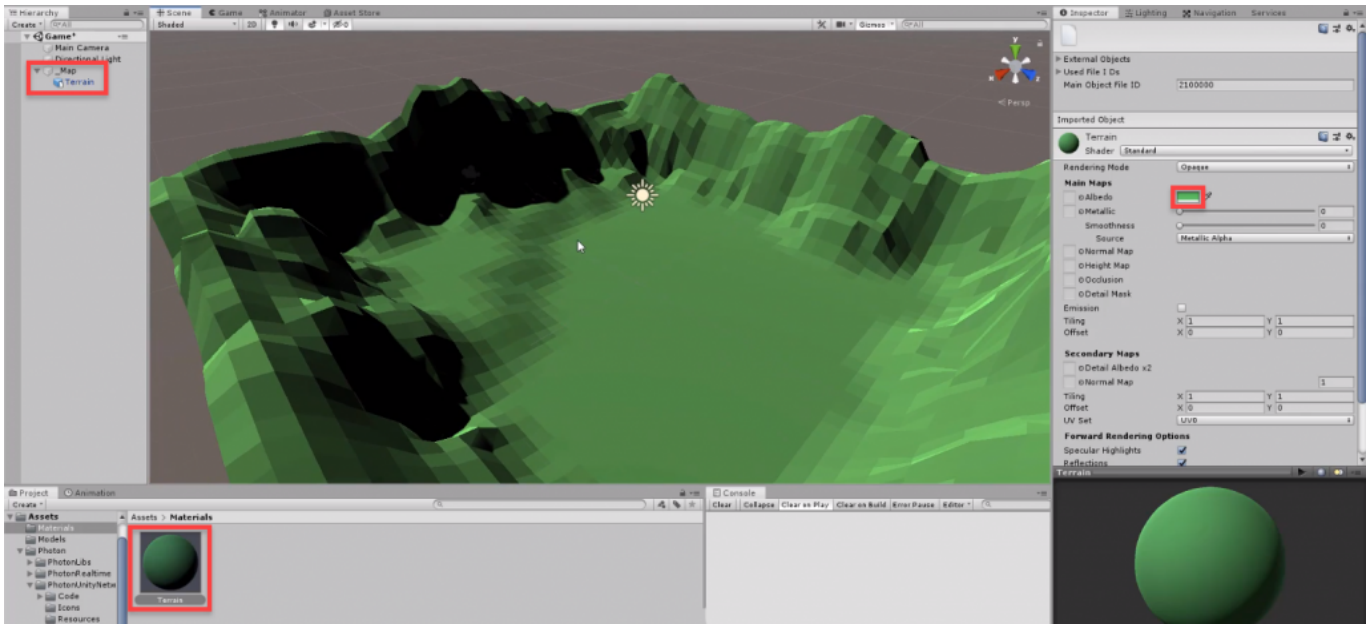
- Set the **Clear Flags** to *Solid Color*
- Set the **Background** to a green-blue color
- Set the **Projection** to *Orthographic*



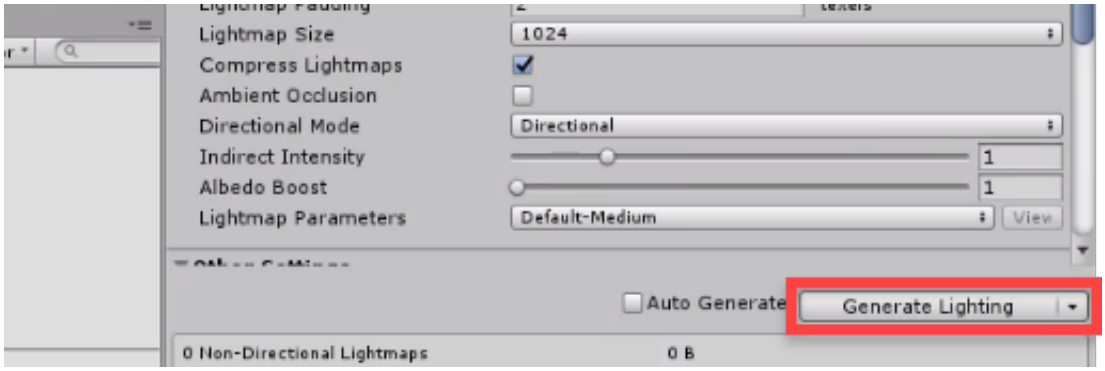
Game Scene

In our **game** scene, let's setup our map. First, we'll drag in the **Terrain** model.

- Create an empty game object called **_Map** and drag the terrain in as a child of that
- Set the **Position** to *0*
- Set the **Scale** to *10*
- Create a new material called **Terrain** and apply it to the terrain model
- Add a **Mesh Collider** component to the terrain



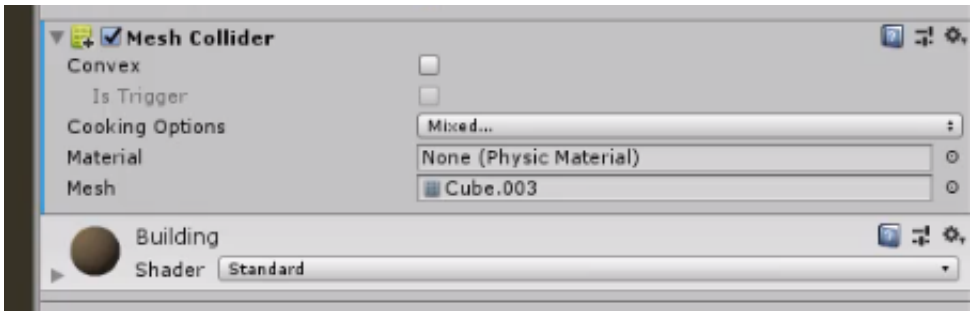
You might notice that the shadows are quite dark. To fix this we can go to the **Lighting** screen (*Window > Rendering > Lighting*) and click on the **Generate Lighting** button at the bottom.



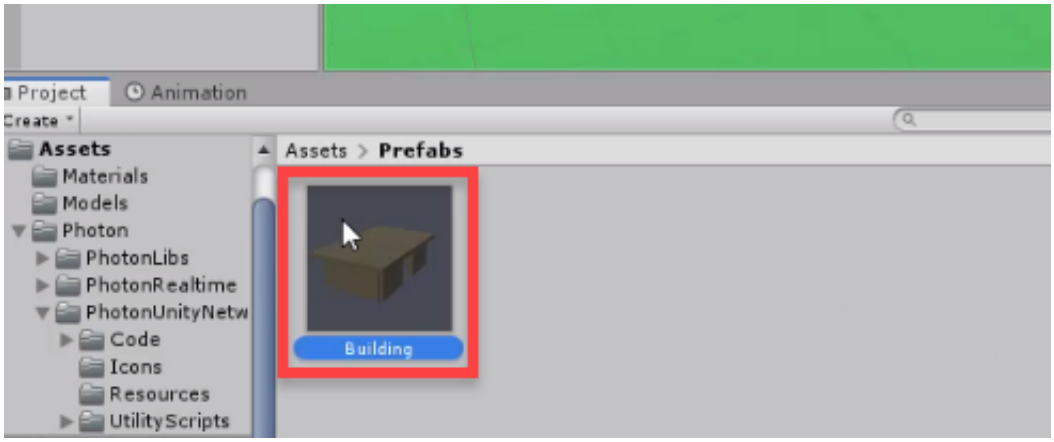
Building our Map

Now we're going to create the various structures and put them around our map. Let's start by dragging in the **Building** model. Like with the terrain, we can create a new material and apply it to the model.

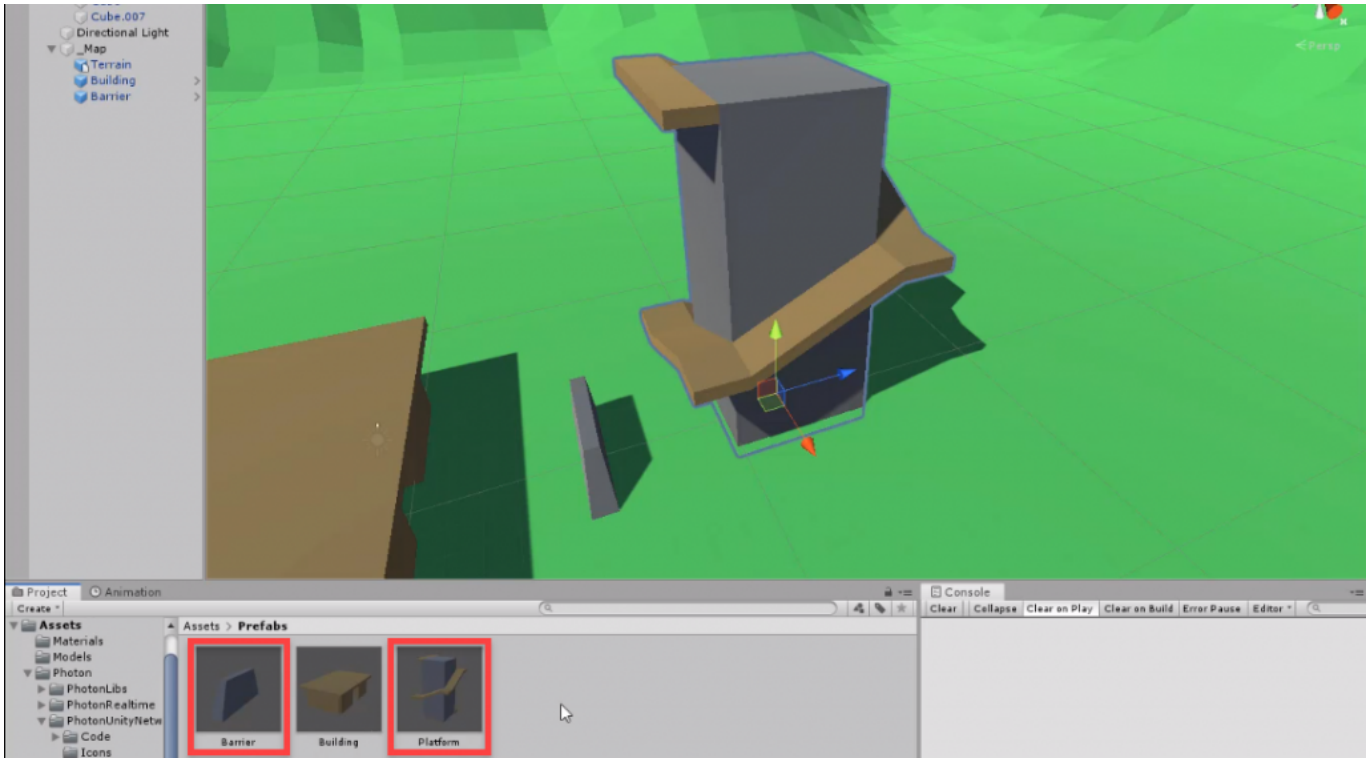
Right now we'd be able to walk through the walls. So what we need to do, is add a **Mesh Collider** component. This will make a custom collider to match the dimensions of the mesh.



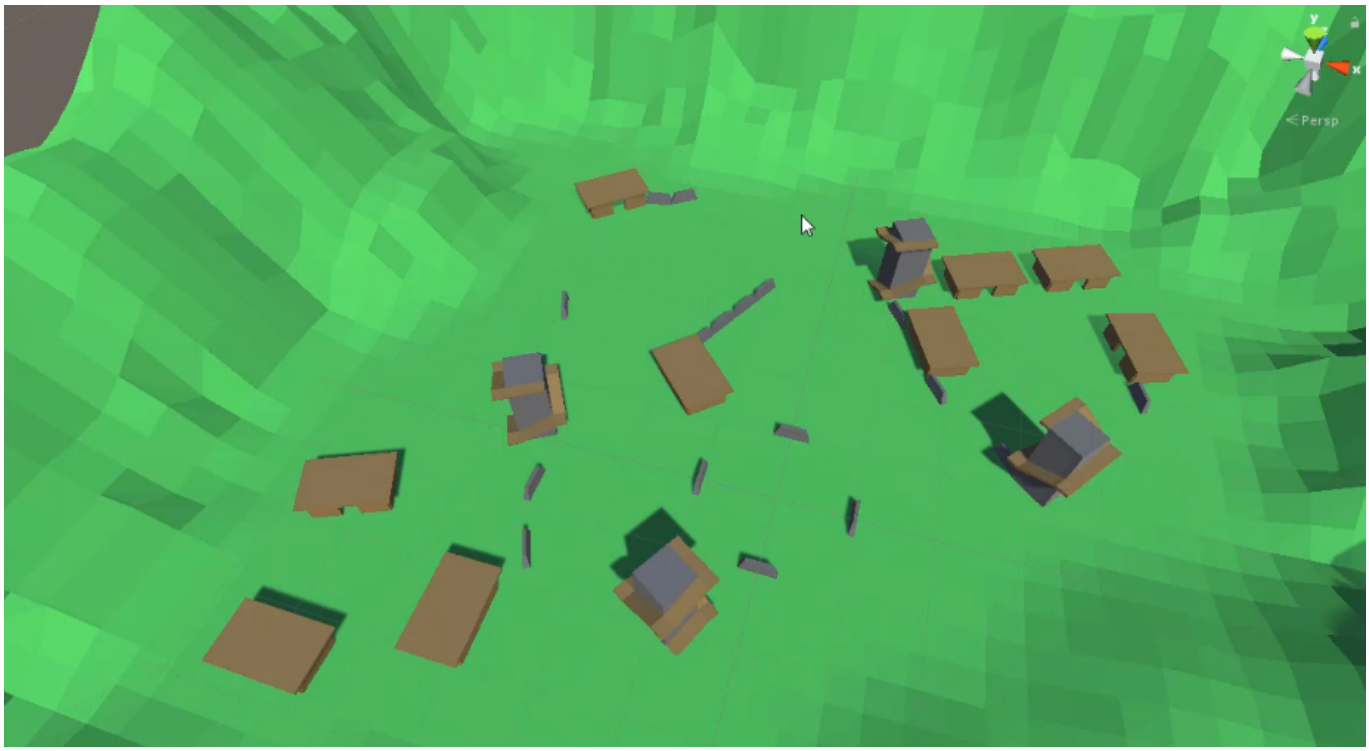
Finally, we can save it in our **Prefabs** folder to both use later, and make sure all changes are applied to all instances.



Do the same for the **Barrier** and **Platform** models.



Finally, we can go around and populate the map with our structures.



In the next lesson, we'll start to create our network manager.

What is the Network Manager?

The network manager will be a custom script that manages connecting to the network.

- Connecting to the master server
- Creating rooms
- Joining rooms
- Changing scenes

NetworkManager Script

In the **menu** scene, create an empty GameObject called **_NetworkManager**. This will hold our script. Then create a new C# script called **NetworkManager** and attach it to the object.

We'll begin by adding in our namespaces we'll be using.

```
using Photon.Pun;
using Photon.Realtime;
```

We're only going to have one variable which will set the max players. We're also going to create an instance. An instance (also known as a singleton) allows us to access the network manager from anywhere in the project without needing to find the object, get the component, etc. We also want to tag the object as **DontDestroyOnLoad**. This means that when we change scenes, the object also carries over.

```
public int maxPlayers = 10;

// instance
public static NetworkManager instance;

void Awake ()
{
    instance = this;
    DontDestroyOnLoad(gameObject);
}
```

So if we press play right now, nothing would happen. In the **Start** function, we need to connect to Photon.

```
void Start ()
{
    // connect to the master server
    PhotonNetwork.ConnectUsingSettings();
}
```

Callback Methods

Right now, we don't know if we've connected to Photon, nothing happens. What we can do, is implement some of Photon's custom callback functions. These are functions that get called when

certain things happen (connect to master server, join room, leave, etc). In order to do this, we need to change our existing **MonoBehaviour** parent class, to Photon's custom **MonoBehaviourPunCallbacks** class.

```
public class NetworkManager : MonoBehaviourPunCallbacks
```

With that done, we can override the **OnConnectedToMaster** function, which gets called when we connect to the master server.

```
public override void OnConnectedToMaster ()
{
    Debug.Log("We've connected to the master server!");
}
```

If we were to press play now, after a few seconds we should see the message in the console.

Creating the Rest of the Functions

Still in our network manager script, we're going to create the rest of our functions. The first one is going to be **CreateRoom**, which gets called when we want to create a room.

```
// attempts to create a room
public void CreateRoom (string roomName)
{
    RoomOptions options = new RoomOptions();
    options.MaxPlayers = (byte)maxPlayers;

    PhotonNetwork.CreateRoom(roomName, options);
}
```

Then we're going to create the **JoinRoom** function, which gets called when we want to join a specific room.

```
// attempts to join a room
public void JoinRoom (string roomName)
{
    PhotonNetwork.JoinRoom(roomName);
}
```

Finally, we have the **ChangeScene** function. This uses Photon's scene loading system which does some behind the scenes stuff with the network.

```
public void ChangeScene (string sceneName)
{
    PhotonNetwork.LoadLevel(sceneName);
}
```

In the next lesson, we'll be working on the camera controller.

CameraController Script

Create a new C# script called **CameraController** and attach it to the main camera. In this lesson, we'll just be setting up the spectator camera functionality. When we create our player controller, we'll do the rest. First up is our variables.

```
[Header("Look Sensitivity")]
public float sensX;
public float sensY;

[Header("Clamping")]
public float minY;
public float maxY;

[Header("Spectator")]
public float spectatorMoveSpeed;

private float rotX;
private float rotY;

private bool isSpectator;
```

The first thing we want to do is lock and hide the mouse cursor in the **Start** function.

```
void Start ()
{
    // lock the cursor to the middle of the screen
    Cursor.lockState = CursorLockMode.Locked;
}
```

For actually checking the input and applying it to our rotation, we'll be doing that inside of the **LateUpdate** function. The reason for this, is that doing it in the **Update** function, can cause jittering.

```
void LateUpdate ()
{

}
```

First up, let's get our mouse inputs.

```
// get the mouse movement inputs
rotX += Input.GetAxis("Mouse X") * sensX;
rotY += Input.GetAxis("Mouse Y") * sensY;
```

Then we can clamp the vertical rotation.


```
// clamp the vertical rotation
rotY = Mathf.Clamp(rotY, minY, maxY);
```

For the spectator camera, we're going to be able to look around as well as moving with the keyboard.

```
// are we spectating?
if(isSpectator)
{
    // rotate the cam vertically
    transform.rotation = Quaternion.Euler(-rotY, rotX, 0);

    // movement
    float x = Input.GetAxis("Horizontal");
    float z = Input.GetAxis("Vertical");
    float y = 0;

    if(Input.GetKey(KeyCode.E))
        y = 1;
    else if(Input.GetKey(KeyCode.Q))
        y = -1;

    Vector3 dir = transform.right * x + transform.up * y + transform.forward * z;
    transform.position += dir * spectatorMoveSpeed * Time.deltaTime;
}
else
{

}
```

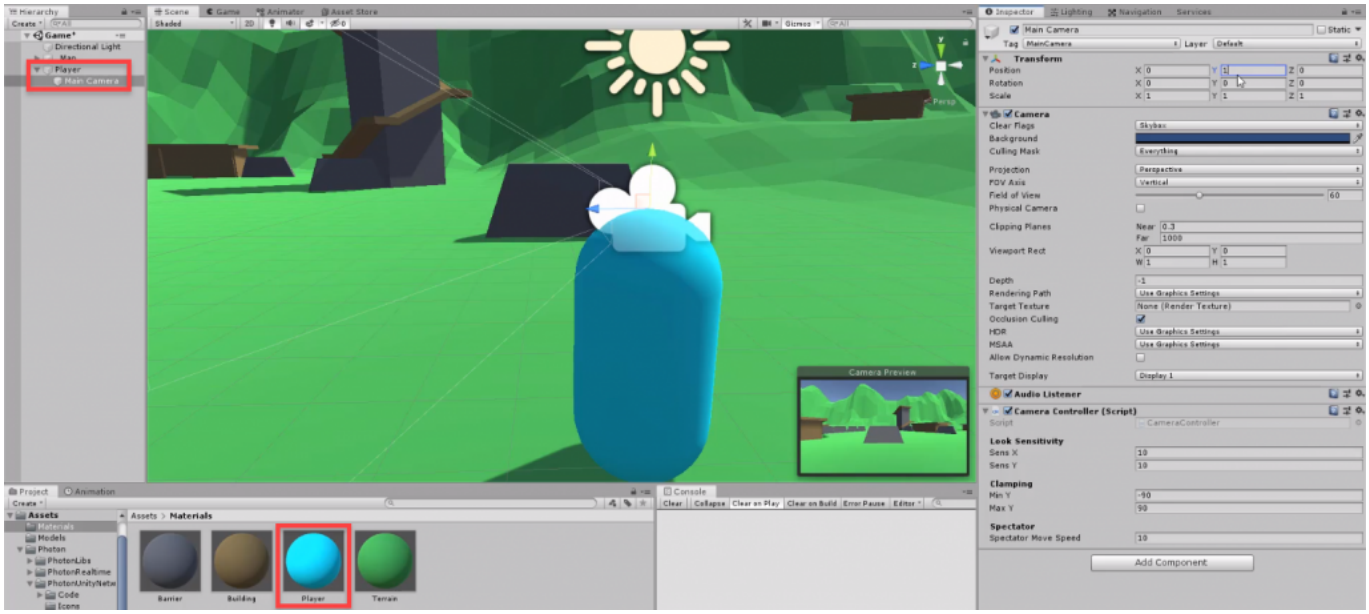
Back in the editor, we can fill in the values.

