Welcome to the Course!

We’re going to be creating a multiplayer RPG inside of Unity, using Photon.

- **Photon Unity Networking** is a popular networking framework for Unity.
- We’ll be creating a **player controller**, allowing you to move around and attack.
- **Pickups** will grant the player either gold or health.
- There will be a **chat box**, allowing players to talk to each other.
- We’ll be making a **lobby** system, allowing players to find games to connect up with each other.
- This will be done using Unity’s **UI** system.

Let’s get started on the project!

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.

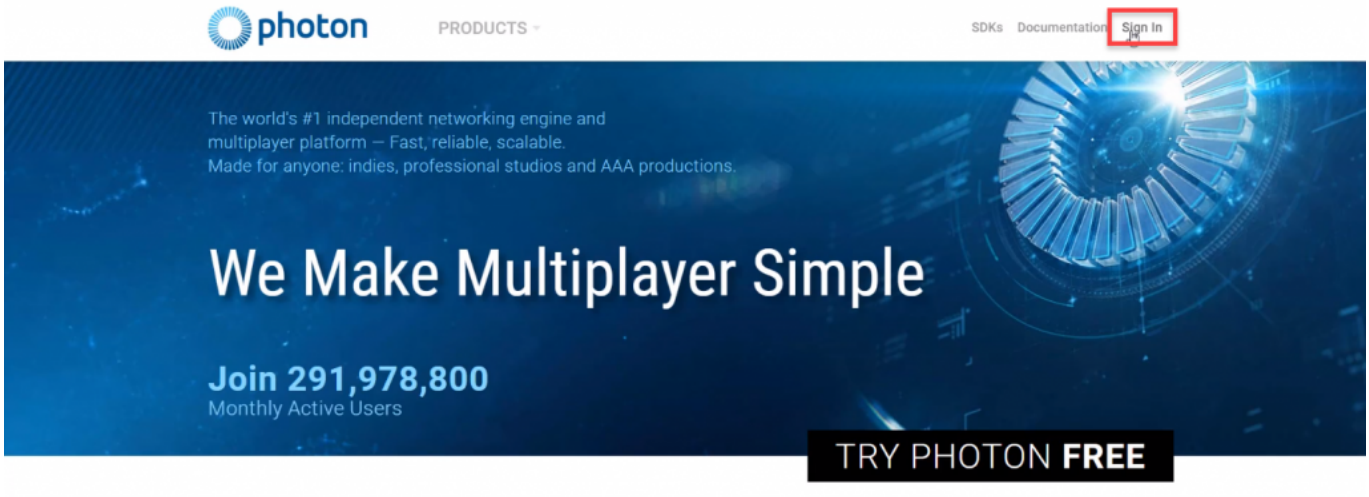
Players

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.

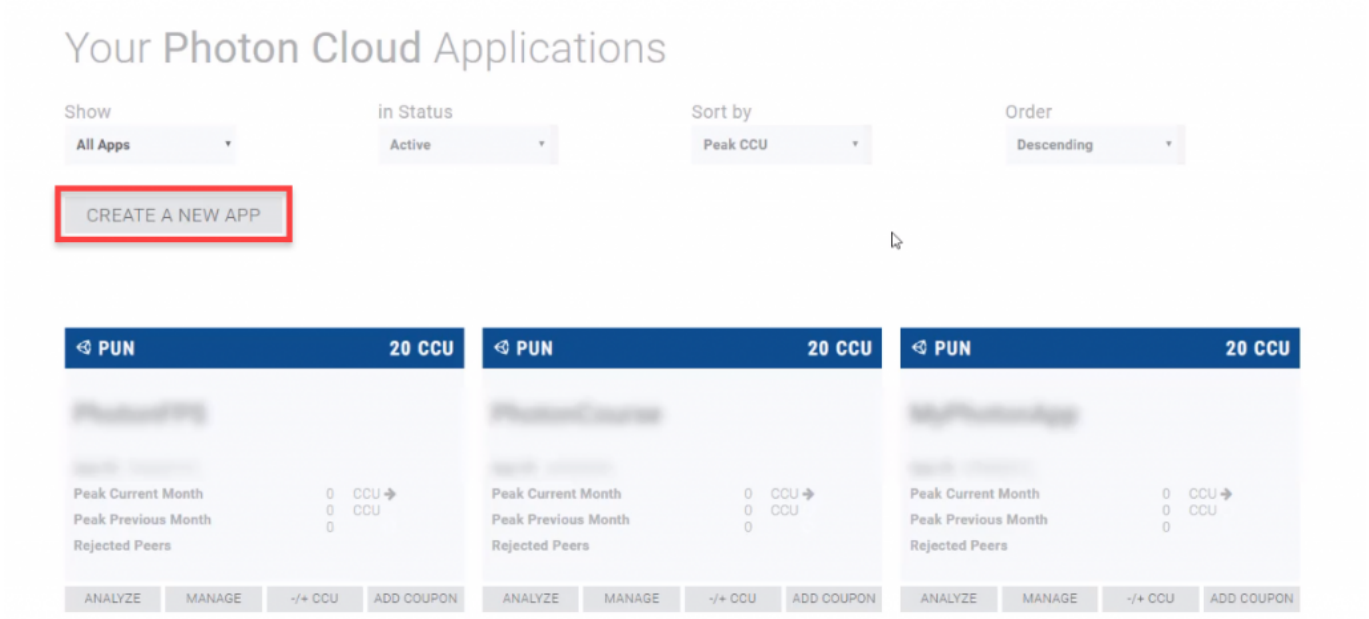
In the next lesson, we'll be setting up Photon.

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.



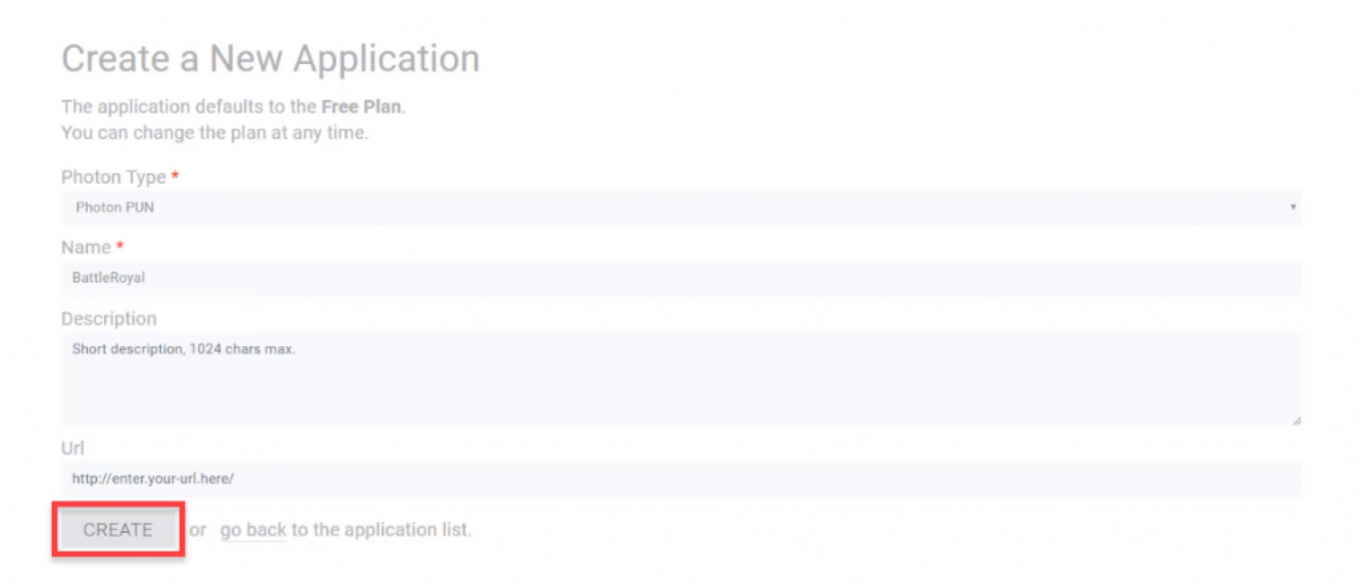
Once you sign in or create an account, you'll be taken to the **Applications** page. Here, we want to create a new app.