- **SensX** = *10*
- **SensY** = *10*
- **MinY** = *-90*
- **MaxY** = *90*
- **SpectatorMove peed** = *10*

Player Controller

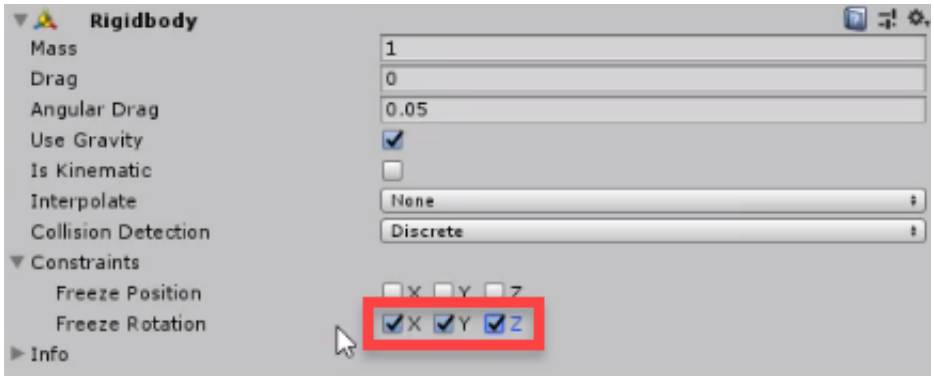
Our player controller will be able to move, jump and shoot. Running over pickups, they'll be able to heal and add ammo. In this lesson, we're going to be setting up the non-networking aspects of the player. This will be the movement and jumping.

Create a new **capsule** and call it **Player**. Then create a new material called **Player** and apply it to the capsule. We can then drag in the main camera as a child of the player and set the Y position to 1.

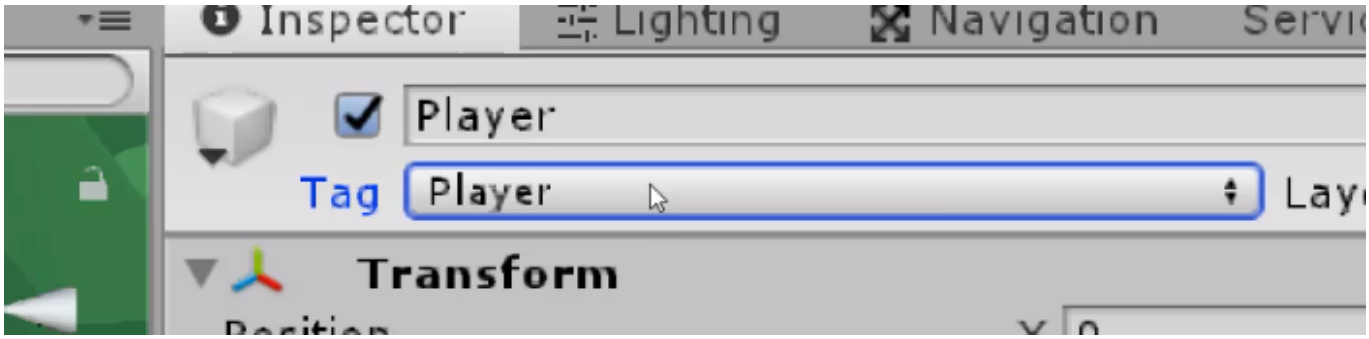


Since we want our player affected by physics, let's add a **Rigidbody** component to the player.

- Freeze all the rotation axis'



Also tag the player as **Player**.



Finishing the Camera Controller Script

Now that we have our player object setup, let's finish the camera controller.

```
...
else
{
    // rotate the camera vertically
    transform.localRotation = Quaternion.Euler(-rotY, 0, 0);

    // rotate the player horizontally
    transform.parent.rotation = Quaternion.Euler(transform.rotation.x, rotX, 0);
}
```

PlayerController Script

Create a new C# script called **PlayerController** and attach it to the player object.

First, let's lay out our variables.

```
[Header("Stats")]
public float moveSpeed;
public float jumpForce;

[Header("Components")]
public Rigidbody rig;
```

Then in the **Update** function, we can call the move and jump functions.

```
void Update ()
{
    Move();

    if(Input.GetKeyDown(KeyCode.Space))
        TryJump();
}
```

The **Move** function, checks for keyboard input and then sets our velocity.


```
void Move ()
{
    // get the input axis
    float x = Input.GetAxis("Horizontal");
    float z = Input.GetAxis("Vertical");

    // calculate a direction relative to where we're facing
    Vector3 dir = (transform.forward * z + transform.right * x) * moveSpeed;
    dir.y = rig.velocity.y;

    // set that as our velocity
    rig.velocity = dir;
}
```

The **TryJump** function, checks to see if the player is standing on the ground and if so, add force upwards.

```
void TryJump ()
{
    // create a ray facing down
    Ray ray = new Ray(transform.position, Vector3.down);

    // shoot the raycast
    if(Physics.Raycast(ray, 1.5f))
        rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
}
```

In the editor, you can fill in the component's properties and test it out.

Remote Procedure Calls

Remove Procedure Calls (known as RPC) allow you to call functions on other player's computers. This is one of the ways we can sync events between all of the players. For example, when we shoot on our computer, we'll send an RPC to all the other players – telling them to spawn a bullet visual. Then when the bullet hits another player, we'll send that player an RPC, damaging them.

You can choose to call an RPC directly to another player, or a group/type of players. These are known as rpc targets and they are:

- **All** – sends the RPC to all players and calls it instantly on your computer
- **AllViaServer** – sends the RPC to all players and you through the server
- **Others** – sends the RPC to all players but you
- **MasterClient** – sends the RPC to the master client (host)
- **AllBuffered** – *All*, but caches the call for new players who join the room
- **AllBufferedViaServer** – *AllViaServer*, but caches the call for new players who join the room
- **OthersBuffered** – *Others*, but caches the call for new players who join the room

PhotonView

A **PhotonView** is a component which identifies an object across the network. Each photon view has a **View ID**, which is unique for each object but the same for that object on all other player's computers. You need a photon view on an object you wish to send an RPC to, spawn across network, destroy across network, synchronize values.

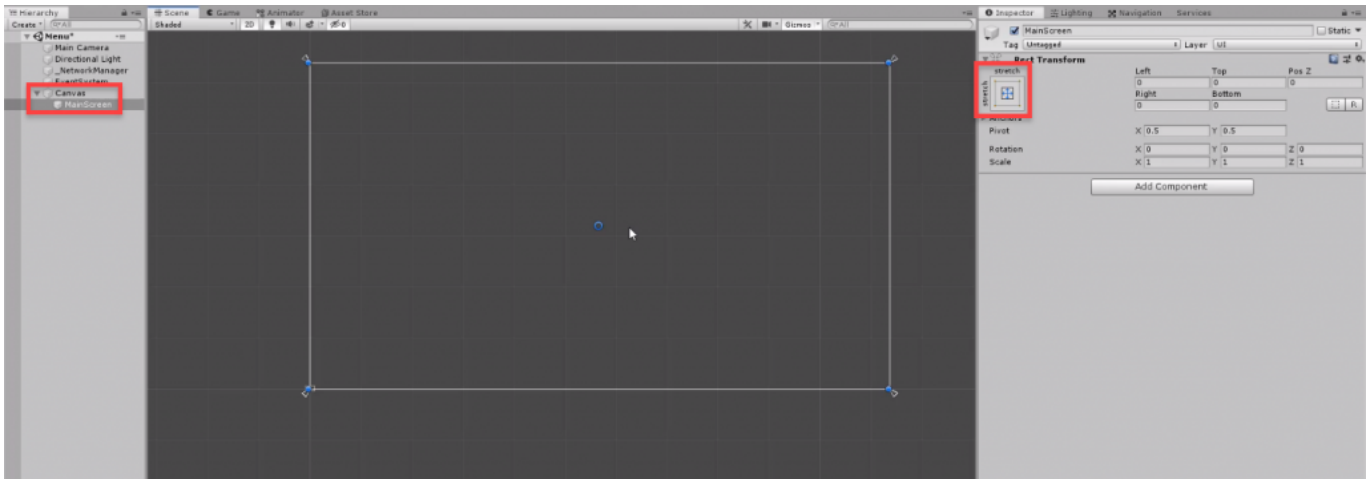
Menu

In this lesson, we're going to create our menu UI. The menu will have four different screens: main screen, create room screen, lobby browser screen and the lobby screen.

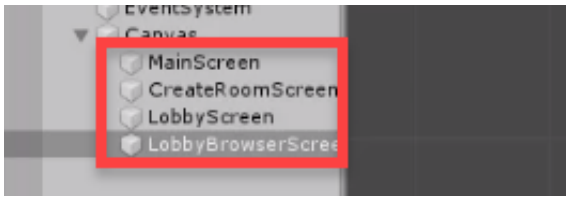
Creating the Menu UI

In the **menu** scene, create a new canvas. Then as a child of the canvas, create an empty object and call it **MainScreen**. We're going to have an empty object for each screen to hold their UI elements.

- Set the **Anchoring** to *stretch-stretch*
- Set the rect to cover the entire canvas like below

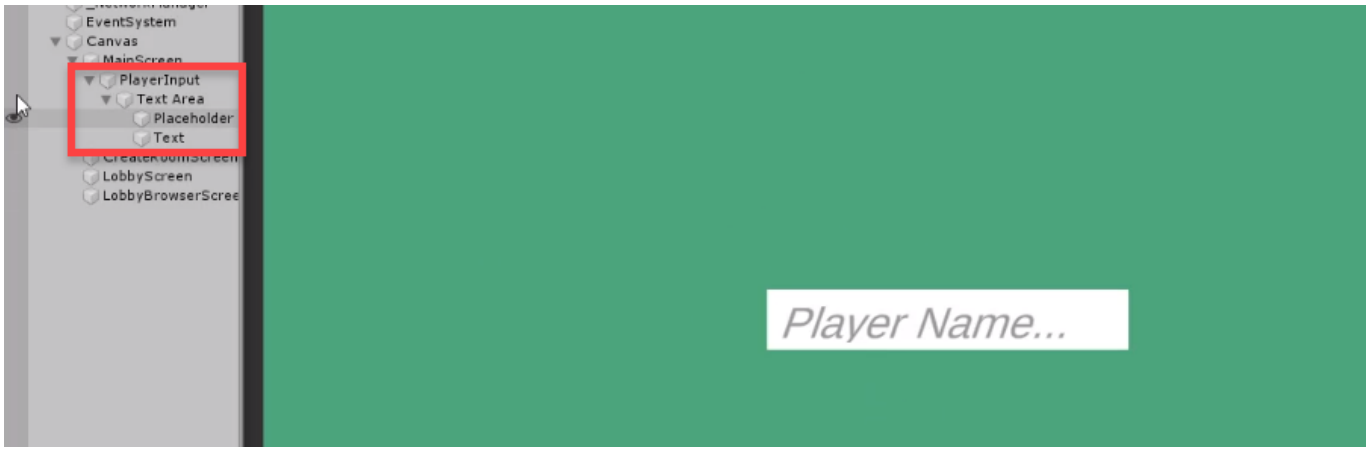


Next, duplicate the object 3 times, for the **CreateRoomScreen**, **LobbyScreen** and **LobbyBrowserScreen**.



Let's now create the player name input field. As a child of the main screen, create a new **Input Field – Text Mesh Pro** and call it **PlayerInput**.

- Set the **Source Image** to *none*
- Set the **Placeholder** text to *Player Name...*
- Set the **Width** to *300*
- Set the **Height** to *50*

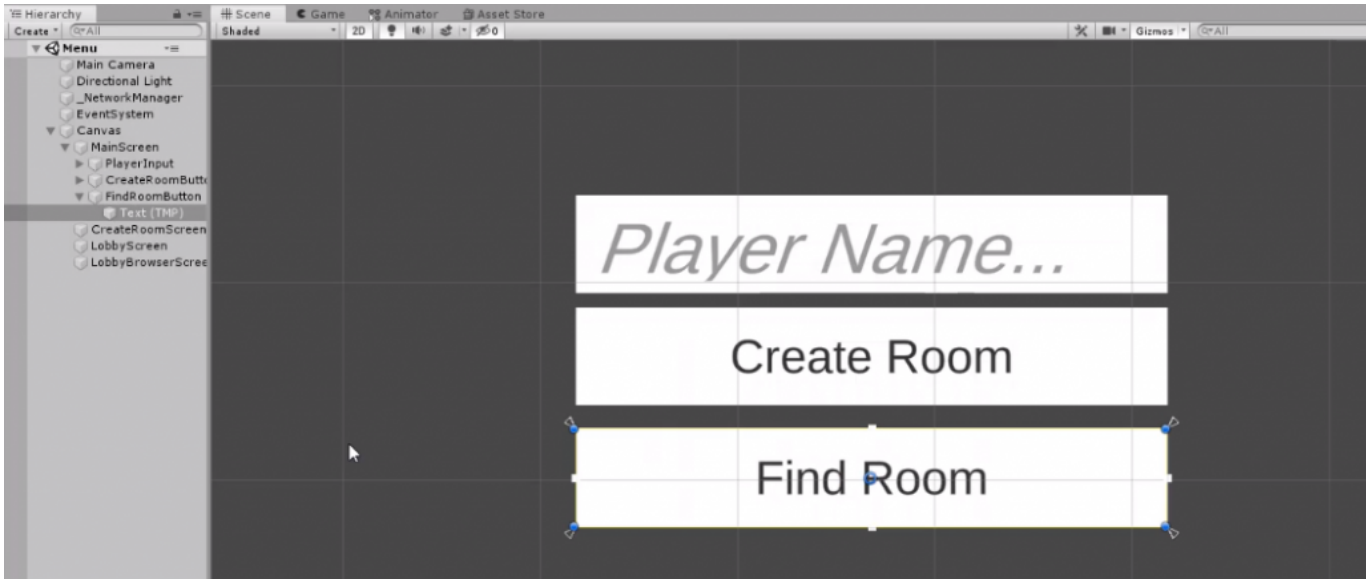


Create a new **Button – Text Mesh Pro** and call it **CreateRoomButton**. Position it underneath the player name input.

- Set the **Source Image** to *none*
- Set the **Text** to *Create Room*
- Set the **Width** to *300*
- Set the **Height** to *50*

Then duplicate the button and call it **FindRoomButton**. Move it underneath the other button.

- Set the **Text** to *Find Room*

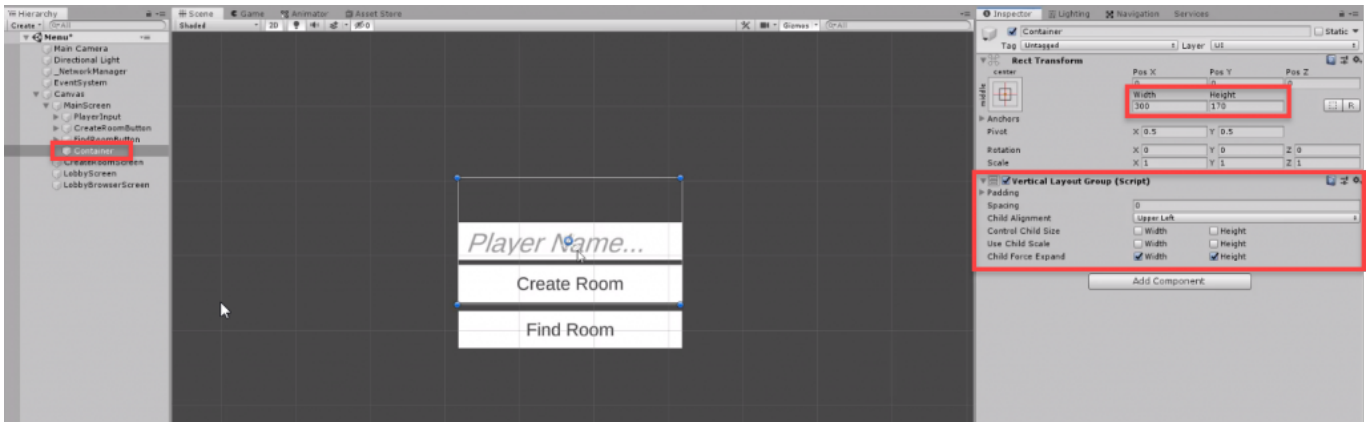


Layout Groups

You might notice that the UI elements don't have even spacing and doing it manually can take some time. To fix this issue, we can use a **layout group**. This is a component which can arrange, position and scale the children UI evenly.

Create a new empty object called **Container** and attach a **Vertical Layout Group** component.

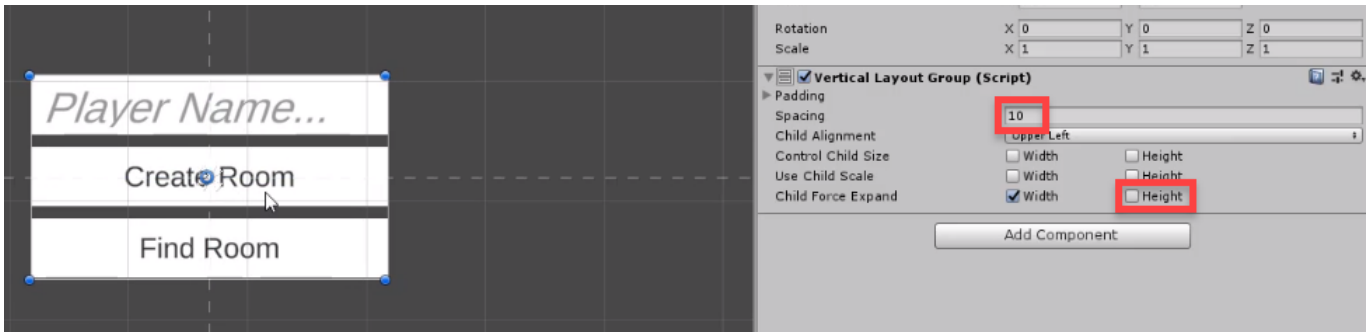
- Set the **Width** to *300*
- Set the **Height** to *170*



You can then drag the three existing elements in as children of the **Container**. You'll see that the elements now have an even spacing between them.

To have it in the way we want though, we'll have to tweak the layout group component first.

- Set the **Spacing** to *10*
- Disable **Child Force Expand – Height**



Create Room Screen

Let's now work on the **Create Room** screen. Disable the main screen so we don't have to see that one. Then for this screen, we can just copy over the **Container** object with our three elements. We'll just reuse it.

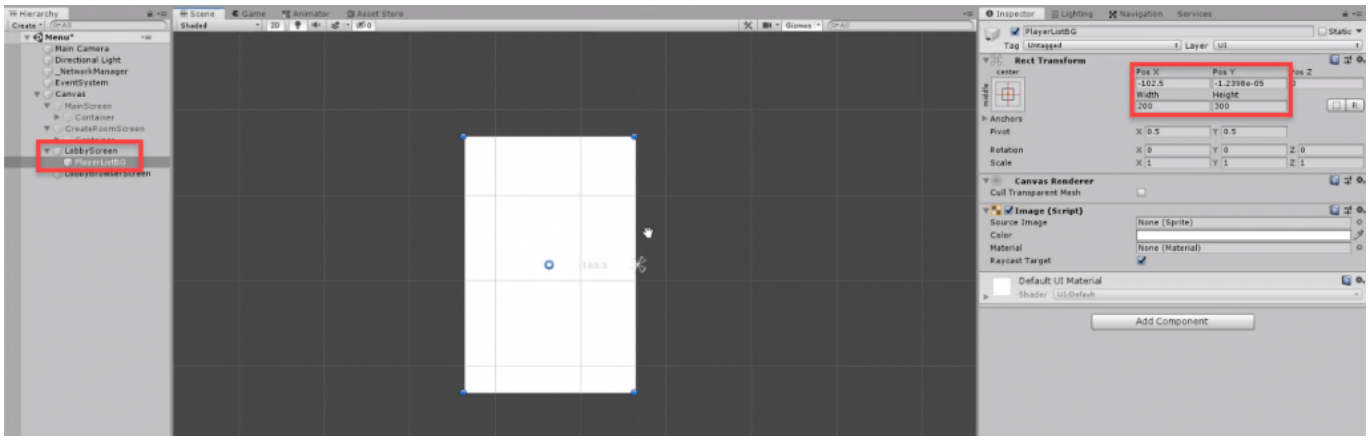
- Change the player name input to **Room Name** input
- Change the create room button to **Create**
- Change the find room button to **Back**

Lobby Screen

Now we can work on the **Lobby** screen. This will display all of the players in the room, display some other info and allow the master client to start the game.

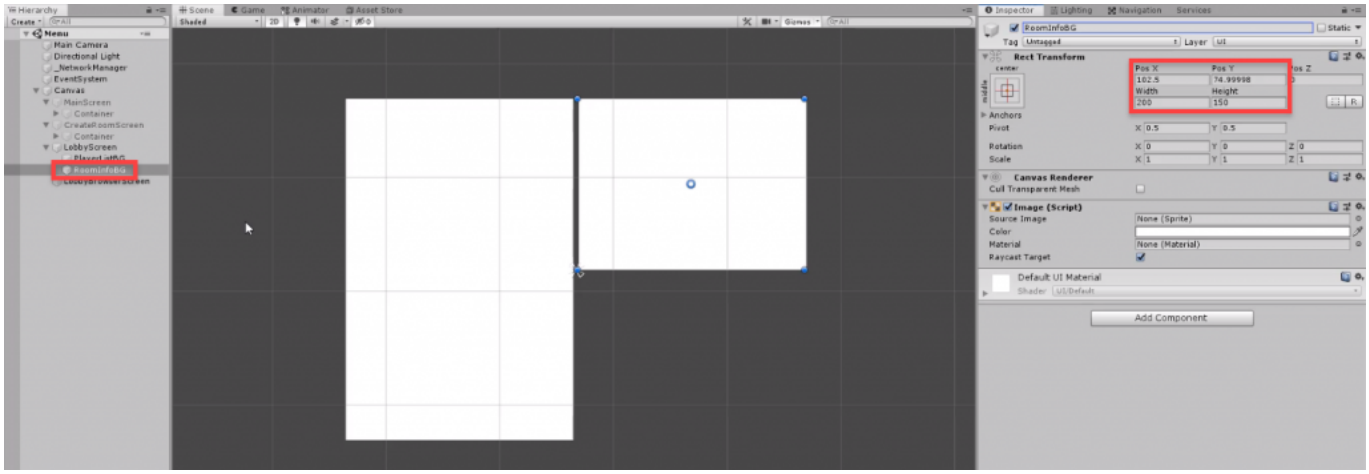
Create a new image object and call it **PlayerListBG**.

- Set the **Position** to *-102.5, 0, 0*
- Set the **Width** to *200*
- Set the **Height** to *300*



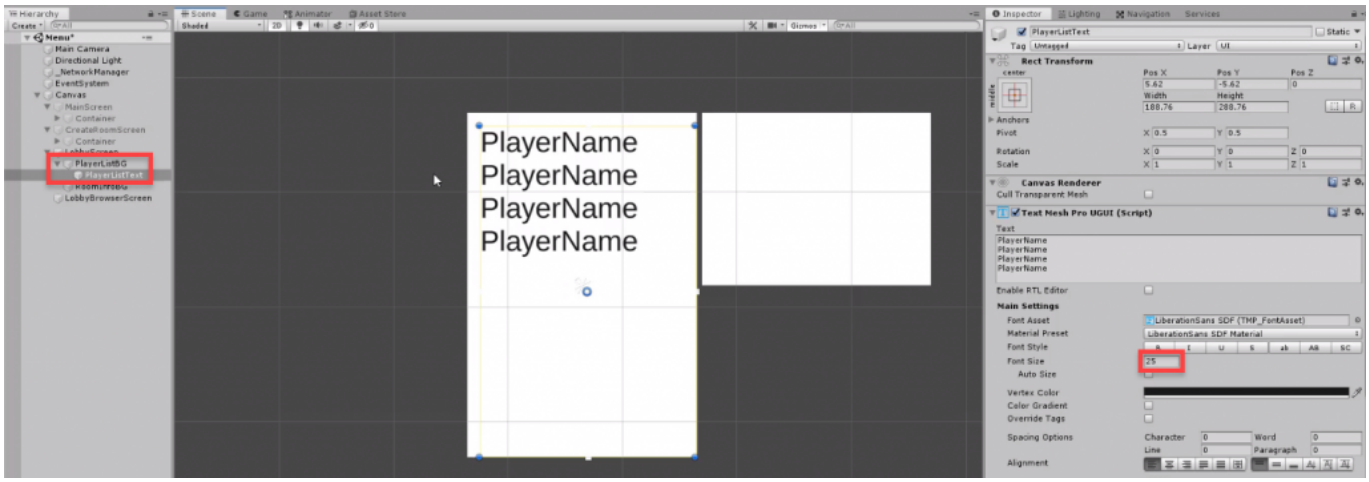
We can then duplicate the image and call it **RoomInfoBG**.

- Set the **Position** to *102.5, 75, 0*
- Set the **Width** to *200*
- Set the **Height** to *150*

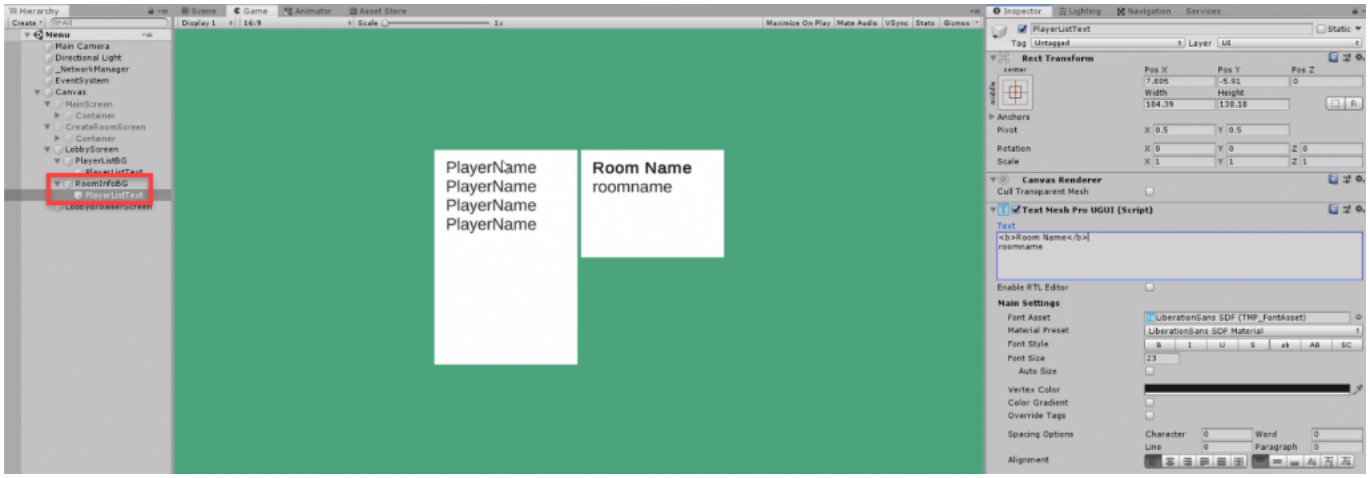


As a child of the player list BG, create a new **Text – Text Mesh Pro** object and call it **PlayerListText**.

- Set the **Font Size** to *25*

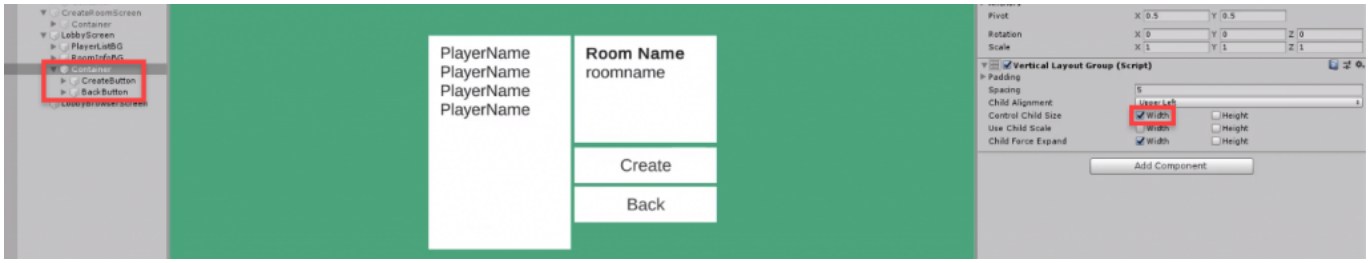


We can then duplicate the text and set it as a child of the room info BG. Call the text **RoomInfoText**.

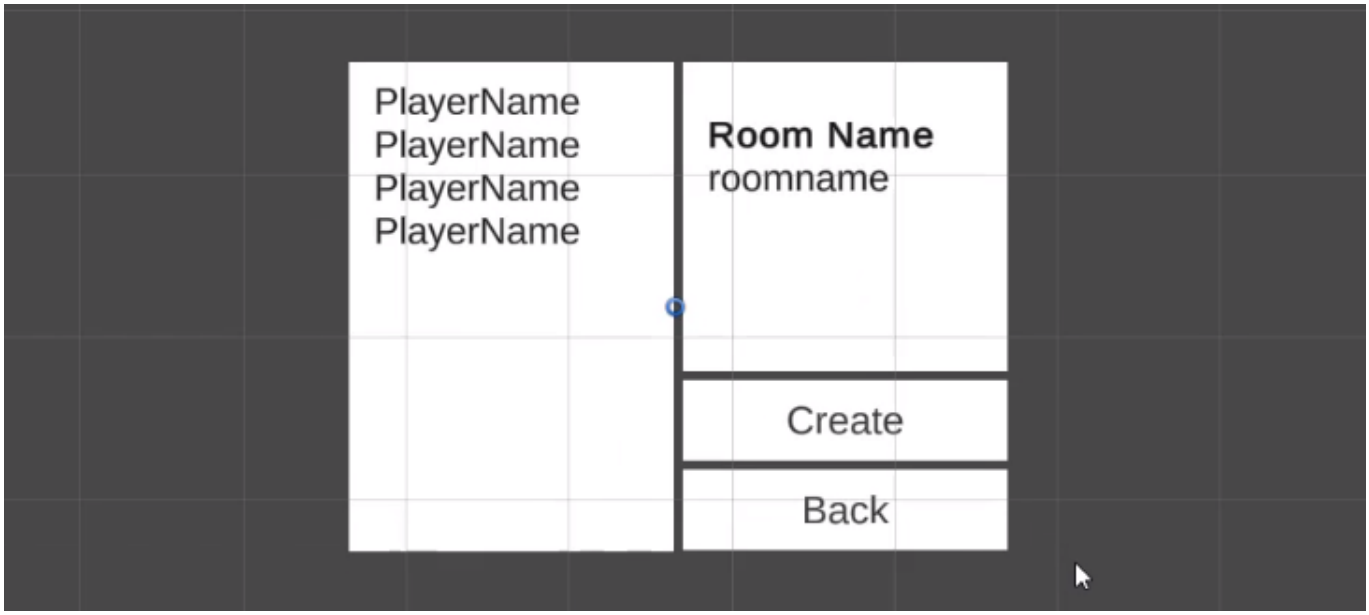


For the buttons, copy over the **Container** from either one of the previous pages.

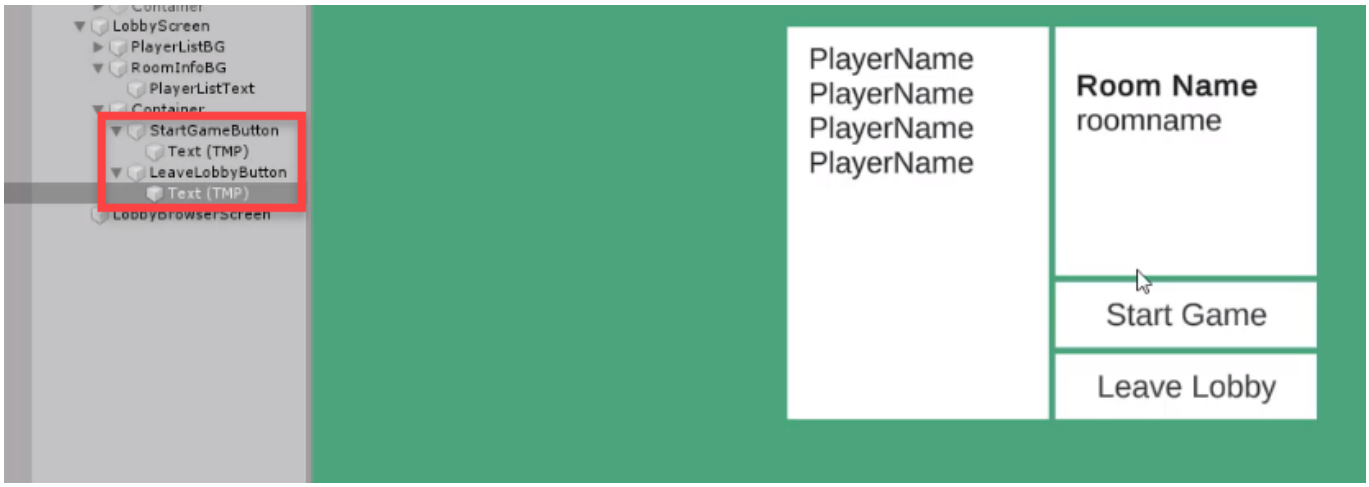
- Delete the **Input** field
- Enable **Control Child Size – Width**



Resize the room info BG to fit the rest of the UI elements like so.



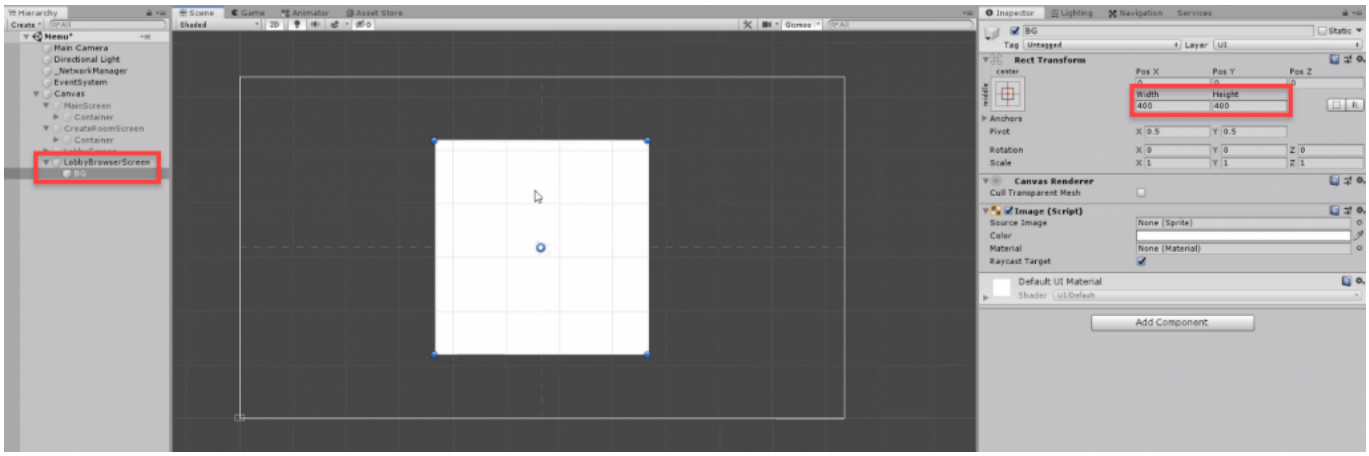
Set the top button to **Start Game** and the bottom button to **Leave Lobby**.



Lobby Browser

Let's now start to create the **lobby browser** screen. Create an image and call it **BG**.