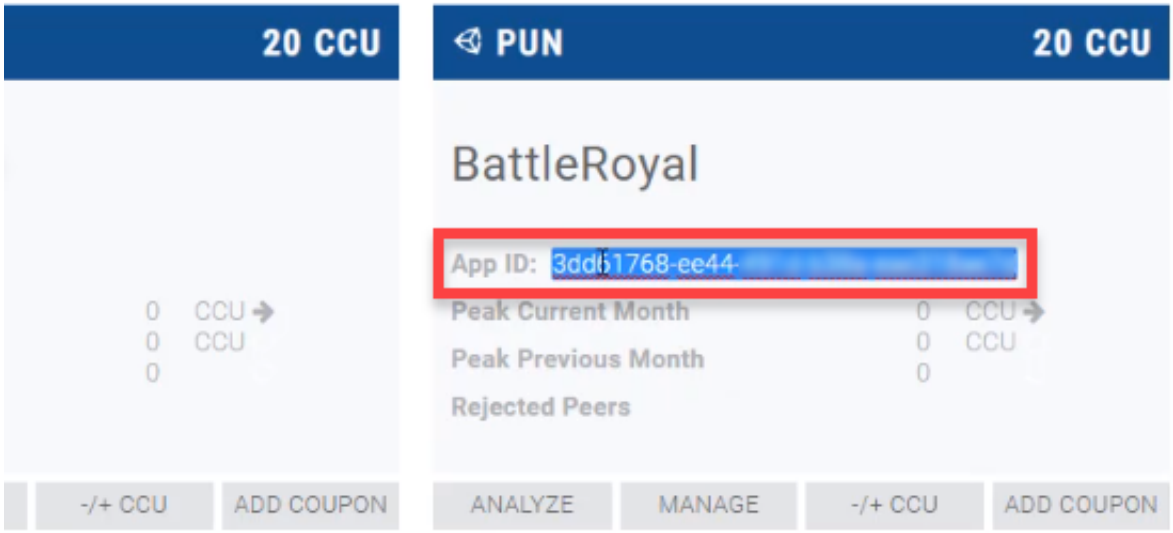
Here, we can fill in the info for our new application.

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

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

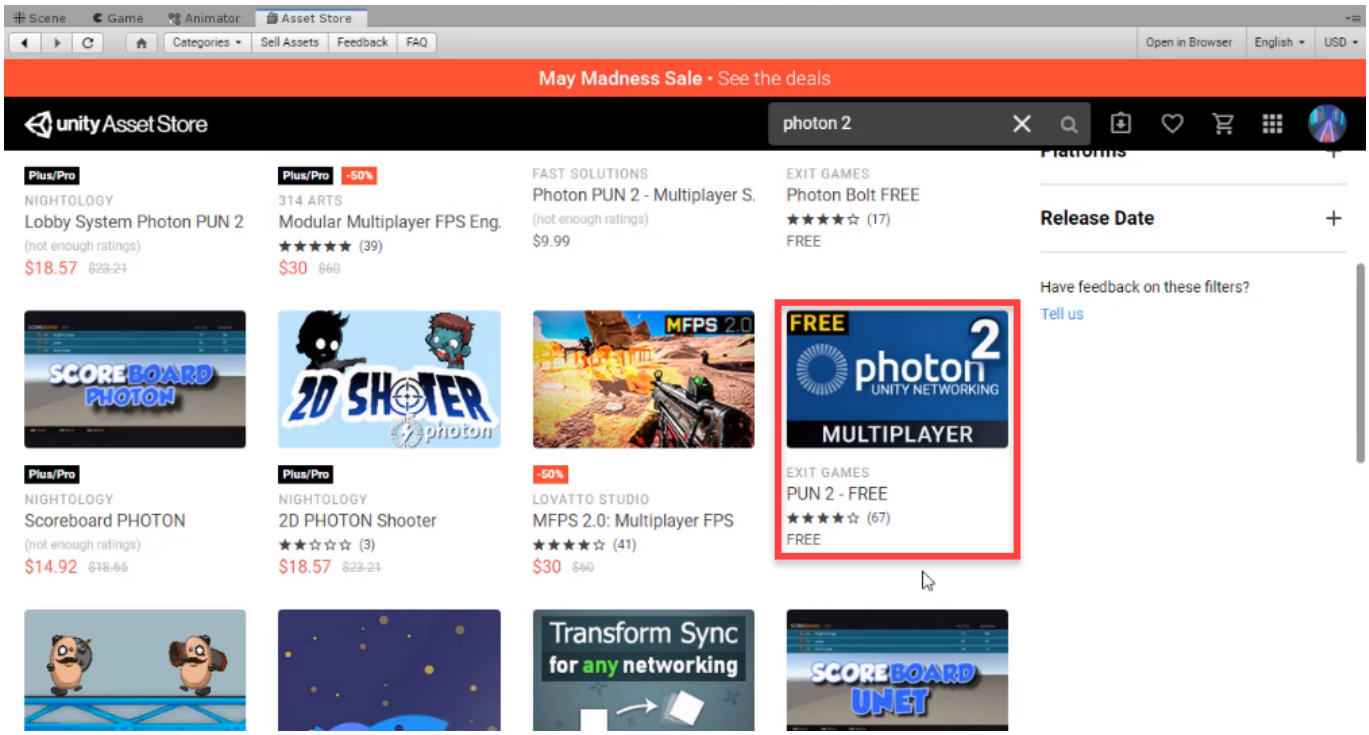


Now you should be taken back to the applications page. Your app should now be in the list. All we need from here is the **App ID**.



Importing Photon

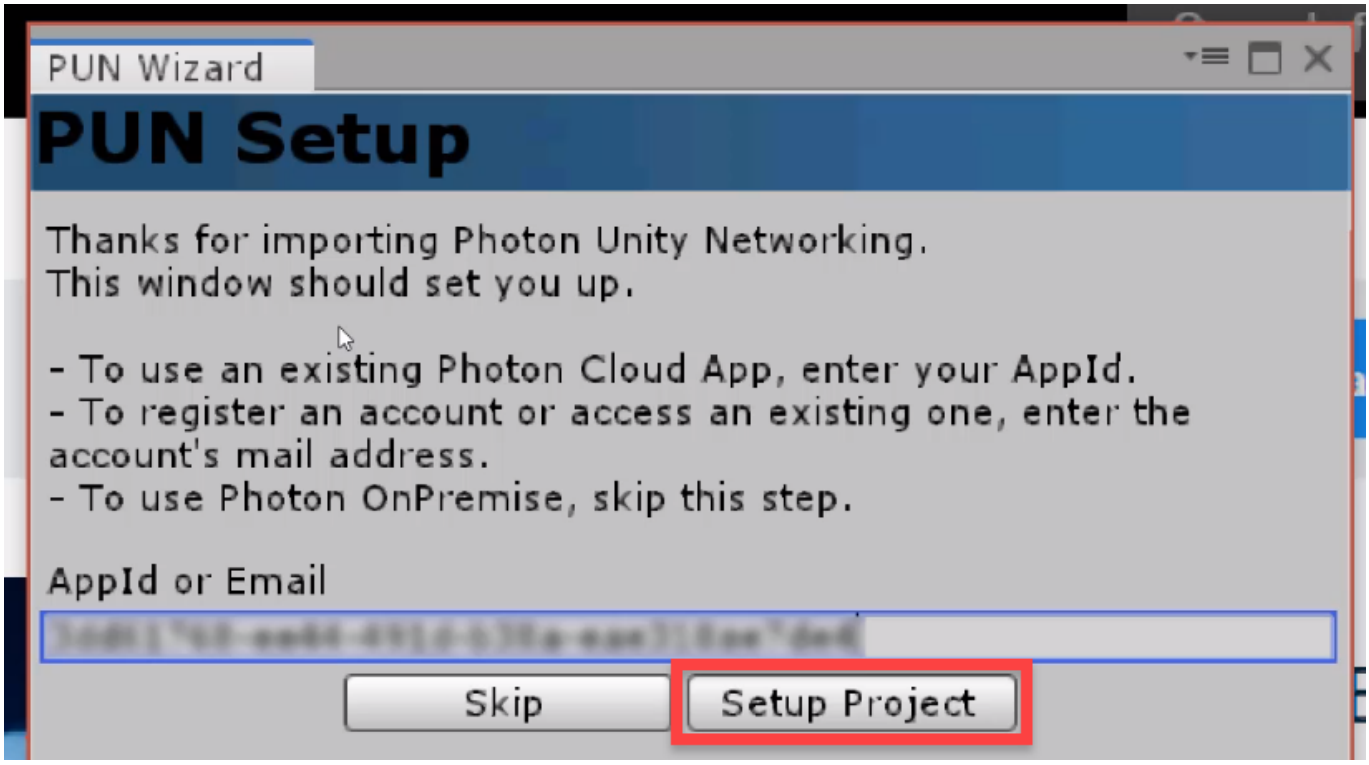
In Unity, we can now import the Photon asset. Open the **Asset Store** window (*Window > Asset Store*) and search for “photon 2”. Import and download the **PUN 2 – FREE** asset.