- Set the **Width** and **Height** to *400*

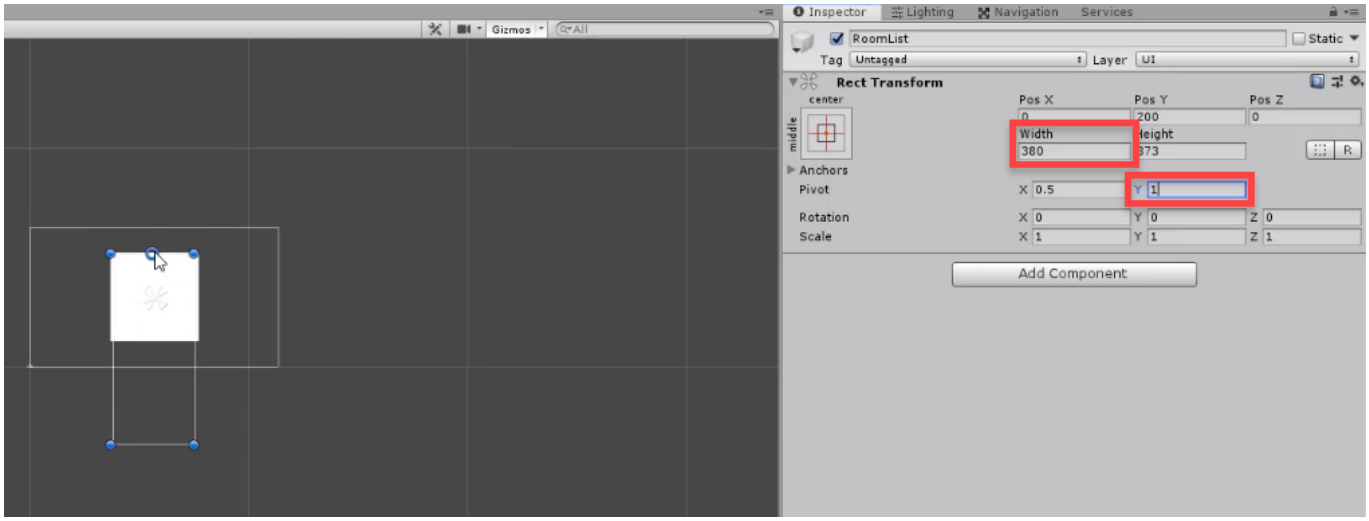


Next create an empty object as a child of BG called **RoomList**. We'll be working on this more next lesson as it's to do with setting up our scroll rect.

Lobby Browser Screen

The container we ended last lesson off with, is to hold all of the room buttons. They will be created in-script and the room list container will be resized to match that. The reason for this, is because this room list will be the contents of the scroll rect. Allowing us to scroll up or down.

To begin, let's set the Y pivot to 1, so that changing the height is easy and uniform for what we want.

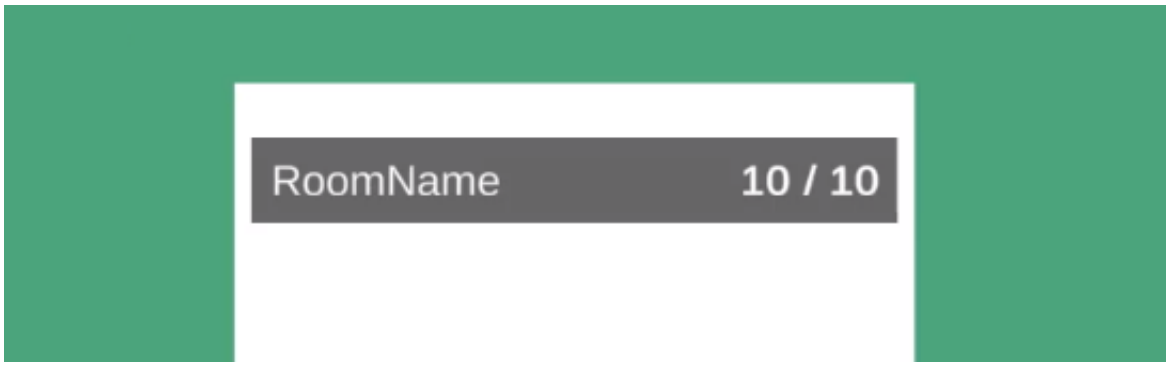


As a child of the **RoomList**, create a new image object and call it **RoomButton**.

- Set the **Width** to *380*
- Set the **Height** to *50*
- Set the **Color** to a dark grey
- Add a **Button** component

Now as a child of the button, we need to create 2 **Text – Text Mesh Pro** objects.

- RoomNameText
- PlayerCountText



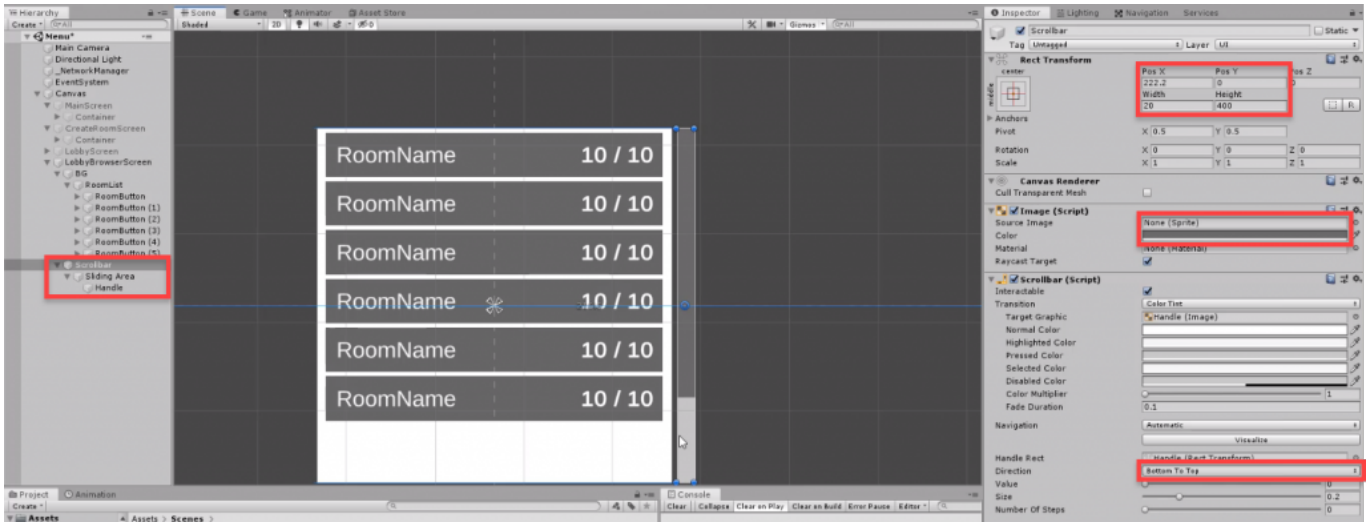
On the room list container, add a **Vertical Layout Group** component.

- Disable **Child Force Expand – Height**
- Set **Spacing** to *5*
- Set **Padding – Top** to *5*

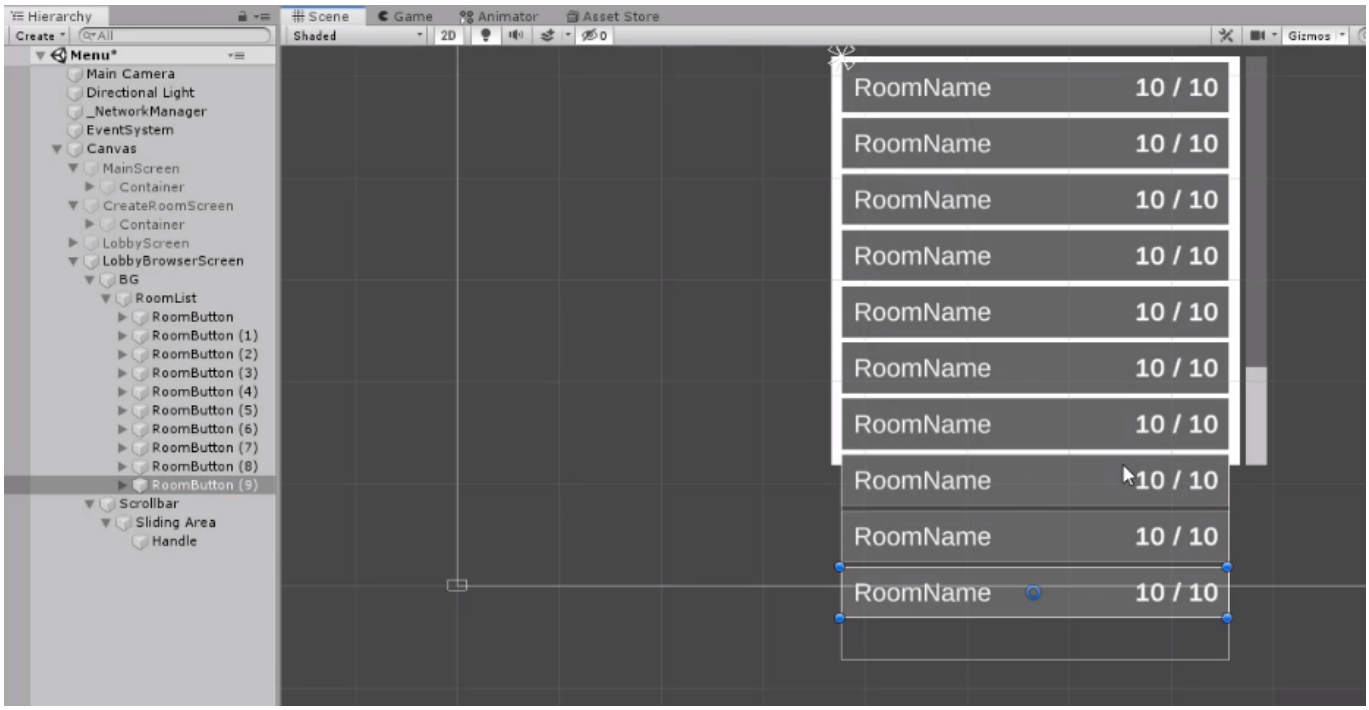
Now we can duplicate the room button and see that they stack up nicely.

Since this is going to be a scroll window, we also need a scroll bar. As a child of the **LobbyBrowserScreen**, create a new UI scrollbar.

- Set the **Width** to *20*
- Set the **Height** to *400*
- Set the **Position** to *222.2, 0, 0*
- Set the **Source Image** to *none* (same for Handle)
- Set the **Color** to a dark grey (light grey for Handle)
- Set the **Direction** to *Bottom To Top*

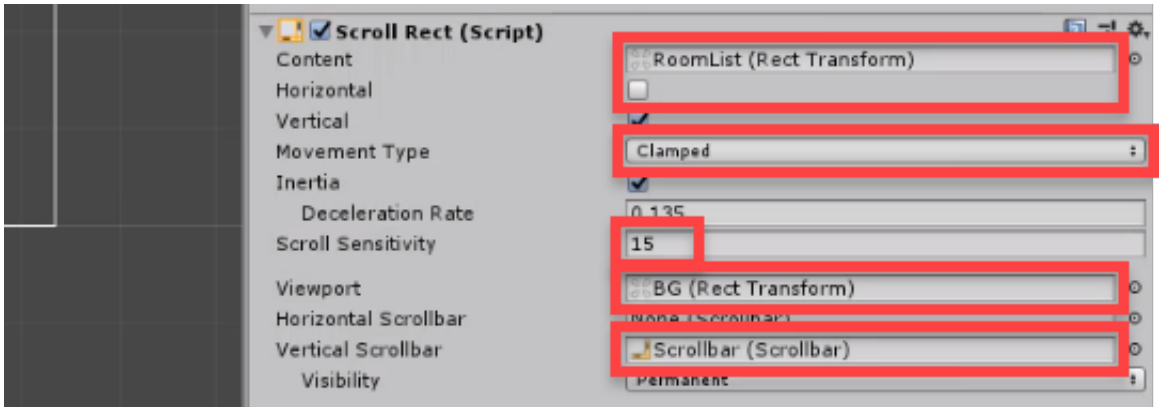


Now let's get the actual scroll window working. First though, we'll duplicate the room button some more. You'll see that they go over the white background and spill downwards. A scroll rect will mask that and allow us to scroll vertically along the room list's height.



Select the **BG** and add the **Scroll Rect** and **Rect Mask 2D** components.

- Set the **Content** to be the **RoomList**
- Disable **Horizontal**
- Set **Movement Type** to *Clamped*
- Set **Scroll Sensitivity** to *15*
- Set the **Viewport** to the **BG**
- Set the **Vertical Scrollbar** to the scrollbar

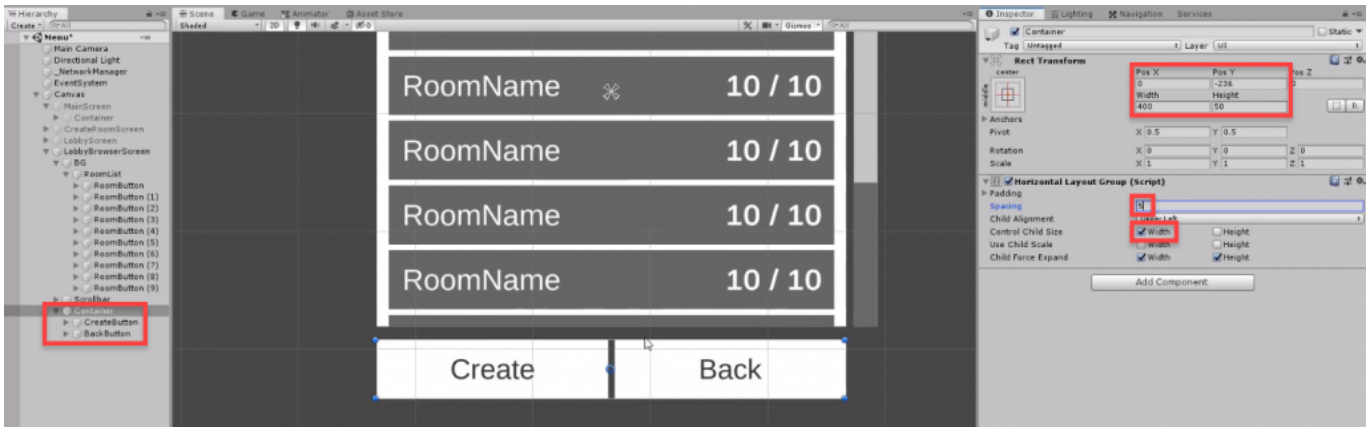


If you press play, you should be able to scroll through the list and see that the BG masks it.

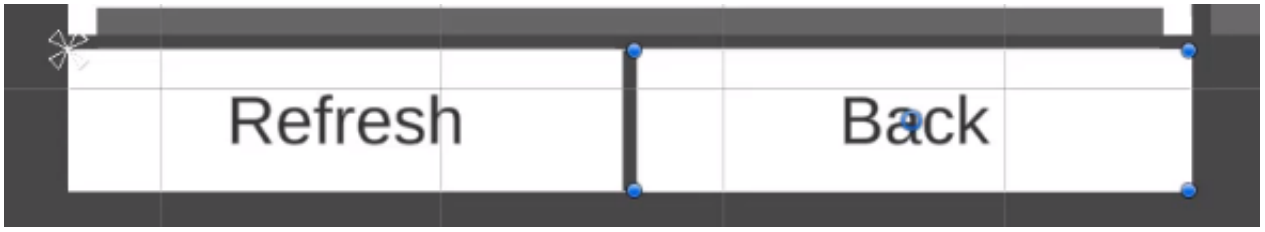
For the buttons, just copy over the previous button's we've used but replace the **Vertical Layout Group** with a **Horizontal Layout Group**.

- Set the **Position** to 0, *-236, 0*
- Set the **Width** to *400*

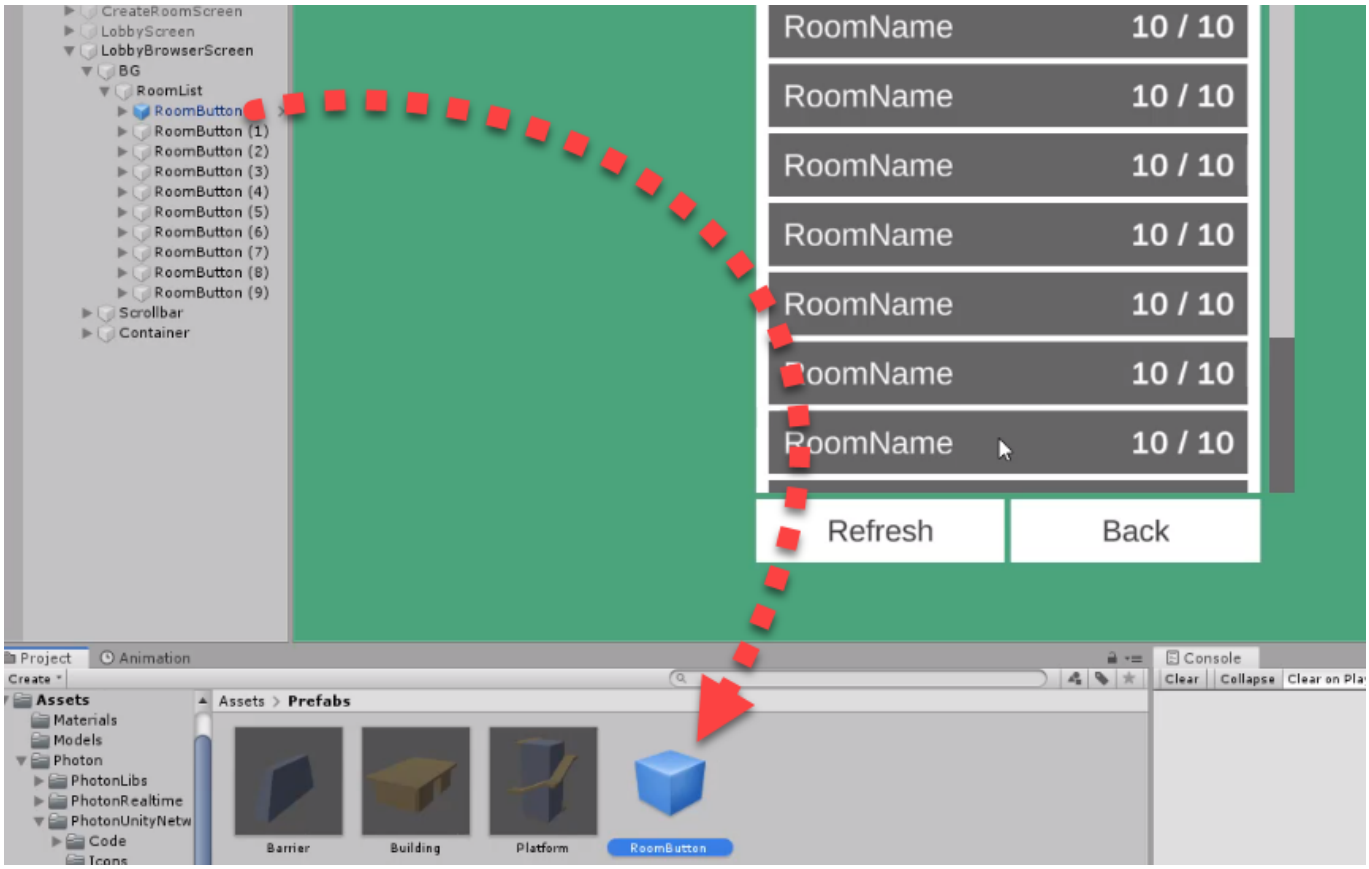
- Set the **Height** to *50*
- Set the **Spacing** to *5*
- Enable **Control Child Size – Width**



Finally, change the buttons to **Refresh** and **Back**.



Since we're going to be spawning the room buttons in-script, let's save it as a prefab. Then remove all of the existing ones in the scene so the RoomList is blank.



Menu Script

Create a new empty GameObject called **_Menu** – this will hold out scripts for the menu. Then create a new c# script and attach it to the object, as well as a **PhotonView** component and set the **View ID** to 2. Add a **PhotonView** to the network manager too and set the **View ID** to 1.

The first thing we need to do is add in our namespaces.

```
using UnityEngine.UI;
using TMPro;
using Photon.Pun;
using Photon.Realtime;
```

Then we need to change the MonoBehaviour inheriting class.

```
public class Menu : MonoBehaviourPunCallbacks, ILobbyCallbacks
```

Next, we can enter in our variables.

```
[Header("Screens")]
public GameObject mainScreen;
public GameObject createRoomScreen;
public GameObject lobbyScreen;
public GameObject lobbyBrowserScreen;

[Header("Main Screen")]
public Button createRoomButton;
public Button findRoomButton;

[Header("Lobby")]
public TextMeshProUGUI playerListText;
public TextMeshProUGUI roomInfoText;
public Button startGameButton;

[Header("Lobby Browser")]
public RectTransform roomListContainer;
public GameObject roomButtonPrefab;

private List<GameObject> roomButtons = new List<GameObject>();
private List<RoomInfo> roomList = new List<RoomInfo>();
```

In the **Start** function, we can do 3 things. First, disable the menu buttons as we don't want the player trying to create/join a room before we've connected to the master server. Then we can show the cursor since we disable it in-game. Finally, if we're still in a game (have we finished a game and returned to the menu?) make the room visible and go to the lobby.

```
void Start ()
{
    // disable the menu buttons at the start
```



```
    createRoomButton.interactable = false;
    findRoomButton.interactable = false;

    // enable the cursor since we hide it when we play the game
    Cursor.lockState = CursorLockMode.None;

    // are we in a game?
    if(PhotonNetwork.InRoom)
    {
        // go to the lobby

        // make the room visible
        PhotonNetwork.CurrentRoom.IsVisible = true;
        PhotonNetwork.CurrentRoom.IsOpen = true;
    }
}
```

Our first function is going to be **SetScreen**. This disables all screens and enables the requested one. This is how we're going to switch between screens.

```
// changes the currently visible screen
void SetScreen (GameObject screen)
{
    // disable all other screens
    mainScreen.SetActive(false);
    createRoomScreen.SetActive(false);
    lobbyScreen.SetActive(false);
    lobbyBrowserScreen.SetActive(false);

    // activate the requested screen
    screen.SetActive(true);
}
```

In the next lesson we'll continue on with the menu script.

Main Screen

Let's now create some functions for the main screen. First, we'll make the **OnPlayerNameValueChanged** function which gets called when we change the player name input field.

```
public void OnPlayerNameValueChanged (TMP_InputField playerNameInput)
{
    PhotonNetwork.NickName = playerNameInput.text;
}
```

When we connect to the master server, let's re-enable the menu buttons so the player can use them.

```
public override void OnConnectedToMaster ()
{
    // enable the menu buttons once we connect to the server
    createRoomButton.interactable = true;
    findRoomButton.interactable = true;
}
```

With our buttons, we can call a function when they're pressed. For the **Create Room** button, we'll call the **OnCreateRoomButton** function.

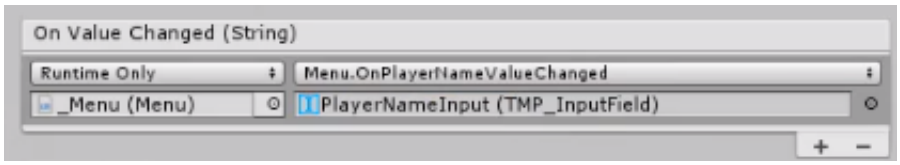
```
// called when the "Create Room" button has been pressed.
public void OnCreateRoomButton ()
{
    SetScreen(createRoomScreen);
}
```

Then for the **Find Room** button.

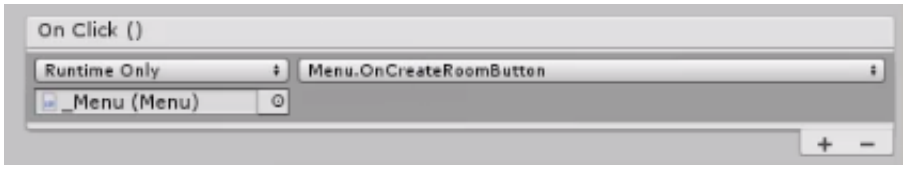
```
// called when the "Find Room" button has been pressed
public void OnFindRoomButton ()
{
    SetScreen(lobbyBrowserScreen);
}
```

Back in the Editor

Back in the Editor, we can start by connecting all the variables to their respective objects. Then select the **PlayerNameInput** input field and scroll down to the **OnValueChanged()** event listener. Drag in the **_Menu** object and select the **Menu.OnPlayerNameValueChanged** function, with the input field as the parameter.



Then for the **Create Room** button, connect that to the corresponding function through the **OnClick()** event. Do the same for the **Find Room** button.



Back Button

Back in the script, we can make a new function for the **Back** button which we have a couple of. Connect that to all the back buttons.

```
// called when the "Back" button gets pressed
public void OnBackButton ()
{
    SetScreen(mainScreen);
}
```

Create Room Screen

The create room screen will allow you to enter in a room name and create that room. Let's create the function for the **Create** button. Connect that to the button object.

```
public void OnCreateButton (TMP_InputField roomNameInput)
{
    NetworkManager.instance.CreateRoom(roomNameInput.text);
}
```


Lobby Screen

Last lesson we created the function for the button to create a room. Now we need a callback function to check for when we join a room, and if so, set the lobby screen.

```
public override void OnJoinedRoom ()
{
    SetScreen(lobbyScreen);
    photonView.RPC("UpdateLobbyUI", RpcTarget.All);
}
```

The **UpdateLobbyUI** function gets called when a player joins or leaves the room. This gets called on all players in the room computers.

```
[PunRPC]
void UpdateLobbyUI ()
{
    // enable or disable the start game button depending on if we're the host
    startGameButton.interactable = PhotonNetwork.IsMasterClient;

    // display all the players
    playerListText.text = "";

    foreach(Player player in PhotonNetwork.PlayerList)
        playerListText.text += player.NickName + "\n";

    // set the room info text
    roomInfoText.text = "<b>Room Name</b>\n" + PhotonNetwork.CurrentRoom.Name;
}
```

When the player leaves a lobby, we need to update the lobby UI for all the players so the player who left, is no longer there.

```
public override void OnPlayerLeftRoom (Player otherPlayer)
{
    UpdateLobbyUI();
}
```

Now we can make functions for the two buttons on the lobby screen. Connect those up to their respective buttons.

```
public void OnStartGameButton ()
{
    // hide the room
    PhotonNetwork.CurrentRoom.IsOpen = false;
    PhotonNetwork.CurrentRoom.IsVisible = false;

    // tell everyone to load the game scene
    NetworkManager.instance.photonView.RPC("ChangeScene", RpcTarget.All, "Game");
```

```
}



public void OnLeaveLobbyButton ()
{
    PhotonNetwork.LeaveRoom();
    SetScreen(mainScreen);
}
```


Lobby Browser Screen

Let's now script the lobby browser screen. The first function is **UpdateLobbyBrowserUI**, which updates the list of currently join-able rooms.

```
void UpdateLobbyBrowserUI ()
{
    // disable all room buttons
    foreach(GameObject button in roomButtons)
        button.SetActive(false);

    // display all current rooms in the master server
    for(int x = 0; x < roomList.Count; ++x)
    {
        // get or create the button object
        GameObject button = x >= roomButtons.Count ? CreateRoomButton() : roomButtons
[x];

        button.SetActive(true);

        // set the room name and player count texts
        button.transform.Find("RoomNameText").GetComponent<TextMeshProUGUI>().text =
roomList[x].Name;
        button.transform.Find("PlayerCountText").GetComponent<TextMeshProUGUI>().text
 = roomList[x].PlayerCount + " / " + roomList[x].MaxPlayers;

        // set the button OnClick event
        Button buttonComp = button.GetComponent<Button>();
        string roomName = roomList[x].Name;
        buttonComp.onClick.RemoveAllListeners();
        buttonComp.onClick.AddListener(() => { OnJoinRoomButton(roomName); });
    }
}



GameObject CreateRoomButton ()
{
    GameObject buttonObj = Instantiate(roomButtonPrefab, roomListContainer.transform)
;
    roomButtons.Add(buttonObj);

    return buttonObj;
}
```

The **OnJoinRoomButton** gets called when we click on a room button.

```
public void OnJoinRoomButton (string roomName)
{
    NetworkManager.instance.JoinRoom(roomName);
}
```

The **OnRefreshButton** gets called when we click on the **Refresh** button.

```
public void OnRefreshButton ()
{
    UpdateLobbyBrowserUI();
}
```

OnRoomListUpdate is an override function that gets called when a player creates/joins/leaves a room. This sends over a list of all the rooms.

```
public override void OnRoomListUpdate (List<RoomInfo> allRooms)
{
    roomList = allRooms;
}
```

Back up in the **SetScreen** function, we need to check if we're setting the lobby browser screen. If so, update the lobby browser UI.

```
if(screen == lobbyBrowserScreen)
    UpdateLobbyBrowserUI();
```


Game Manager

Create a new C# script called **GameManager** and attach it to a new game object called **_GameManager**. Also add a **PhotonView** component, with a **View ID** of 2.

What we want to do now, is drag the player object down into the **Resources** folder.

Let's also add a **PhotonView** component to the player.

First, let's add our namespaces.

```
using Photon.Pun;
using Photon.Realtime;
using System.Linq;
```

Then we need to change our inheriting class in order to access some photon components.

```
public class GameManager : MonoBehaviourPun
```

Next, we can fill in our variables.

```
[Header("Players")]
public string playerPrefabLocation;
public PlayerController[] players;
public Transform[] spawnPoints;
public int alivePlayers;

private int playersInGame;

// instance
public static GameManager instance;

void Awake ()
{
    instance = this;
}
```


In the **Start** function, we want to set the size of the players array and alive players integer.

```
void Start ()
{
    players = new PlayerController[PhotonNetwork.PlayerList.Length];
    alivePlayers = players.Length;

    photonView.RPC("ImInGame", RpcTarget.AllBuffered);
}
```

ImInGame, is a function which gets called to all players when someone loads into the Game scene. This keeps track of how many players are in the game and if everyone is in, spawn in the players.

```
[PunRPC]
void ImInGame ()
{
    playersInGame++;

    if(PhotonNetwork.IsMasterClient && playersInGame == PhotonNetwork.PlayerList.Leng
th)
        photonView.RPC("SpawnPlayer", RpcTarget.All);
}
```

The **SpawnPlayer** instantiates a player across the network.

```
[PunRPC]
void SpawnPlayer ()
{
    GameObject playerObj = PhotonNetwork.Instantiate(playerPrefabLocation, spawnPoint
s[Random.Range(0, spawnPoints.Length)].position, Quaternion.identity);

    // initialize the player for all other players

}
```


Initializing the Player

When we spawn in a player, we need to initialize them. So in the **PlayerController** script, let's create the **Initialize** function. Also adding in the **Photon.Pun** and **Photon.Realtime** namespaces, along with the **MonoBehaviourPun** inheriting class.

```
[PunRPC]
public void Initialize (Player player)
{
    id = player.ActorNumber;
    photonPlayer = player;

    GameManager.instance.players[id - 1] = this;

    // is this not our local player?
    if(!photonView.IsMine)
    {
        GetComponentInChildren<Camera>().gameObject.SetActive(false);
        rig.isKinematic = true;
    }
}
```

Let's also add some new variables.

```
public int id;
public Player photonPlayer;
```

On the player prefab object, we want to add a **Photon Transform View** component. This will sync the position of this object across the network. Make sure to drag the transform view into the photon view's observed components list in order for it to work.

Game Manager

Back in the **GameManager** script, we can finish off the **SpawnPlayer** function.

```
playerObj.GetComponent<PlayerController>().photonView.RPC("Initialize", RpcTarget.All
, PhotonNetwork.LocalPlayer);
```

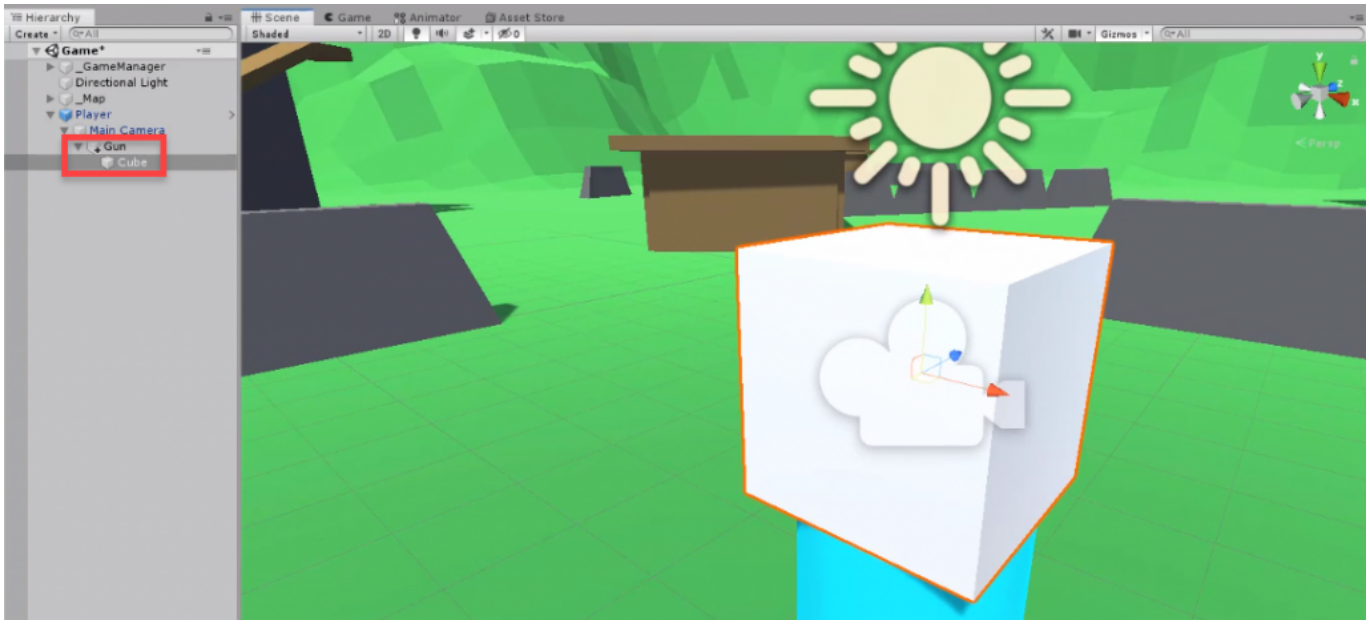
In the editor, set the player prefab location to **Player** and create some empty objects for spawn points, dragging them into the array.

We can now delete the player from the game scene and try run the game from the menu scene.

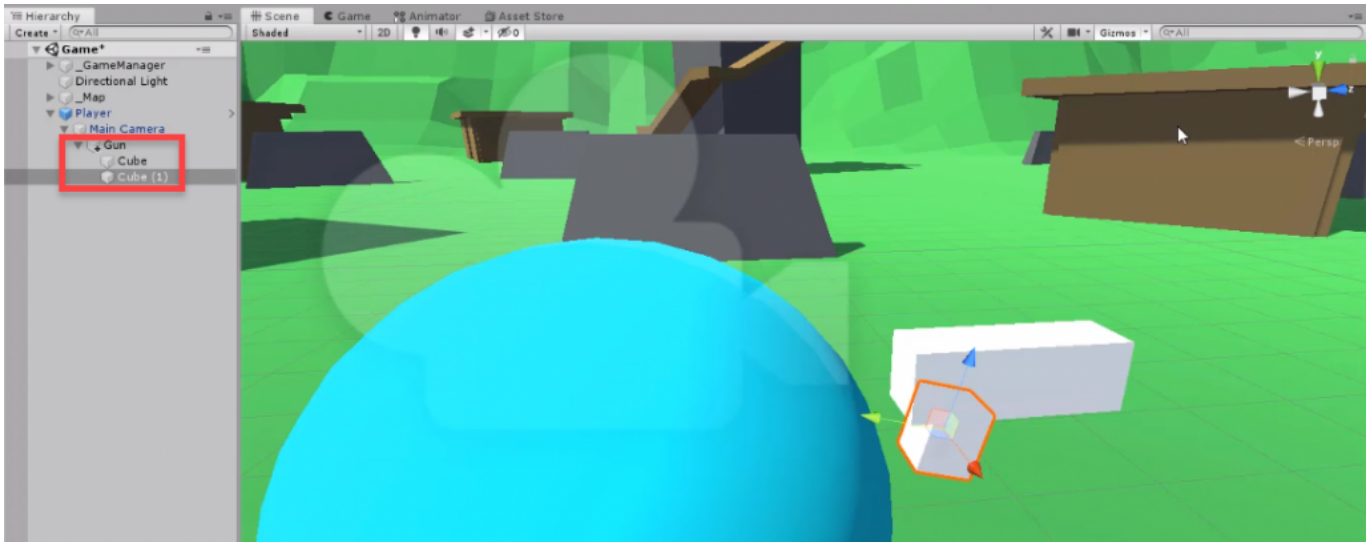
Gun Model

Drag in the player prefab as we're now going to create the gun model.

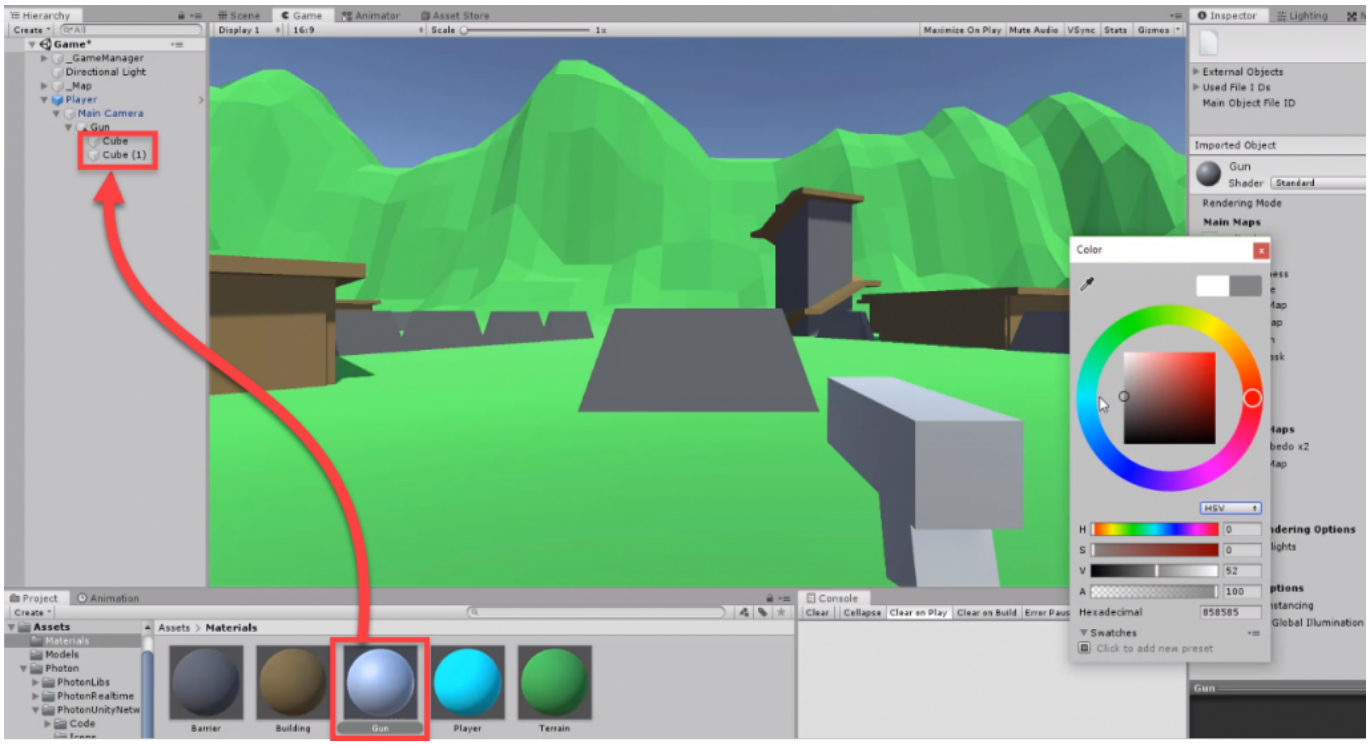
As the child of the camera, create a new empty game object called **Gun**. Then as a child of that, create a cube and remove the collider.