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

- PhotonChat
- PhotonRealtime > Demos
- PhotonUnityNetworking > Demos

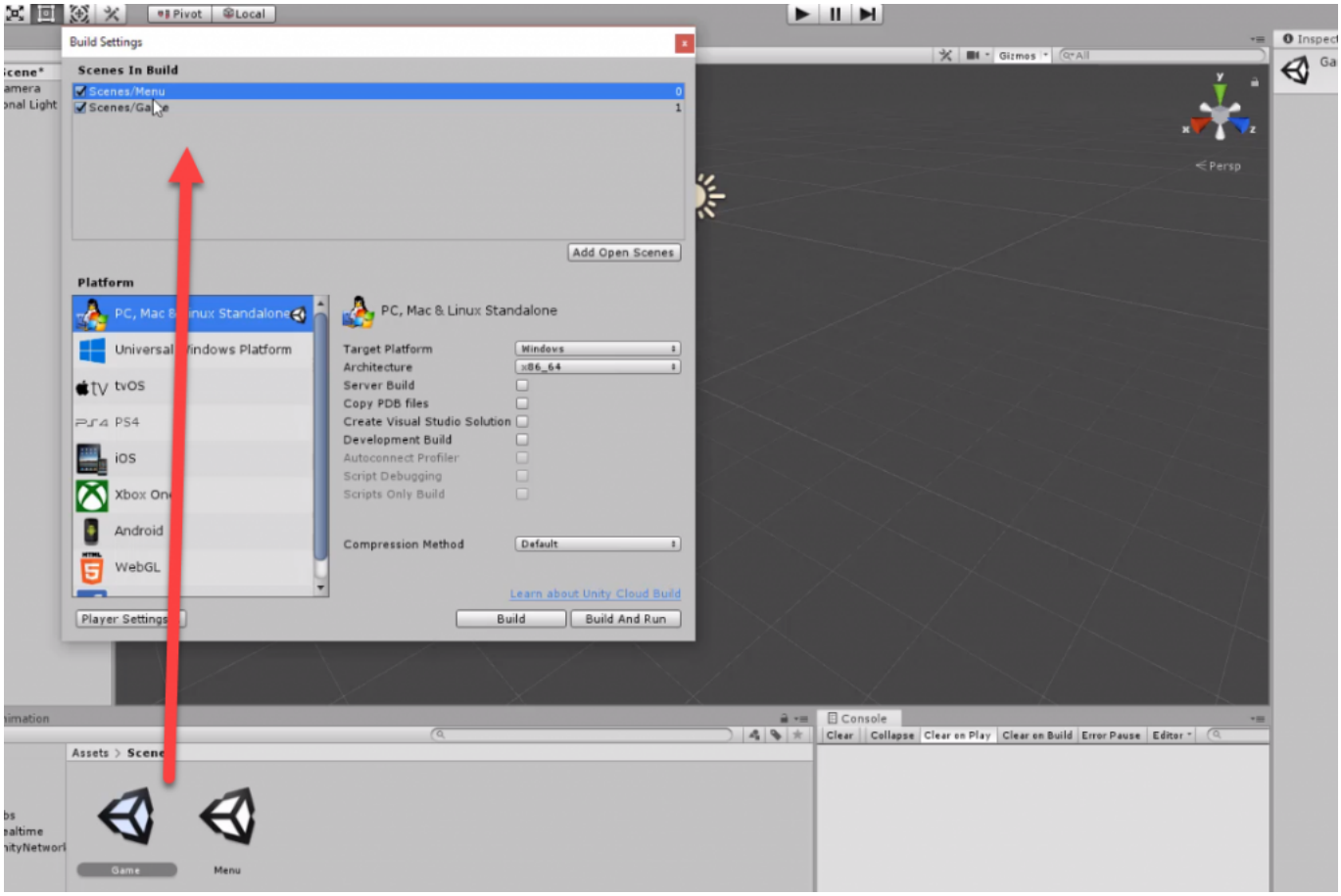
After you import the asset, the **PUN Wizard** window will popup. Here, we just want to paste in our app id, then click on **Setup Project**.



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

Setting up the Project

Before we begin, let's first set up our folders and scenes. In the **Scenes** folder, delete the default **SampleScene**. Then create 2 new scenes: **Menu** and **Game**. While we're at it, let's also add these to our **Build Settings** screen *(File > Build Settings)*.



Let's now create the rest of our folders. Create the following:

- Animations
- Prefabs
- Resources
- Scripts
- Sprites

All the sprite sheets are included in the **project files**. If you wish to download some other free sprite sheets though, you can do that here: https://kenney.nl/assets?q=2d

Scene Setup

Let's now go to our **game** scene and select the camera. We need to change it from a 3D camera, into a 2D one.

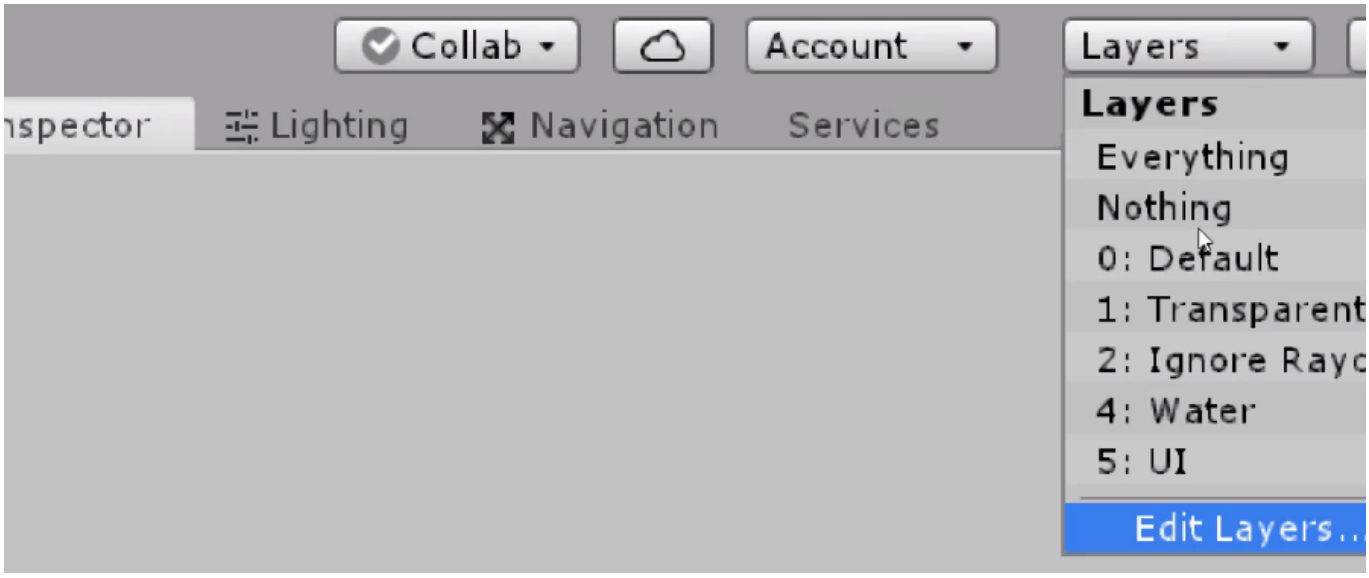
- Set the **Projection** to *Orthographic*
- Set the **Clear Flags** to *Solid Color*

Now we can go to the **menu** scene and do the same thing. This time though, let's change the background color.