Scale, move and duplicate the cube to form the shape of a gun like below.



Let's also create a new material called **Gun** and apply it to the models. I made the albedo color a light grey.



Taking Damage

In the **PlayerController** script, let's add a new variable to keep track of the player who attacked us last.

```
private int curAttackerId;
```

We also need to know our health and some other things.

```
public int curHp;
public int maxHp;
public int kills;
public bool dead;
private bool flashingDamage;
public MeshRenderer mr;
```

At the top of the **Update** function, we want to make sure that only the player who controls this player, will run the update function.

```
if(!photonView.IsMine || dead)
    return;
```

When the player gets hit, the **TakeDamage** function gets called.

```
[PunRPC]
public void TakeDamage (int attackerId, int damage)
```



```
{
    if(dead)
        return;

    curHp -= damage;
    curAttackerId = attackerId;

    // flash the player red
    photonView.RPC("DamageFlash", RpcTarget.Others);

    // update the health bar UI

    // die if no health left
    if(curHp <= 0)
        photonView.RPC("Die", RpcTarget.All);
}
```

DamageFlash visually flashes the player red when they get hit.

```
[PunRPC]
void DamageFlash ()
{
    if(flashingDamage)
        return;

    StartCoroutine(DamageFlashCoRoutine());

    IEnumerator DamageFlashCoRoutine ()
    {
        flashingDamage = true;

        Color defaultColor = mr.material.color;
        mr.material.color = Color.red;

        yield return new WaitForSeconds(0.05f);

        mr.material.color = defaultColor;
        flashingDamage = false;
    }
}
```

The **Die** function gets called when our health goes below 0.

```
[PunRPC]
void Die ()
{

}
```


Player Weapon

Create a new C# script called **PlayerWeapon** and attach it to the player prefab.

```
using Photon.Pun;
using Photon.Realtime;
```

Let's fill in our variables.

```
[Header("Stats")]
public int damage;
public int curAmmo;
public int maxAmmo;
public float bulletSpeed;
public float shootRate;

private float lastShootTime;

public GameObject bulletPrefab;
public Transform bulletSpawnPos;

private PlayerController player;

void Awake ()
{
    // get required components
    player = GetComponent<PlayerController>();
}
```

The **TryShoot** function gets called when we press the left mouse button.

```
public void TryShoot ()
{
    // can we shoot?
    if(curAmmo <= 0 || Time.time - lastShootTime < shootRate)
        return;

    curAmmo--;
    lastShootTime = Time.time;

    // update the ammo UI

    // spawn the bullet
    player.photonView.RPC("SpawnBullet", RpcTarget.All, bulletSpawnPos.transform.posi
tion, Camera.main.transform.forward);
}
```

The **SpawnBullet** function, gets called when we shoot. This is called on everyone's computer.


```
[PunRPC]
void SpawnBullet (Vector3 pos, Vector3 dir)
{
    // spawn and orientate it
    GameObject bulletObj = Instantiate(bulletPrefab, pos, Quaternion.identity);
    bulletObj.transform.forward = dir;


}
```

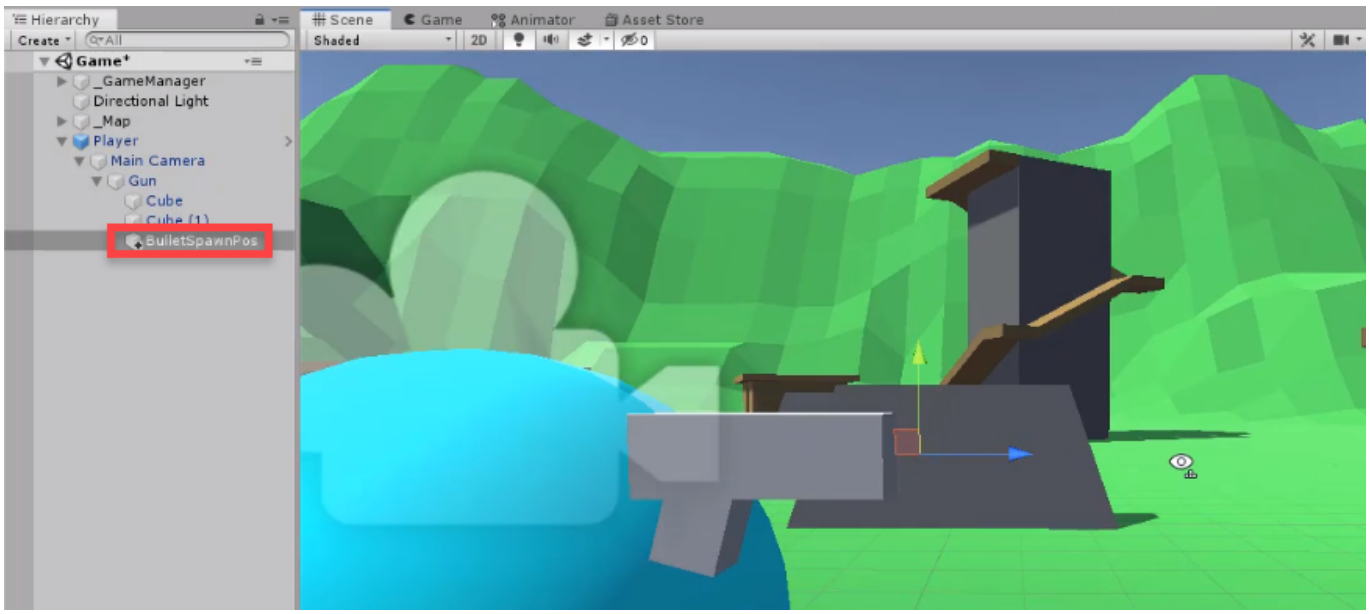
Back in the **PlayerController** script, we can shoot when we press the left mouse button.

```
if(Input.GetMouseButtonDown(0))
    weapon.TryShoot();
```

Let's add the player weapon component to the variables.

```
public PlayerWeapon weapon;
```

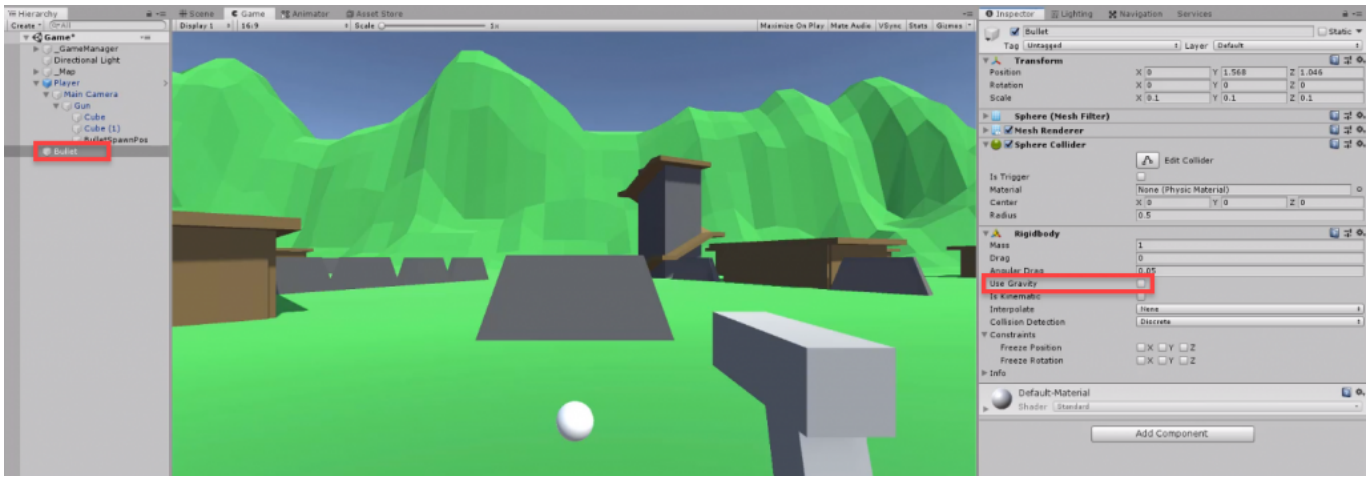
Back in the editor, we can create a new empty game object as a child of **Gun**. This will be where the bullets are spawned from.



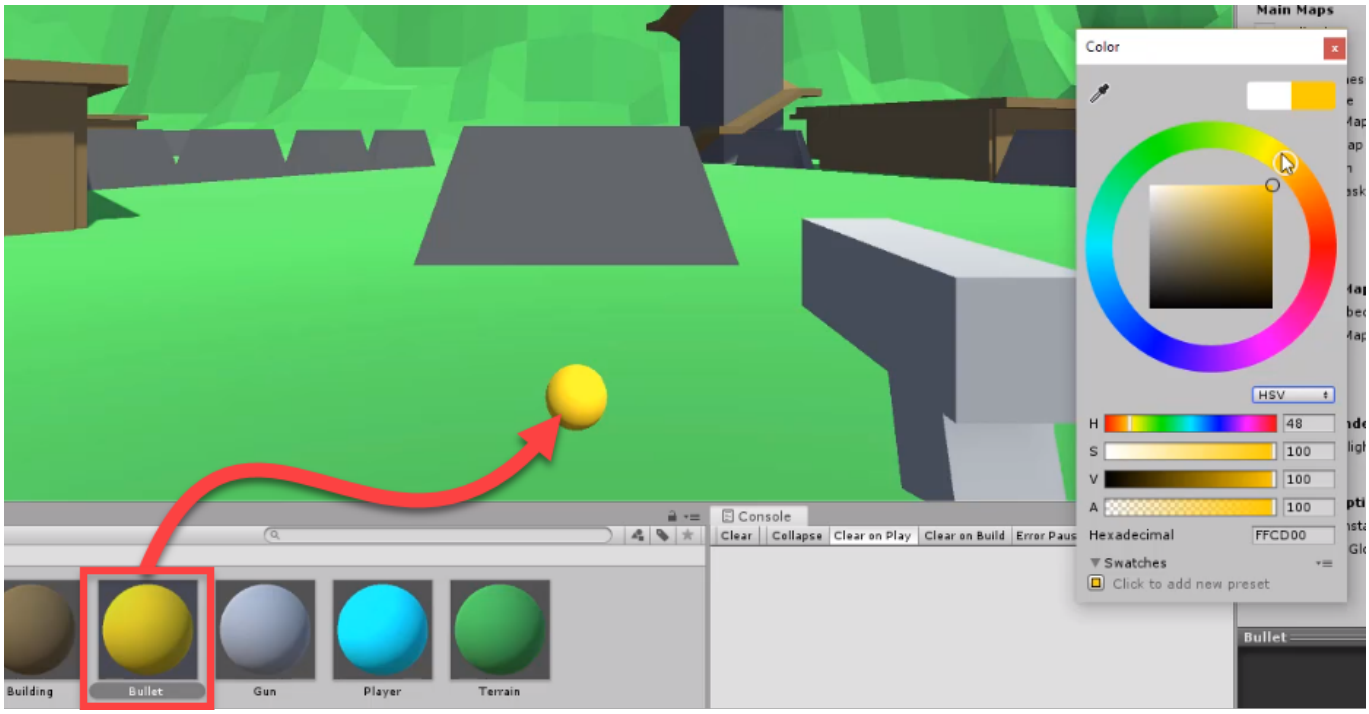
Creating a Bullet

Let's start this lesson by creating the bullet object. Create a new sphere object and call it **Bullet**.

- Set the **Scale** to *0.1*
- Add a **Rigidbody** component
- Disable **Use Gravity**
- Set **Is Trigger** on the Sphere Collider (not seen in image)



We can also create a new material called **Bullet**, set the color to yellow and apply it to the bullet sphere.



Bullet Script

Create a new C# script and call it **Bullet**.

Let's start with the variables.


```
private int damage;
private int attackerId;
private bool isMine;

public Rigidbody rig;
```

The **Initialize** function gets called when the bullet is created.

```
public void Initialize (int damage, int attackerId, bool isMine)
{
    this.damage = damage;
    this.attackerId = attackerId;
    this.isMine = isMine;

    Destroy(gameObject, 5.0f);
}
```

GetPlayer Functions

Soon we'll be needing to find other players in the game. A way we can do this is through two different functions in the **GameManager** script.

```
public PlayerController GetPlayer (int playerId)
{
    return players.First(x => x.id == playerId);
}

public PlayerController GetPlayer (GameObject playerObj)
{
    return players.First(x => x.gameObject == playerObj);
}
```

Back to the Bullet Script

The **OnTriggerEnter** function gets called when our trigger enters the collider of another object.

```
void OnTriggerEnter (Collider other)
{
    if(other.CompareTag("Player") && isMine)
    {
        PlayerController player = GameManager.instance.GetPlayer(other.gameObject);

    }
}
```

We'll finish this off in the next lesson.

Bullet Script

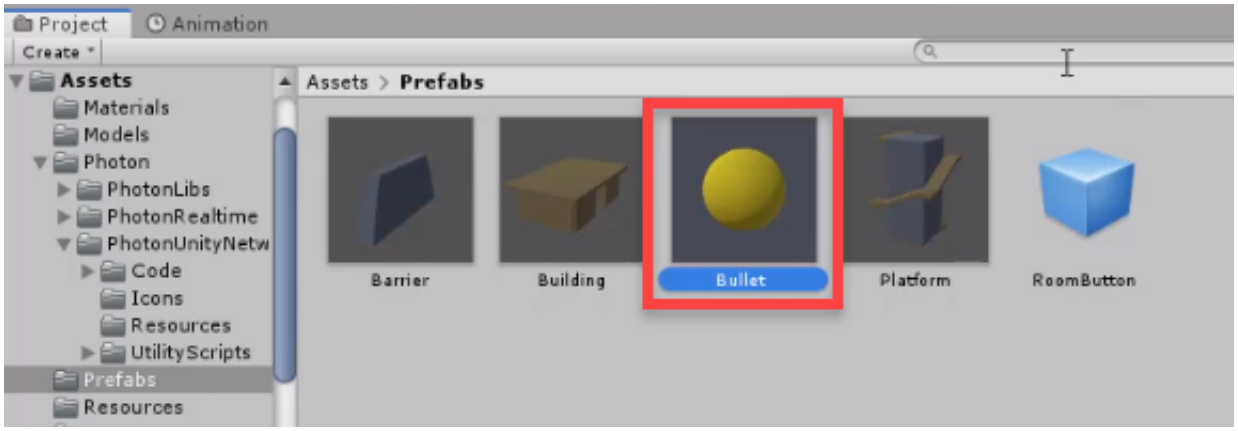
Let's continue off in the **Bullet** script and finish off the **OnTriggerEnter** function.

```
void OnTriggerEnter (Collider other)
{
    // did we hit a player?
    // if this is the local player's bullet, damage the hit player
    // we're using client side hit detection
    if(other.CompareTag("Player") && isMine)
    {
        PlayerController player = GameManager.instance.GetPlayer(other.gameObject);

        if(player.id != attackerId)
            player.photonView.RPC("TakeDamage", player.photonPlayer, attackerId, dama
ge);
    }

    Destroy(gameObject);
}
```

Back in the **PlayerWeapon** script, we can also finish off the **SpawnBullet** function.

```
// get bullet script
Bullet bulletScript = bulletObj.GetComponent<Bullet>();

// initialize it and set the velocity
bulletScript.Initialize(damage, player.id, player.photonView.IsMine);
bulletScript.rig.velocity = dir * bulletSpeed;
```

Back in the editor, we can make the bullet a prefab and connect it to the player object.



Game Manager

In the **GameManager** script, we'll create a new variable to keep track of how long we wait after a game has been won, before going to the menu.

```
public float postGameTime;
```

Then we can create a few functions which will check if a player has won and if so, win the game.

```
public void CheckWinCondition ()
{
    if(alivePlayers == 1)
        photonView.RPC("WinGame", RpcTargets.All, players.First(x => !x.dead).id);
}
```

```
[PunRPC]
void WinGame (int winningPlayer)
{
    // set the UI win text

    Invoke("GoBackToMenu", postGameTime);
}
```

```
void GoBackToMenu ()
{
    NetworkManager.instance.ChangeScene("Menu");
}
```

Player Controller

Over in the **PlayerController** script, we can check the win condition when we die.

```
[PunRPC]
void Die ()
{
    curHp = 0;
    dead = true;

    GameManager.instance.alivePlayers--;

    // host will check win condition
    if(PhotonNetwork.IsMasterClient)
        GameManager.instance.CheckWinCondition();

    // is this our local player?
    if(photonView.IsMine)
    {
```

```
        if(curAttacker != 0)
            GameManager.instance.GetPlayer(curAttacker).photonView.RPC("AddKill", RpcTarget.All);

        // set the cam to spectator
        GetComponentInChildren<CameraController>().SetAsSpectator();

        // disable the physics and hide the player
        rig.isKinematic = true;
        transform.position = new Vector3(0, -50, 0);
    }
}
```

The **AddKill** function gives the player a kill.

```
[PunRPC]
public void AddKill ()
{
    kills++;
}
```

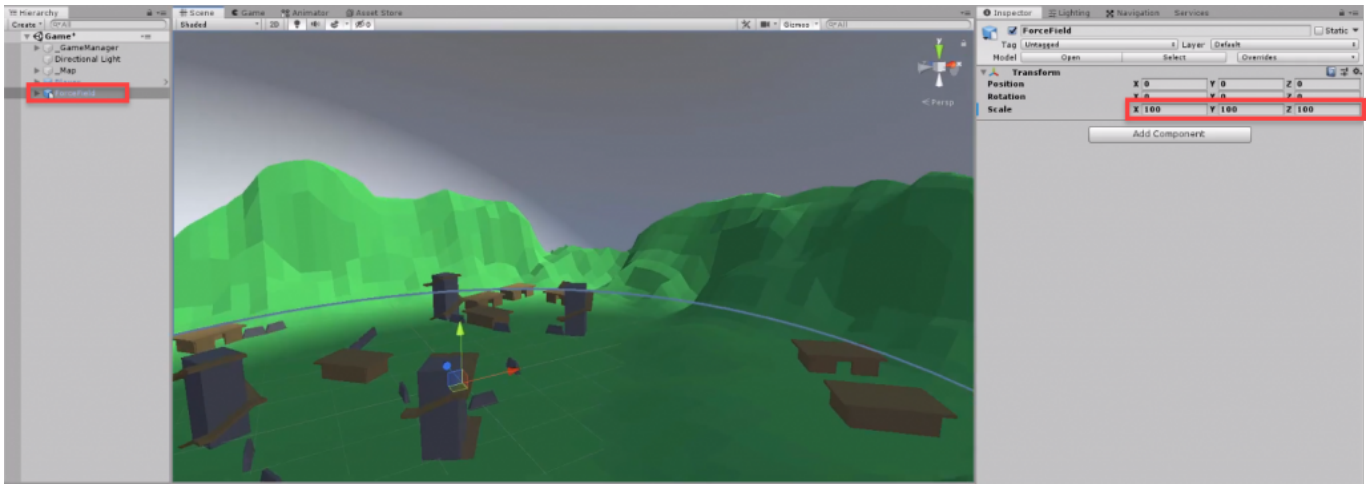
Camera Controller

In the **CameraController** script, we need to create the **SetAsSpectator** function.

```
public void SetAsSpectator ()
{
    isSpectator = true;
    transform.parent = null;
}
```

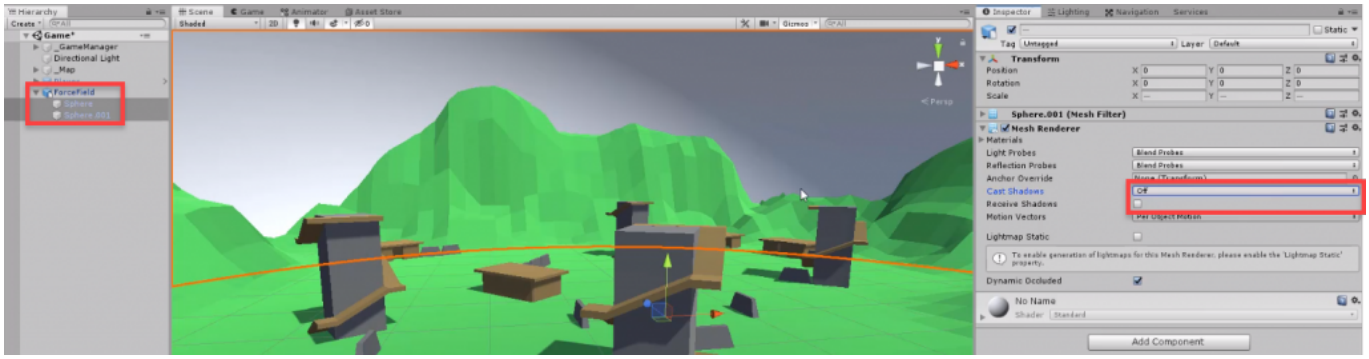

Force Field Object

Drag in the **ForceField** model and set the scale to *100*.

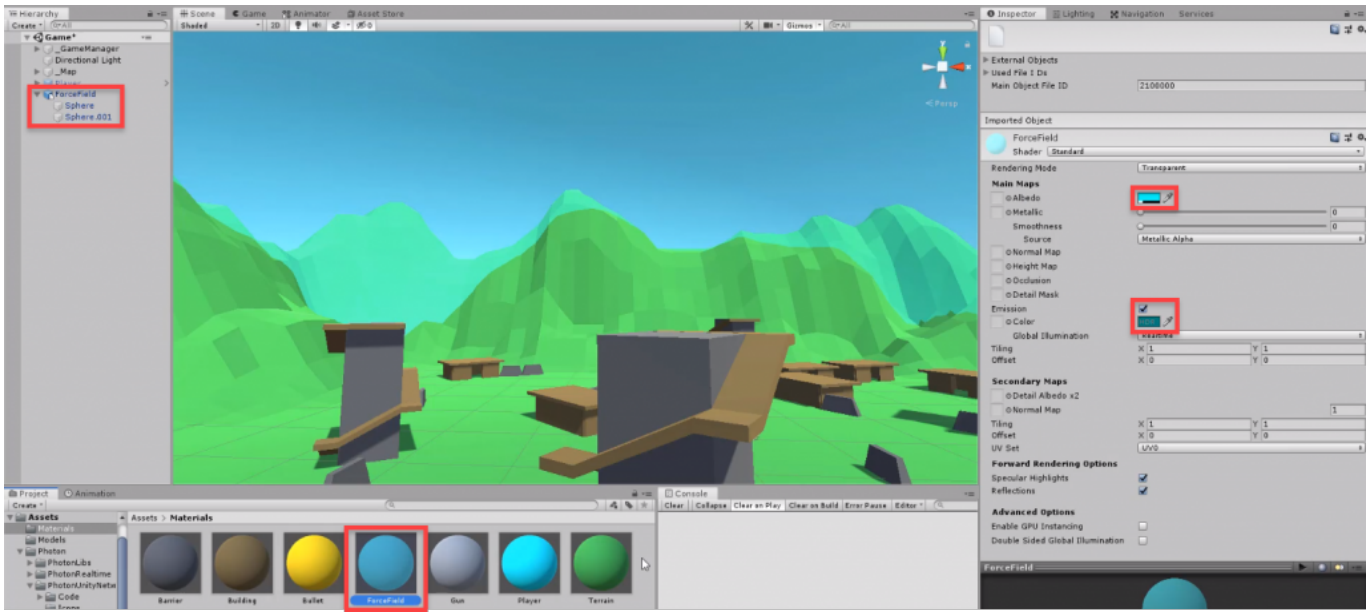


You might notice that the force field casts a shadow – we don't want that. Inside of the object, select the 2 model objects.

- Set **Cast Shadows** to *Off*
- Disable **Receive Shadows**



Let's also then create a new material for the force field.



ForceField Script

Create a new C# script called **ForceField**. We'll have a few variables.

```
public float shrinkWaitTime;
public float shrinkAmount;
public float shrinkDuration;
public float minShrinkAmount;

public int playerDamage;

private float lastShrinkEndTime;
private bool shrinking;
private float targetDiameter;
private float lastPlayerCheckTime;
```

We can then set some initial values in the **Start** function.

```
void Start ()
{
    lastShrinkEndTime = Time.time;
    targetDiameter = transform.localScale.x;
}
```

The **Update** function will manage the shrinking and damaging.

```
void Update ()
{
    if(shrinking)
    {
        transform.localScale = Vector3.MoveTowards(transform.localScale, Vector3.one
* targetDiameter, (shrinkAmount / shrinkDuration) * Time.deltaTime);
```



```
        if(transform.localScale.x == targetDiameter)
            shrinking = false;
    }
    else
    {

    }
}
```

We'll finish this script off in the next lesson.

Finishing the ForceField Script

Back in our **ForceField** script, we can continue off in the else statement.

```
else
{
    // can we shrink again?
    if(Time.time - lastShrinkEndTime >= shrinkWaitTime && transform.localScale.x > mi
nShrinkAmount)
        Shrink();
}
```

The **Shrink** function, will calculate a new diameter and begin to shrink.

```
void Shrink ()
{
    shrinking = true;

    // make sure we don't shrink below the min amount
    if(transform.localScale.x - shrinkAmount > minShrinkAmount)
        targetDiameter -= shrinkAmount;
    else
        targetDiameter = minShrinkAmount;

    lastShrinkEndTime = Time.time + shrinkDuration;
}
```

Back in the editor we can figure out the values. These will be changed later.

- Set **Shrink Wait Time** to *2*
- Set **Shrink Amount** to *10*
- Set **Shrink Duration** to *1*
- Set **Min Shrink Amount** to *15*
- Set **Player Damage** to *20*

Now if you press play, you should be able to see the force field shrink.

The **CheckPlayers** function loops through the players every second and checks their distance from the center. If they're outside of the force field, damage them.

```
void CheckPlayers ()
{
    if(Time.time - lastPlayerCheckTime > 1.0f)
    {
        lastPlayerCheckTime = Time.time;

        // loop through all players
        foreach(PlayerController player in GameManager.instance.players)
        {
            if(player.dead || !player)
                continue;
```



```
            if(Vector3.Distance(Vector3.zero, player.transform.position) >= transform
.localScale.x)
            {
                player.photonView.RPC("TakeDamage", player.photonPlayer, 0, playerDam
age);
            }
        }
    }
}
```

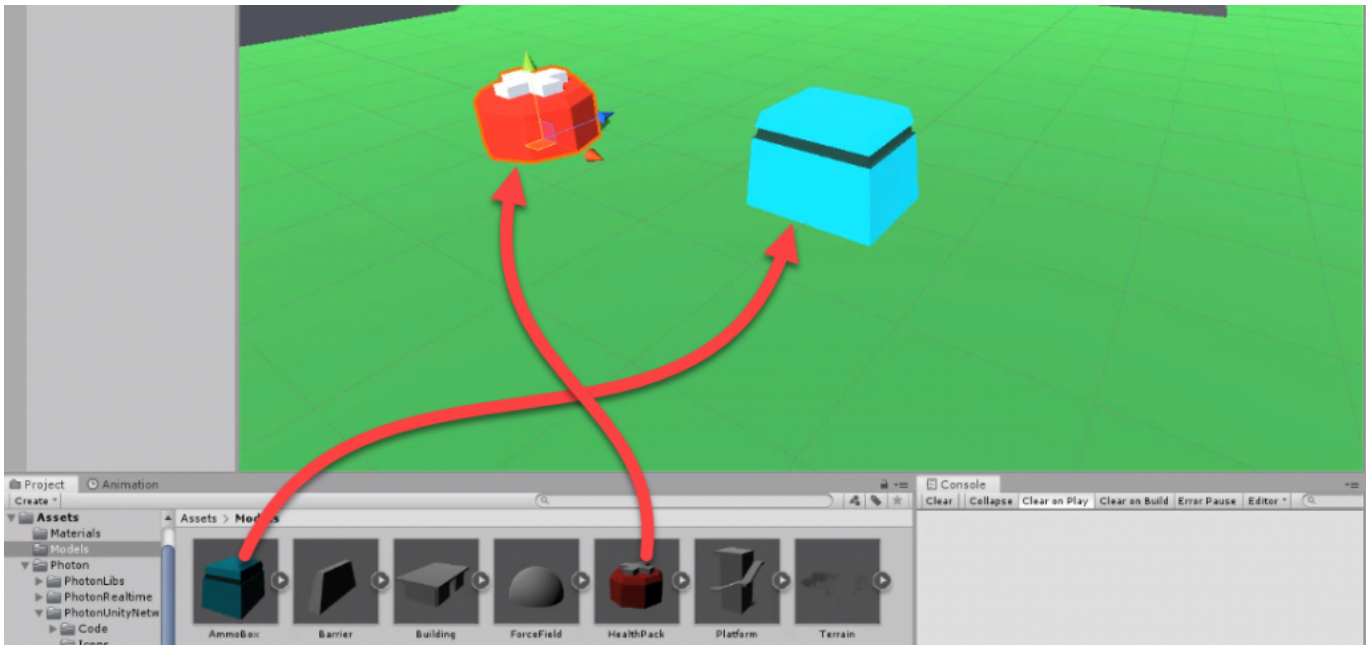
We'll be calling this in the **Update** function.

```
CheckPlayers();
```

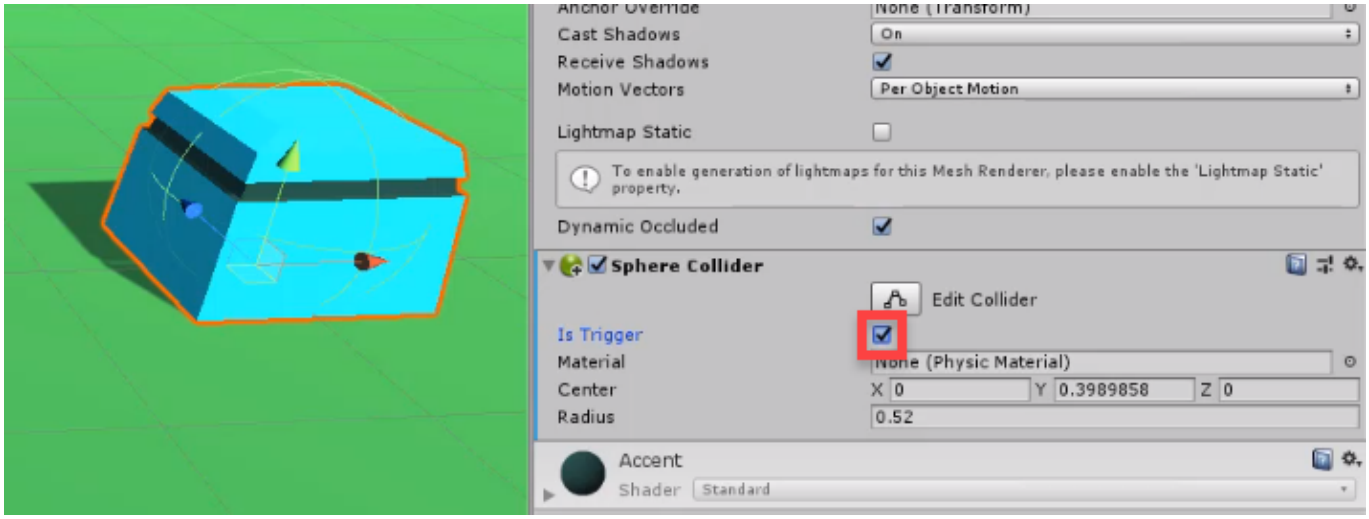
You should now be able to get damaged when you leave the force field.

Creating the Pickups

We're going to have 2 pickups – health and ammo. To begin, drag in the **HealthPack** and **AmmoBox** models.



For both models, add a **Sphere Collider** component and enable **Is Trigger**.



Pickup Script

Now, create a new C# script called **Pickup** and attach it to both pickups. First, we want to create an enumerator for our pickup types. This is basically a list of different options we can create and use it as a data type. Write this down above the class.

```
public enum PickupType
{
    Health,
    Ammo
}
```


Let's also include the Photon namespace.

```
using Photon.Pun;
```

Then back in the class, we can write down our variables.

```
public PickupType type;
public int value;
```

Pickup Functions

When a player picks up a pickup, one of two functions will be called. One to heal the player, and the other to give them ammo. In the **PlayerController** script, add the **Heal** function.

```
[PunRPC]
public void Heal (int amountToHeal)
{
    curHp = Mathf.Clamp(curHp + amountToHeal, 0, maxHp);

    // update the health bar UI
}
```

Then in the **PlayerWeapon** class, we can create the **GiveAmmo** function.

```
[PunRPC]
public void GiveAmmo (int ammoToGive)
{
    curAmmo = Mathf.Clamp(curAmmo + ammoToGive, 0, maxAmmo);

    // update the ammo text
}
```

Back in the Pickup Script

Back in the **Pickup** script, the only function we're going to create is **OnTriggerEnter** which gets called when our collider intersects another one.

```
void OnTriggerEnter (Collider other)
{
    if(!PhotonNetwork.IsMasterClient)
        return;

    if(other.CompareTag("Player"))
    {
        // get the player
```



```
        PlayerController player = GameManager.instance.GetPlayer(other.gameObject);

        if(type == PickupType.Health)
            player.photonView.RPC("Heal", player.photonPlayer, value);
        else if(type == PickupType.Ammo)
            player.photonView.RPC("GiveAmmo", player.photonPlayer, value);

        // destroy the object
        PhotonNetwork.Destroy(gameObject);
    }
}
```

In the Editor

Back in the Editor, we can first attach a **PhotonView** component to each pickup. Then set their respective **Pickup Type** and value. Save these two as prefabs in the **Prefabs** folder.

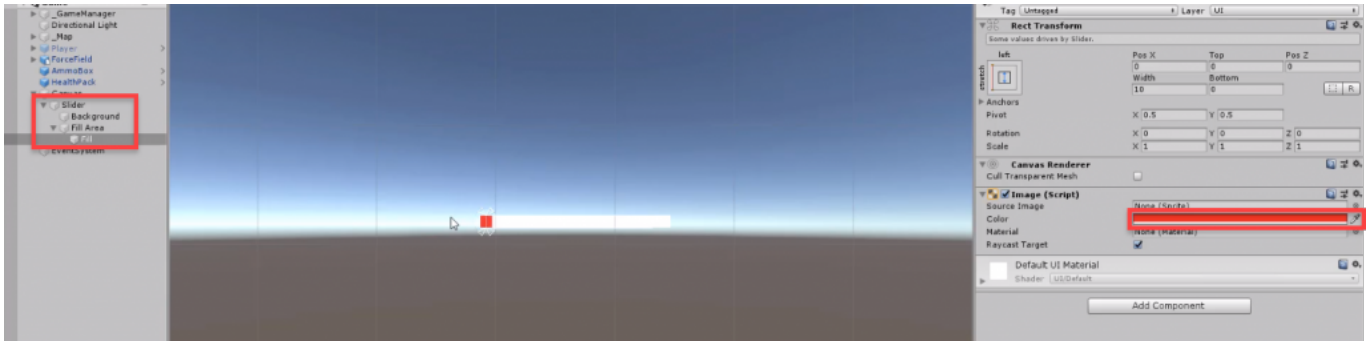
Creating the Game UI

In this lesson, we're going to be creating our game UI. First, create a new canvas. We're going to create a few elements:

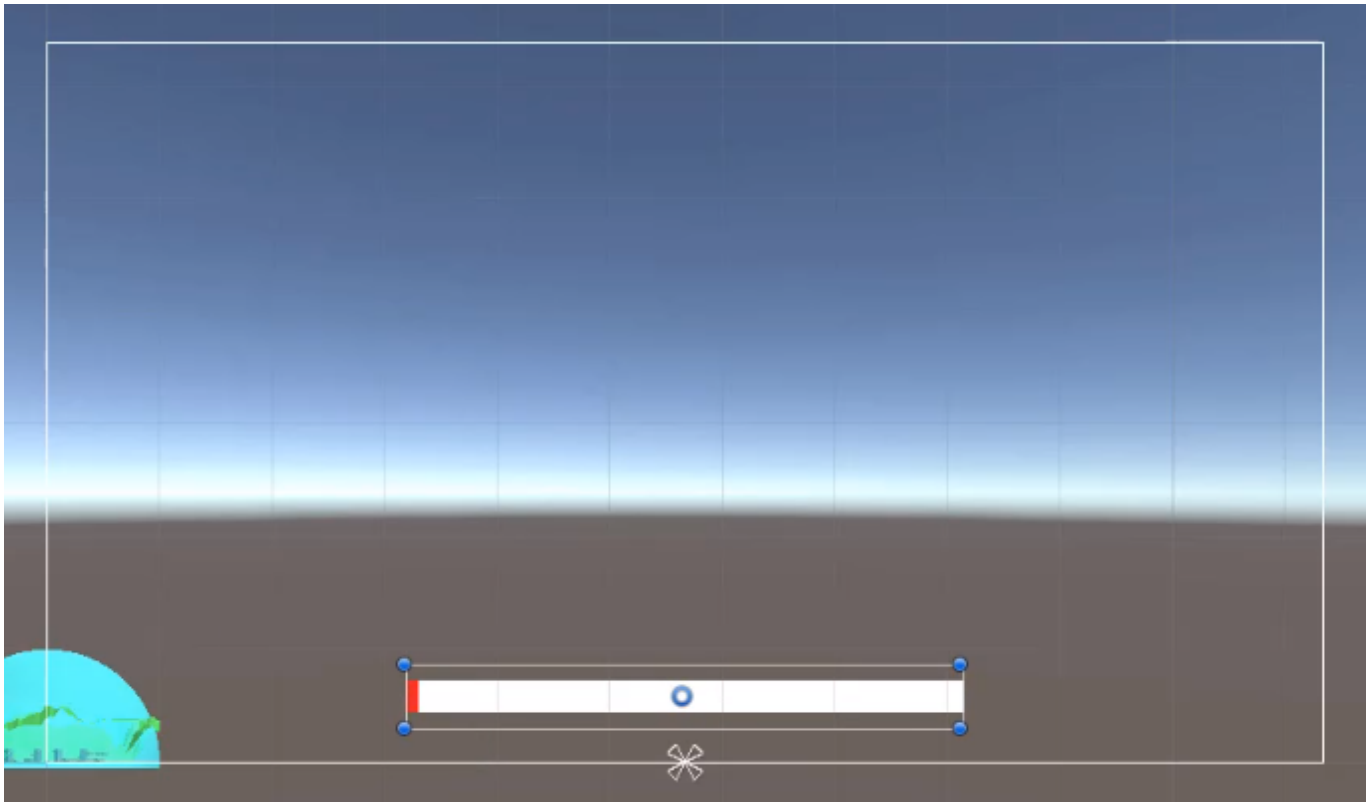
- Health bar at the bottom center of the screen
- Ammo counter at the bottom right of the screen
- Player info at the top left of the screen
- Win text in the center of the screen

Health Bar

To create the health bar, add a new slider in the canvas. Set the **Background** and **Fill** source image to *None*, and then set the background color to white, and fill color to red. Also delete the handle as we won't be needing to drag it manually.