- Set the **Background** color to *green/blue*

Tags and Sorting Layers

At the top right of the screen, select the **Layers** drop down and click on **Edit Layers…**



Create the following tags and sorting layers:

Tags

- Enemy

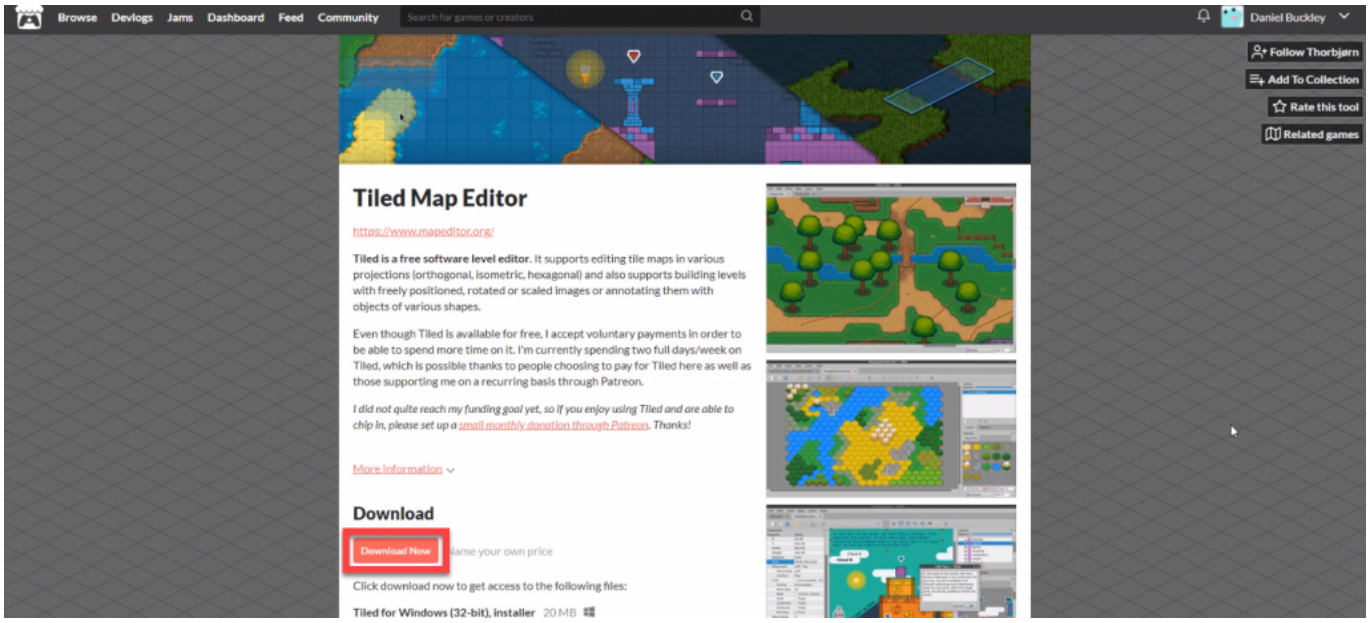
Sorting Layers

- BG
- Pickup
- Enemy
- Player
- UI

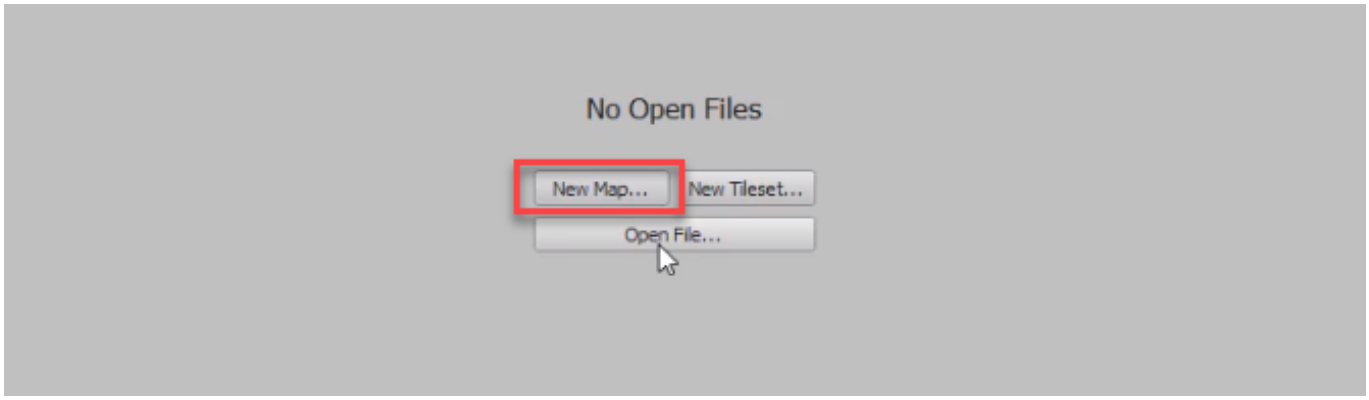
In the next lesson we'll used **Tiled** to create our tilemap and import it into Unity.

Tiled

Tiled is a free software which allows us to easily create tilemaps. Go to https://thorbjorn.itch.io/tiled and download the **Tiled Map Editor**.



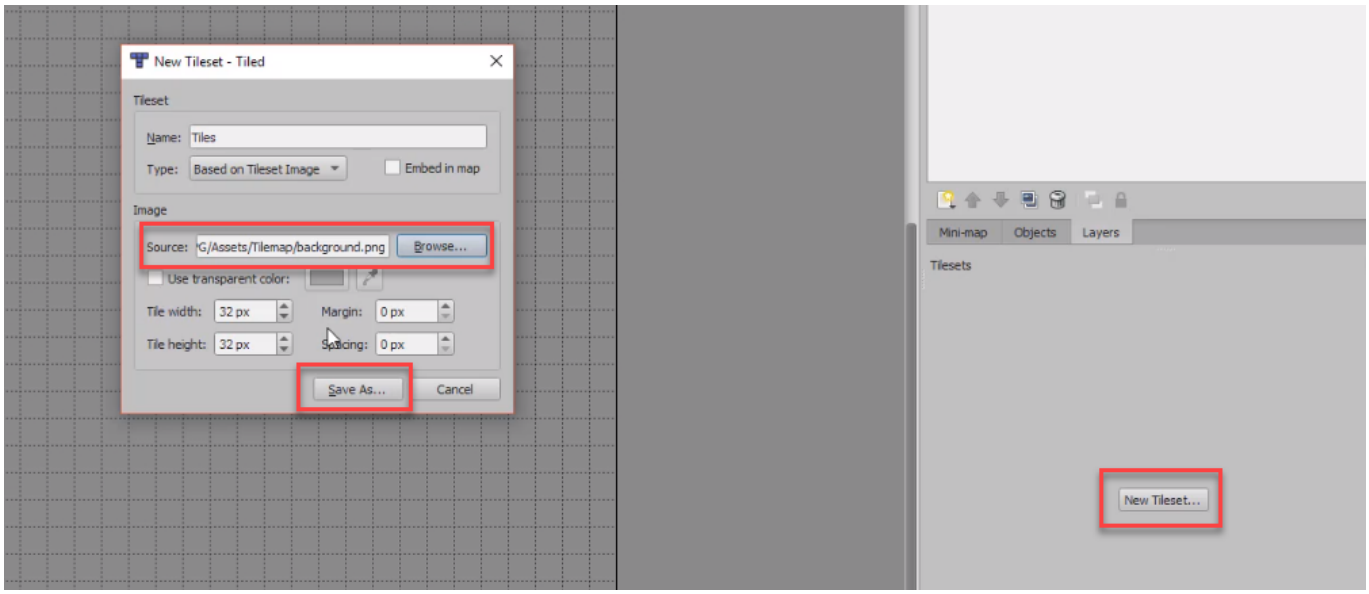
Once you download and install it, you should be able to launch the program. Select **New Map...**