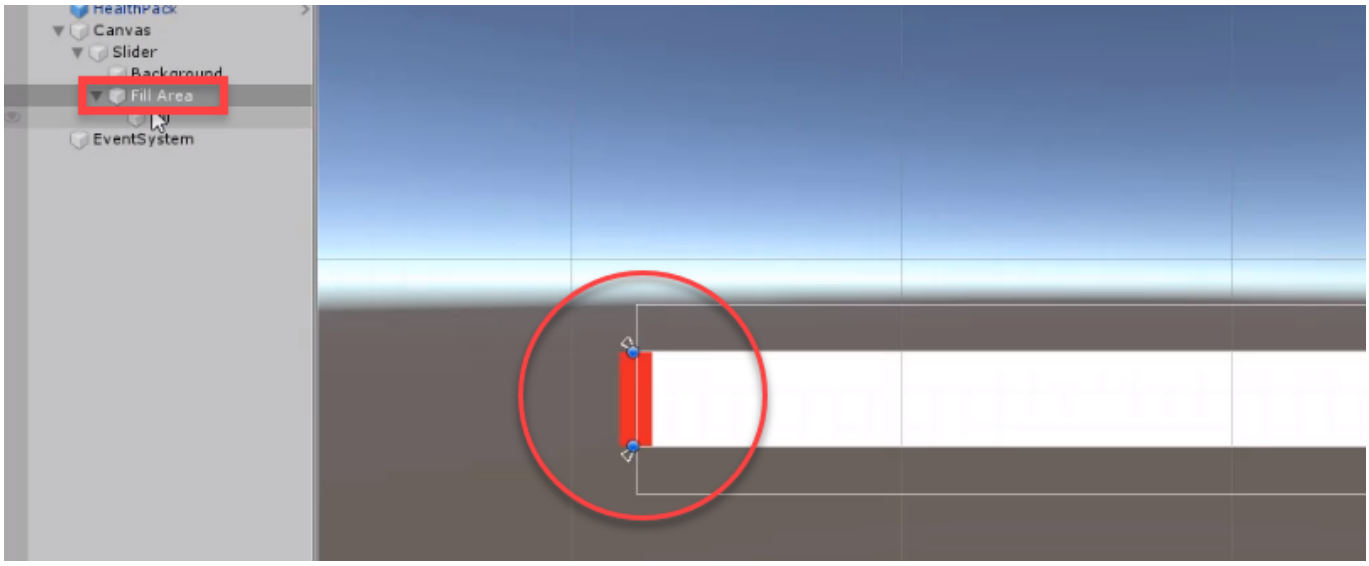


Position it down at the bottom center of the screen with the anchoring following suit.

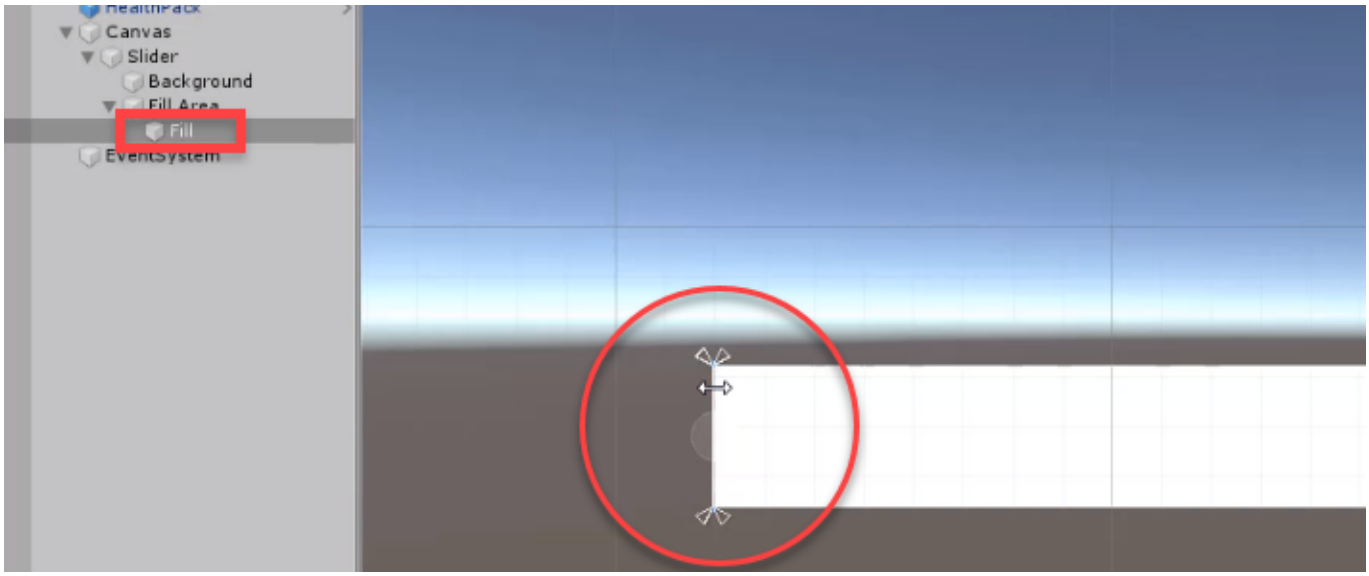


You might notice that the slider is at a value of 0, yet still has some red visible. To fix this, select

the **Fill Area** and resize the rect horizontally so it's the same width as the slider.



Then select the **Fill** and resize it to a width of 0.

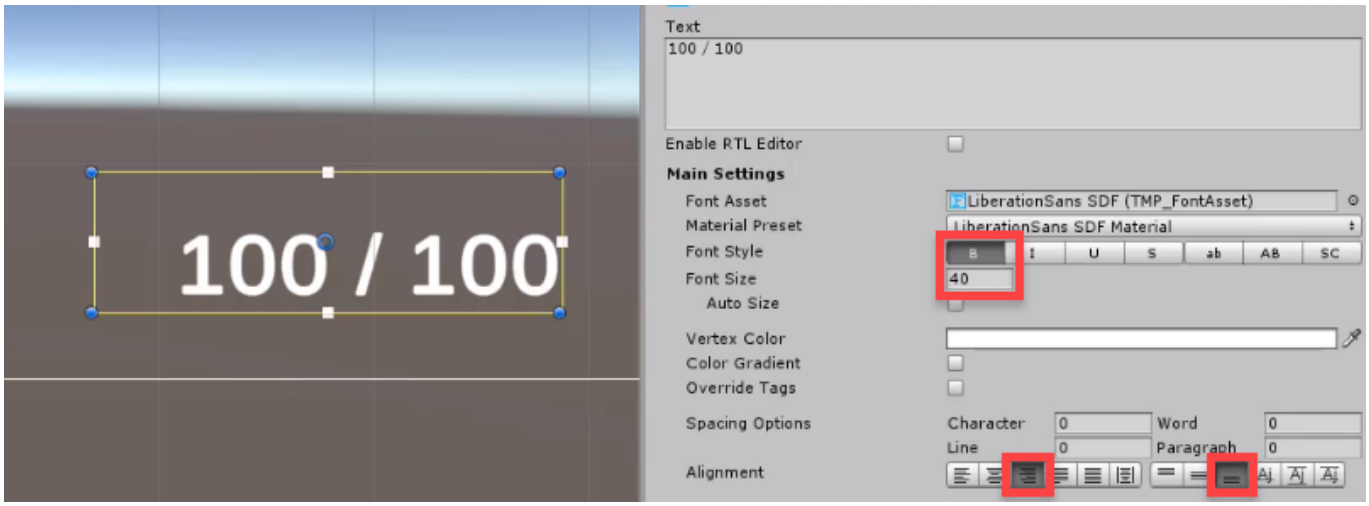


Now you should be able to change the value and have the correct fill amount.

Ammo Text

Create a new **Text Mesh Pro – Text** element and call it **AmmoText**. Position it to the bottom left corner of the screen and do the same with the anchoring.

- Set the **Font Style** to *Bold*
- Set the **Font Size** to *40*
- Set the **Alignment** to *right-bottom*



Player Info Text

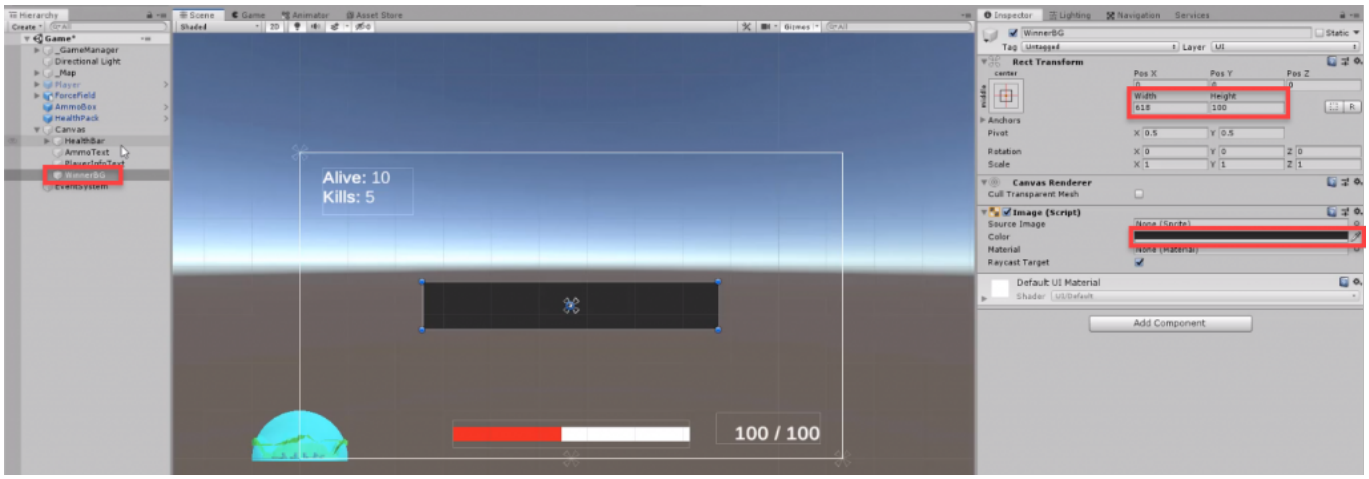
For the player info, duplicate the ammo text and move it to the top left corner of the screen, setting the same for the anchoring. Rename it to **PlayerInfoText**.

- Set the **Font Size** to *35*
- Set the **Alignment** to l*eft-top*

Win Text

For the win text, create a new image for the background. Call it **WinnerBG** and set the **Color** to black-grey.

- Set the **Width** to *618*
- Set the **Height** to *100*



Now add a new text mesh pro text element as the child of the image.

- Set the **Width** and **Height** to the same as the image
- Set the **Font Style** to *Bold*
- Set the **Font Size** to *50*
- Set the **Alignment** to *middle-center*
- Set the **Vertex Color** to *green* (not seen in image)

We can now deactivate the **WinnerBG**.

GameUI Script

Create a new C# script called **GameUI** and attach it to the **_GameManager** object.

We first need to add our namespaces.

```
using UnityEngine.UI;
using TMPro;
using Photon.Pun;
```

Let's now add our variables.

```
public Slider healthBar;
public TextMeshProUGUI playerInfoText;
public TextMeshProUGUI ammoText;
public TextMeshProUGUI winText;
public Image winBackground;

private PlayerController player;

// instance
public static GameUI instance;

void Awake ()
{
    instance = this;
}
```

Then we need to create the **Initialize** function, which gets called when our local player is spawned in.

```
public void Initialize (PlayerController localPlayer)
{
    player = localPlayer;
    healthBar.maxValue = player.maxHp;
    healthBar.value = player.curHp;

    UpdatePlayerInfoText();
    UpdateAmmoText();
}
```

In the **PlayerController** script, let's initialize the game UI in the **Initialize** function.

```
else
{
    GameUI.instance.Initialize(this);
}
```


Then back in the **GameUI** script, we can continue with the **UpdateHealthBar** function.

```
public void UpdateHealthBar ()
{
    healthBar.value = player.curHp;
}
```

The **UpdatePlayerInfoText** function updates the player info text.

```
public void UpdatePlayerInfoText ()
{
    playerInfoText.text = "<b>Alive:</b> " + GameManager.instance.alivePlayers + "\n<
b>Kills:</b> " + player.kills;
}
```

The **UpdateAmmoText** function updates the players ammo text.

```
public void UpdateAmmoText ()
{
    ammoText.text = player.weapon.curAmmo + " / " + player.weapon.maxAmmo;
}
```

Finally, the **SetWinText** function gets called when a player has won the game.

```
public void SetWinText (string winnerName)
{
    winBackground.gameObject.SetActive(true);
    winText.text = winnerName + " wins";
}
```

Hooking up the Functions

Let’s now go through and connect the functions to where they need to be called from.

UpdateHealthBar needs to be called from:

- PlayerController.TakeDamage()
- PlayerController.Heal()

UpdateAmmoText needs to be called from:

- PlayerWeapon.TryShoot()
- PlayerWeapon.GiveAmmo()

UpdatePlayerInfoText needs to be called from:

- PlayerController.AddKill()

SetWinText needs to be called from:

- GameManager.WinGame()
    - `GameUI.instance.SetWinText(GetPlayer(winningPlayer).photonPlayer.NickName);`

Force Field

Set the force field's properties to:

- **Shrink Wait Time:** 30
- **Shrink Amount:** 20
- **Shrink Duration:** 10
- **Min Shrink Amount:** 15
- **Player Damage:** 20

Game Manager

Set the game manager's properties to:

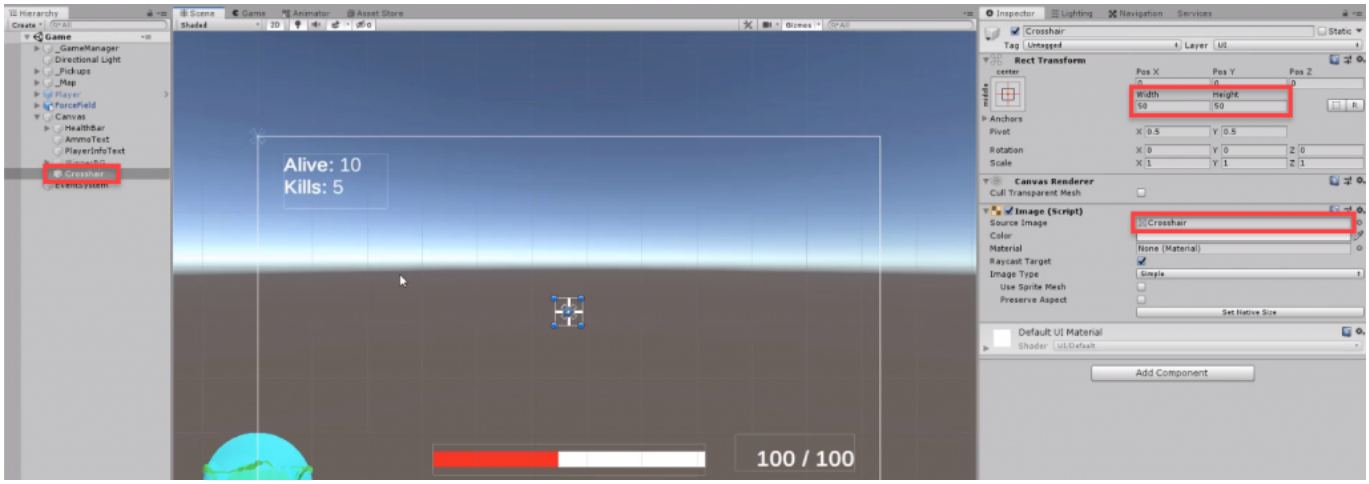
- **Post Game Time:** 3

Now duplicate the spawn points and place many more around the map to have a more varied spread of players. Do the same for the 2 types of pickups.

Crosshair

We can also add a crosshair to our game. Inside our game canvas, create a new image and call it **Crosshair**.

- Set the **Width** and **Height** to *50*
- Set the **Source Image** to *Crosshair*



Fixing the Problems that Occur When a Player Disconnects

Right now if you we're to play the game with multiple players and one disconnects, some problems may occur. To fix this, we need to add in support for players disconnecting. First, when a player disconnects due to a bad internet connection or some other problem, we don't want them to remain in the game scene. This will confuse them and the other players.

So in the **NetworkManager** script, let's add a **OnDisconnected** callback function which gets called when the player disconnects from Photon.

```
public override void OnDisconnected(DisconnectCause)
{
    PhotonNetwork.LoadLevel("Menu");
}
```

Then for all the other players still in the game, we need them to update their UI to show that someone has left. The master client will also check the win condition.

```
public override void OnPlayerLeftRoom (Player otherPlayer)
{
    GameManager.instance.alivePlayers--;
    GameUI.instance.UpdatePlayerInfoText();

    if(PhotonNetwork.IsMasterClient)
    {
        GameManager.instance.CheckWinCondition();
    }
}
```

In the **GameManager** script, we also need to add support for if a player is null. This is because when a player leaves the room, all their owned photon views are destroyed, so the player list will have a null value in it. Replace the code inside of the **GetPlayer (int playerId)** with:

```
public PlayerController GetPlayer (int playerId)
{
    foreach(PlayerController player in players)
    {
        if(player != null && player.id == playerId)
            return player;
    }

    return null;
}
```

And replace the code inside of **GetPlayer (GameObject playerObject)** with:

```
public PlayerController GetPlayer (GameObject playerObject)
{
    foreach(PlayerController player in players)
```

```
    {
        if(player != null && player.gameObject == playerObject)
            return player;
    }

    return null;
}
```

Now if a player disconnects from your game, they will go back to the menu and all remaining players will have their UI updated and the win condition checked.

Congratulations on Finishing the Course

Let’s see what we learned…

- We learned how to use **Photon**, a multiplayer framework for Unity.
- We created a player controller who can move, jump, shoot, heal and add ammo.
- Health and ammo pickups.
- A force field which decreases in size over time.
- A fully working menu system with a lobby and lobby browser.
- And this was done using Unity’s UI system.