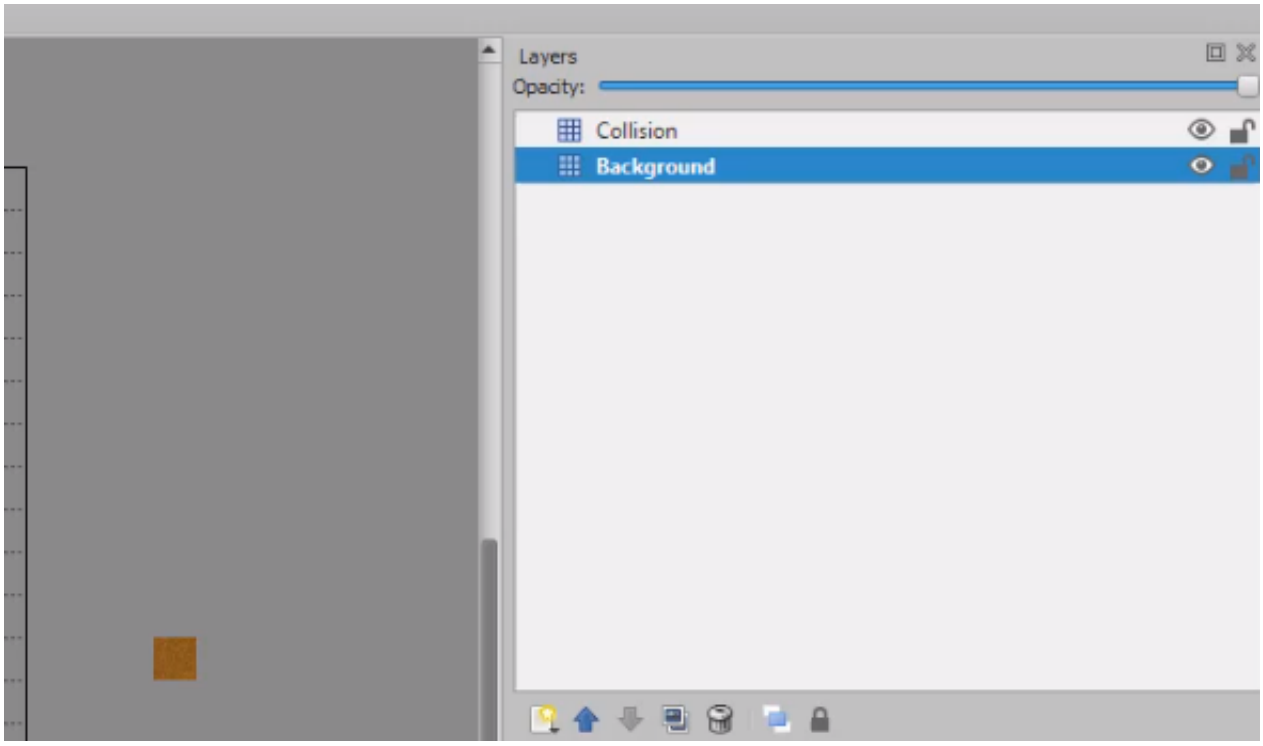
Set the **Width** and **Height** to *30*, then click **Save As...,** create a new folder in the project called **Tilemap** and save it there

When in the editor, we need to import a tile set. Click on the **New Tileset...** button.

- Set the **Width** and **Height** to *32*
- Set the **Source** to the background image in the project



We now need to create 2 layers. One for the **Background** (tiles that we can walk on) and one for **Collision** (tiles that we collide with).



Now just create any sort of map you wish.

Now export both the map and tileset into the **Tilemap** folder.

SuperTiled2Unity

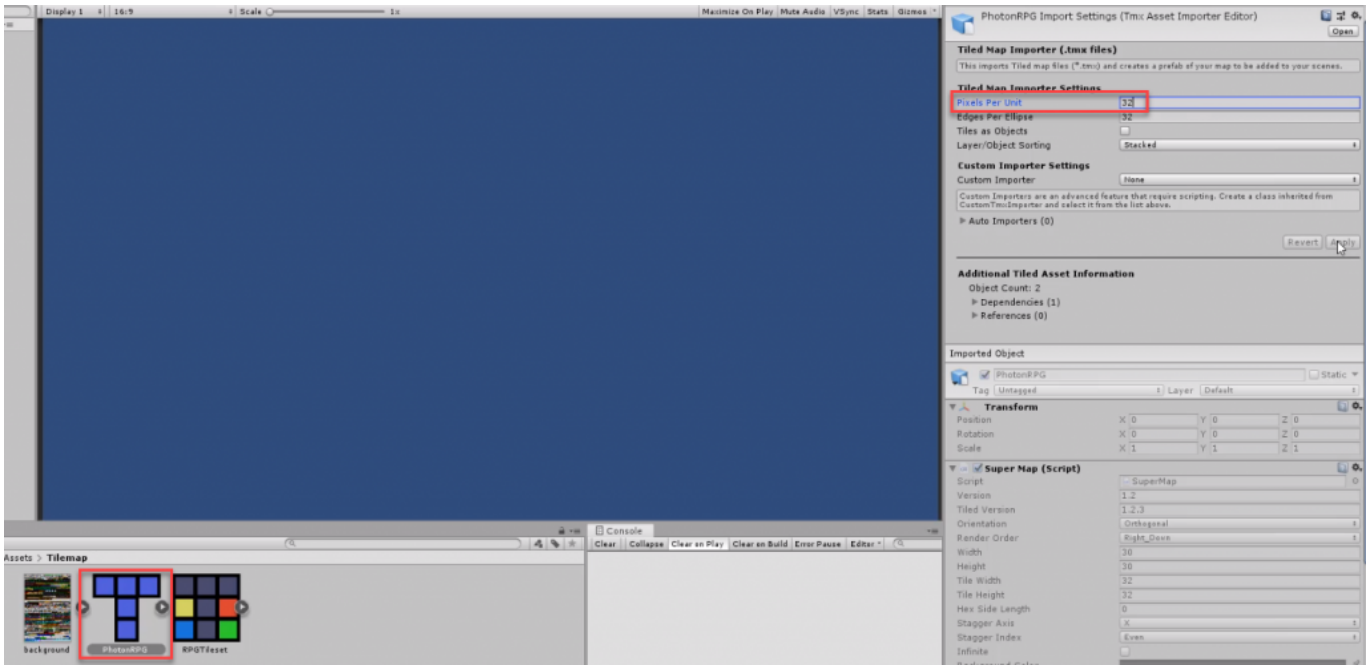
SuperTiled2Unity is a free Unity asset which converts Tiled maps into the tilemap format Unity uses. Download it from https://seanba.itch.io/supertiled2unity

This will download a **.unitypackage** file which you can just drag into the project window and import the asset.

Select our .tmx tilemap and:

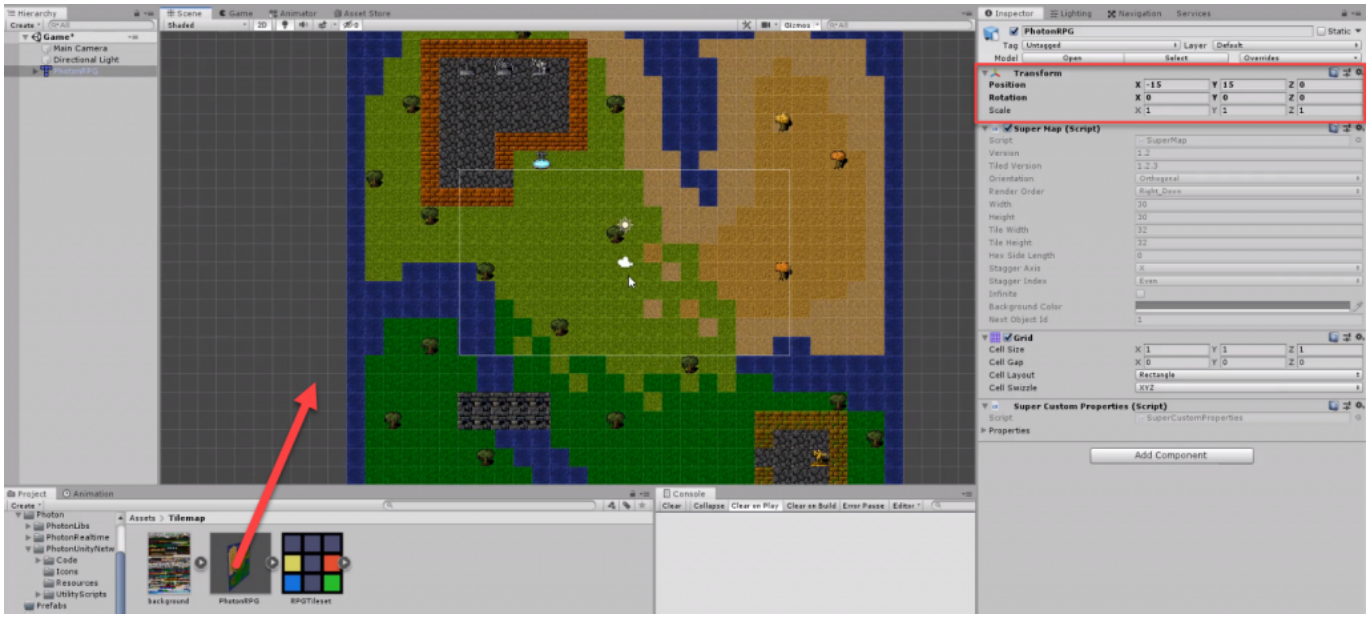
- Set **Pixels Per Unit** to *32*
- Enable **Tiles as Objects** (not see in image)

Click **Apply**. Do the same for the tileset.

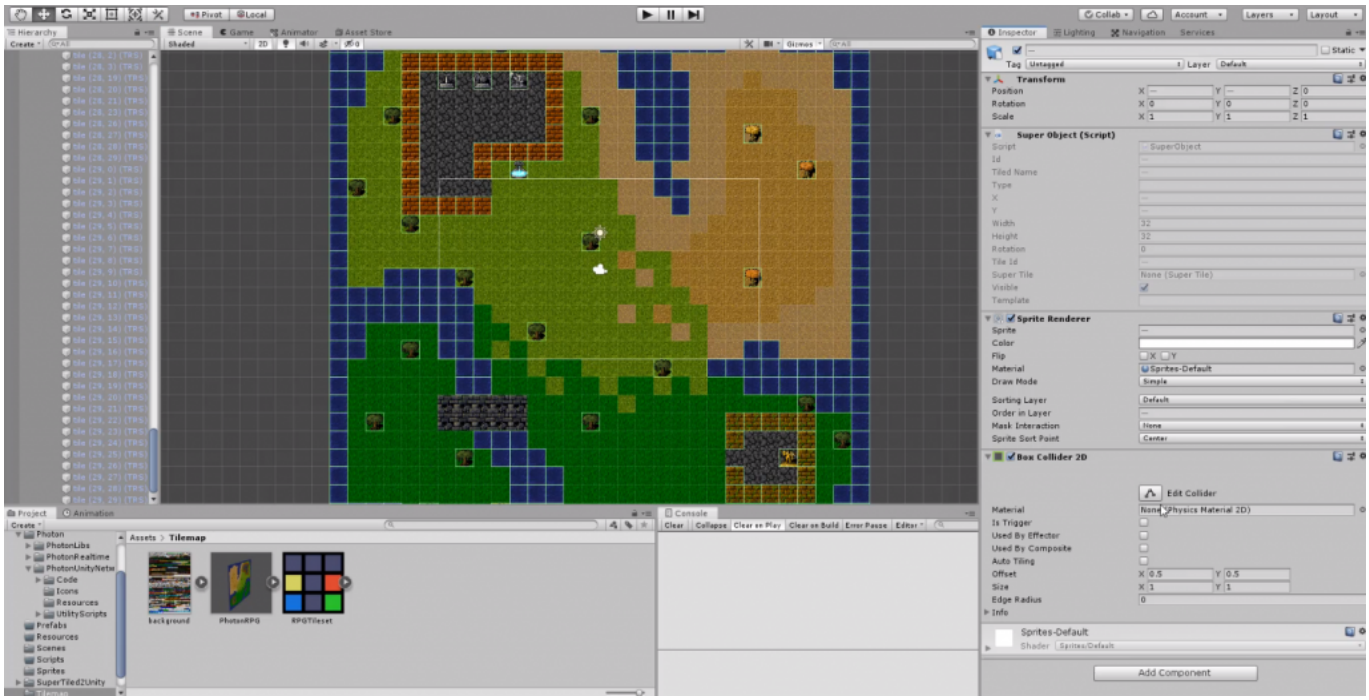


Now we can drag in the .tmx tilemap file into the scene.

- Set the **Position** to *-15, 15, 0*



Next, select all of the *collision* tiles and add a **Box Collider 2D** component to them.



NetworkManager Script

Create a new C# script called **NetworkManager** and attach it to a new GameObject called **_NetworkManager**. Open the script up in Visual Studio.

First, we need to add 2 new namespaces. These are required in order to use Photon.

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

Then we can move onto our only variable, which just keeps track of the max number of players allowed in a game.

```
public int maxPlayers;
```

Let's also create an instance (or singleton). This will allow us to access the script from anywhere in the project easily without needing to get the component, etc.

```
// instance
public static NetworkManager instance;

void Awake ()
{
    if(instance != null && instance != this)
        gameObject.SetActive(false);
    else
    {
        instance = this;
        DontDestroyOnLoad(gameObject);
    }
}
```

In the **Start** function, we'll connect to Photon.

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

In order to know when we connect to the server, let's implement a callback function. In order to do that though, we need to change our inheriting class.

```
public class NetworkManager : MonoBehaviourPunCallbacks
```


Now we can add the **OnConnectedToMaster** callback function and then join the lobby.

```
public override void OnConnectedToMaster ()
{
    PhotonNetwork.JoinLobby();
}
```

The **CreateRoom** function gets called when we want to create a new room.

```
public void CreateRoom (string roomName)
{
    RoomOptions options = new RoomOptions();
    options.MaxPlayers = (byte)maxPlayers;

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


NetworkManager Script

The **JoinRoom** function gets called when we want to join an existing room;

```
public void JoinRoom (string roomName)
{
    PhotonNetwork.JoinRoom(roomName);
}
```

The **ChangeScene** function gets called when we want to change the current scene to another one using Photon's system.

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


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.

The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected code.

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.

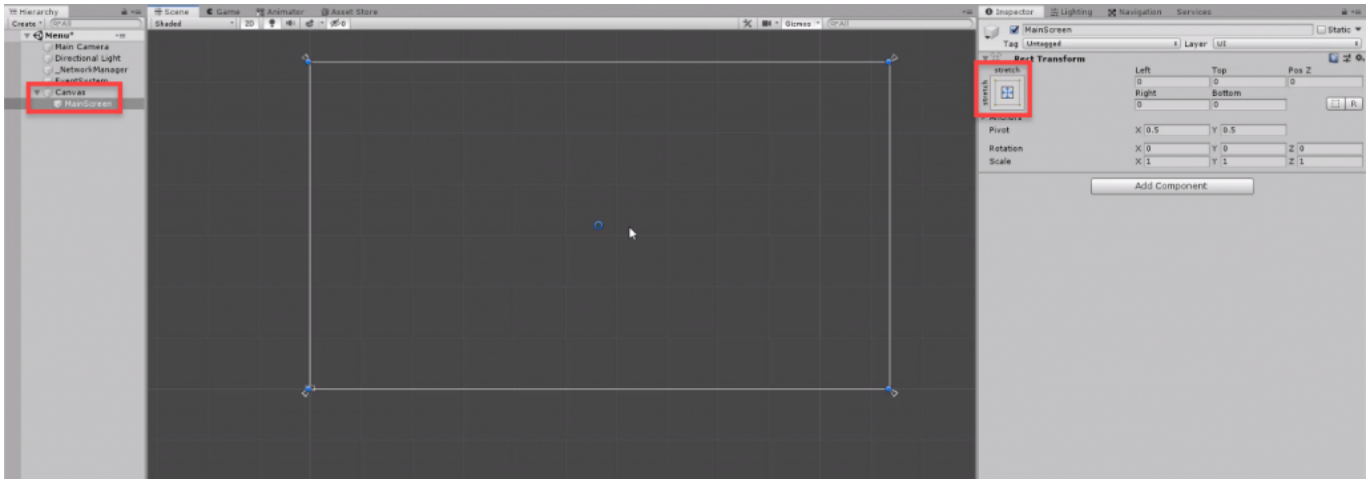
The following instructions have been updated, and differs from the video:

Before creating the Menu UI you need to create the _NetworkManager object the same way you created it for the Game Scene.

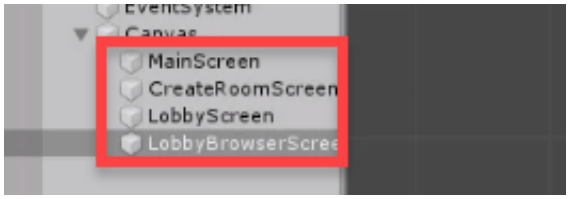
Creating the Menu UI

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

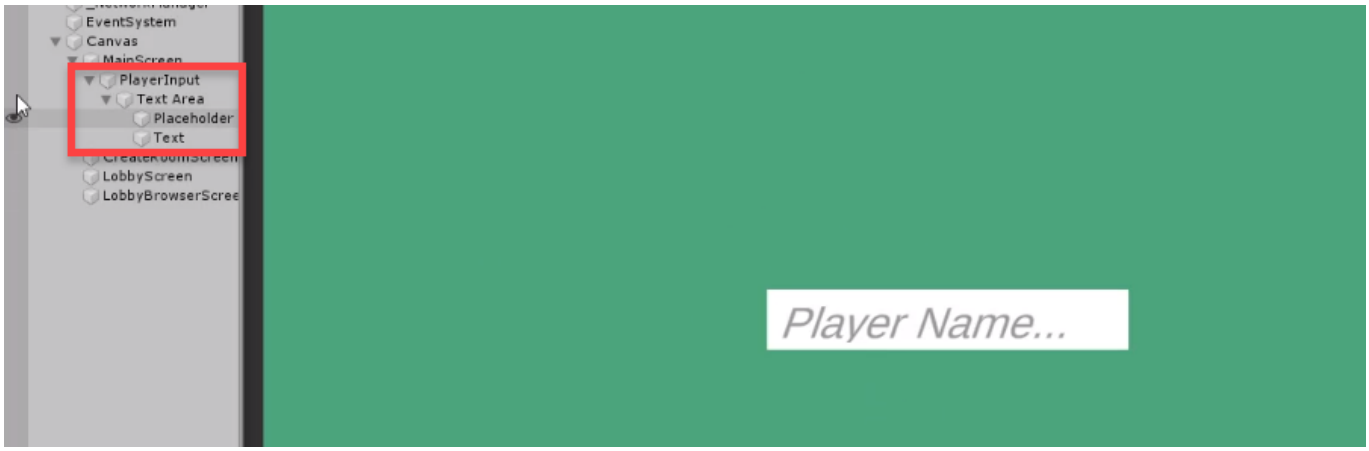


Next, duplicate the object 3 times, for the **CreateRoomScreen**, **LobbyScreen** and **LobbyBrowserScreen**.



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*

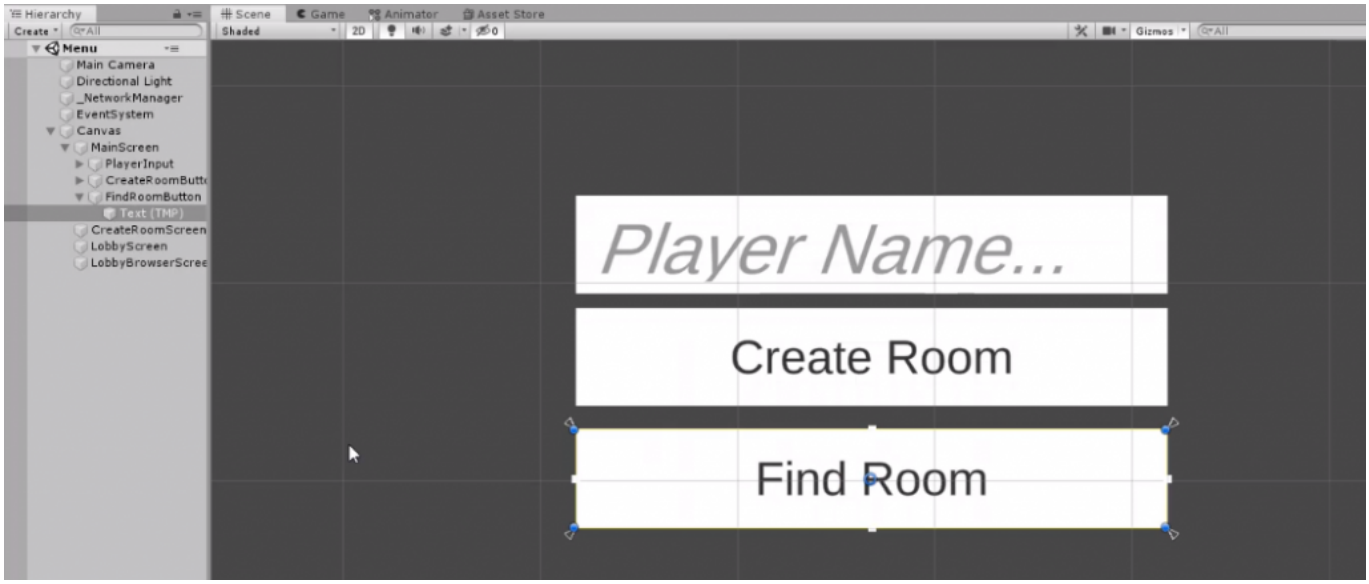


Create a new **Button – Text Mesh Pro** and call it **CreateRoomButton**. Position it underneath the player name input.

- 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*

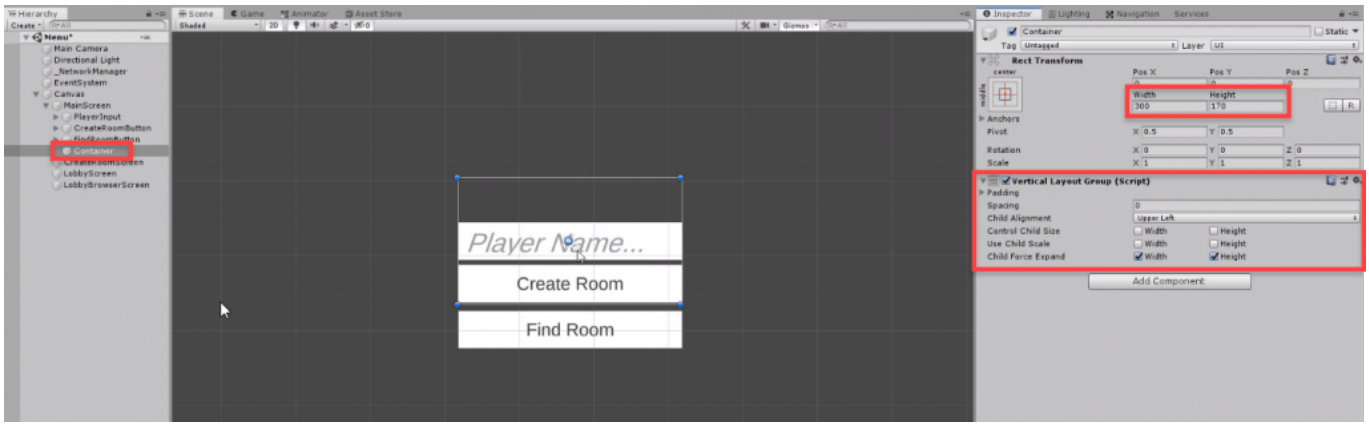


Layout Groups

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.

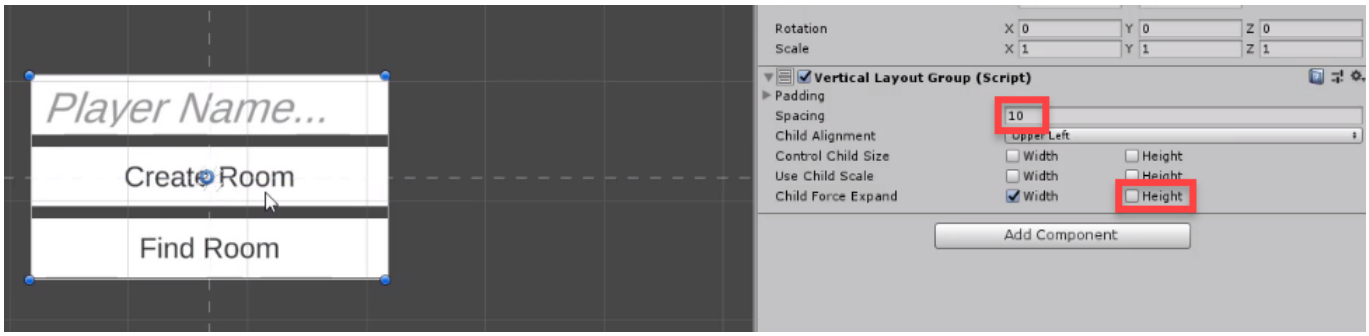
- Set the **Width** to *300*
- Set the **Height** to *170*



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.

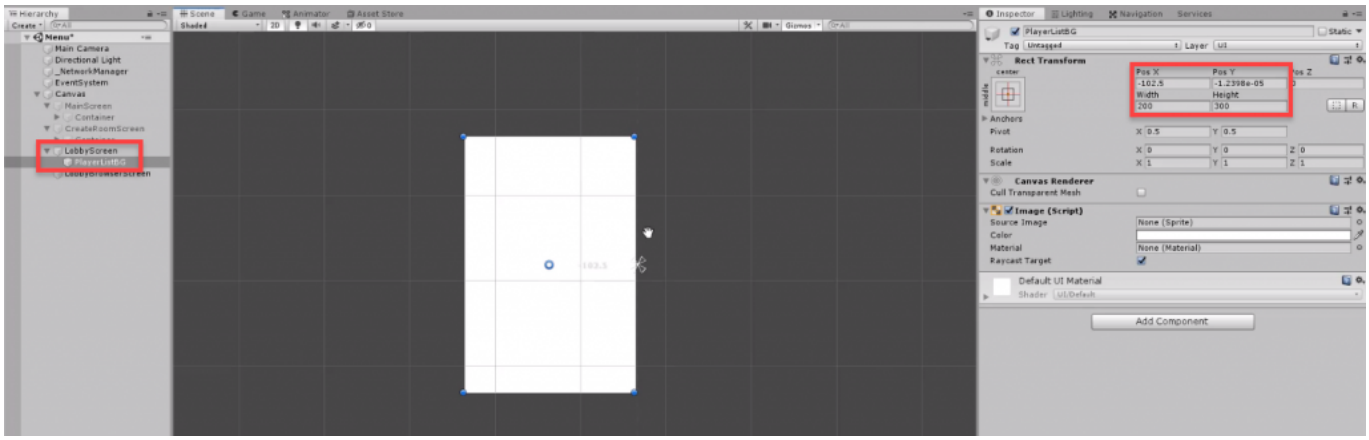
- Change the player name input to **Room Name** input
- Change the create room button to **Create**
- Change the find room button to **Back**

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.

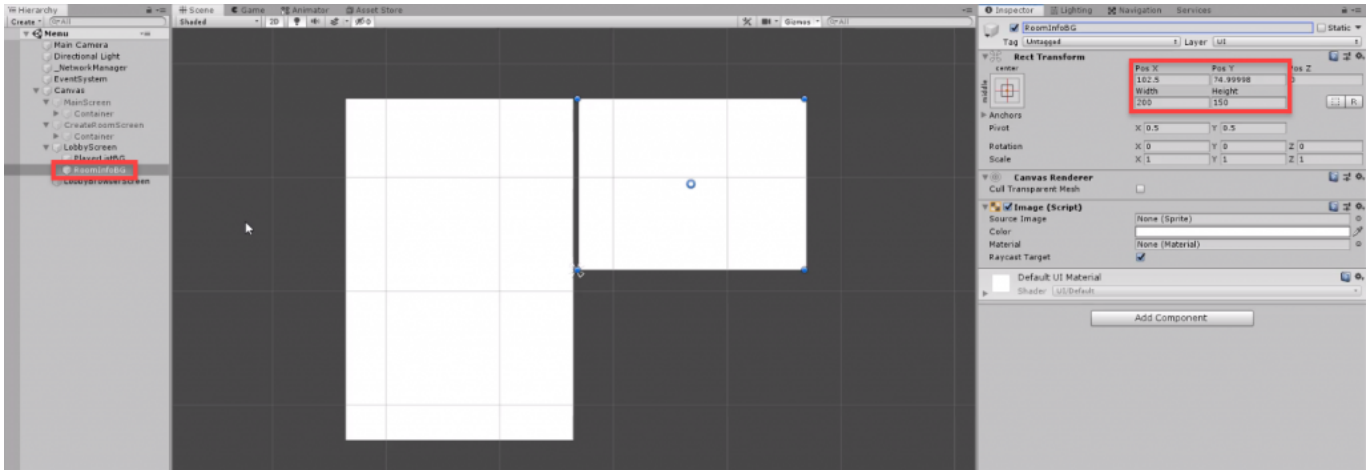
Create a new image object and call it **PlayerListBG**.

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



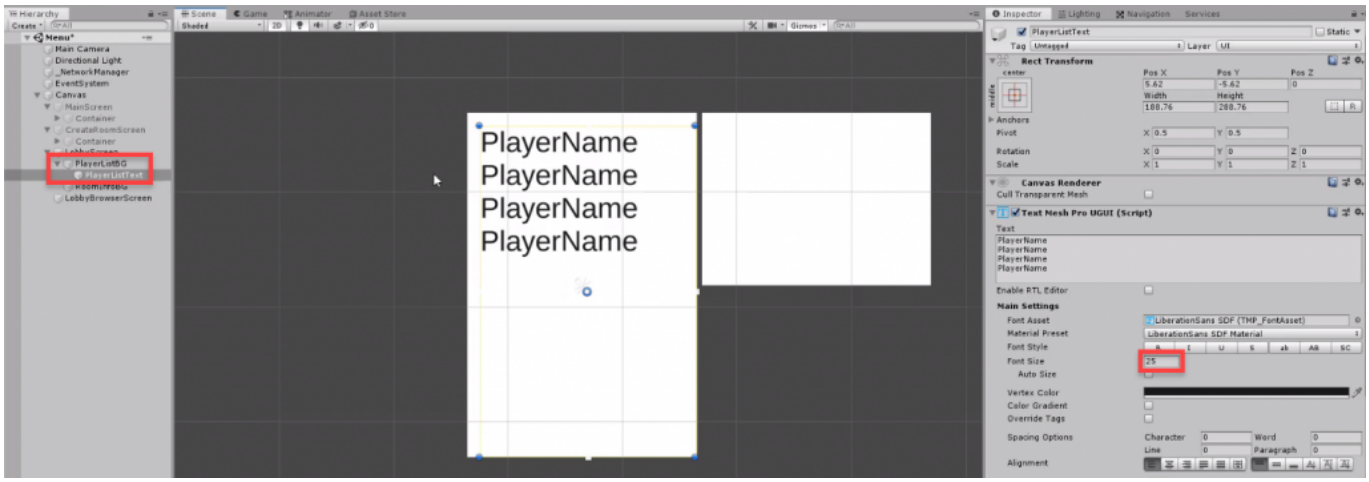
We can then duplicate the image and call it **RoomInfoBG**.

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

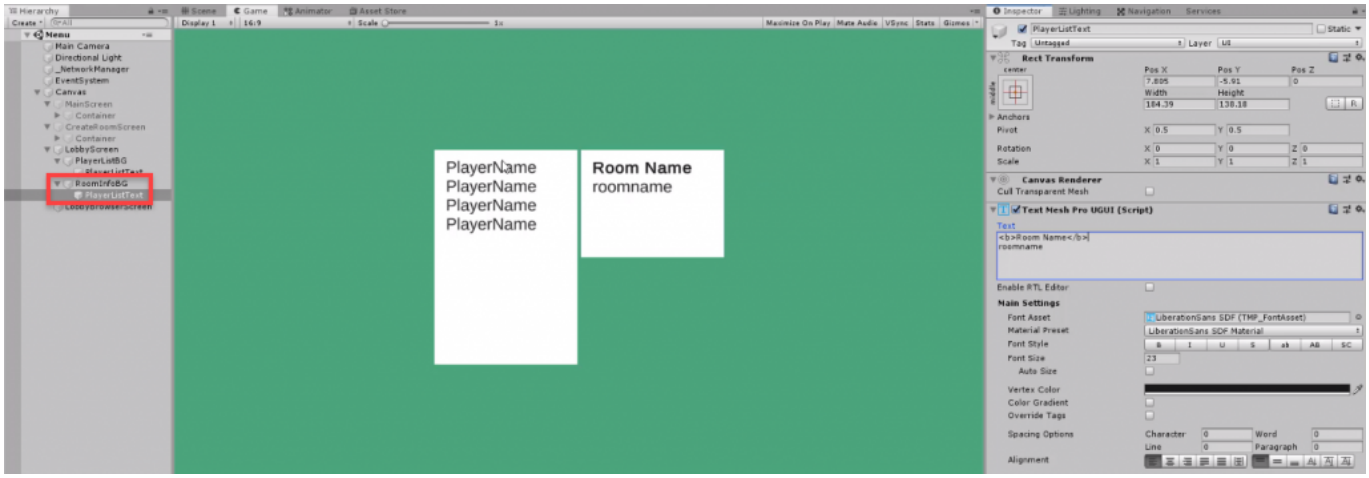


As a child of the player list BG, create a new **Text – Text Mesh Pro** object and call it **PlayerListText**.

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

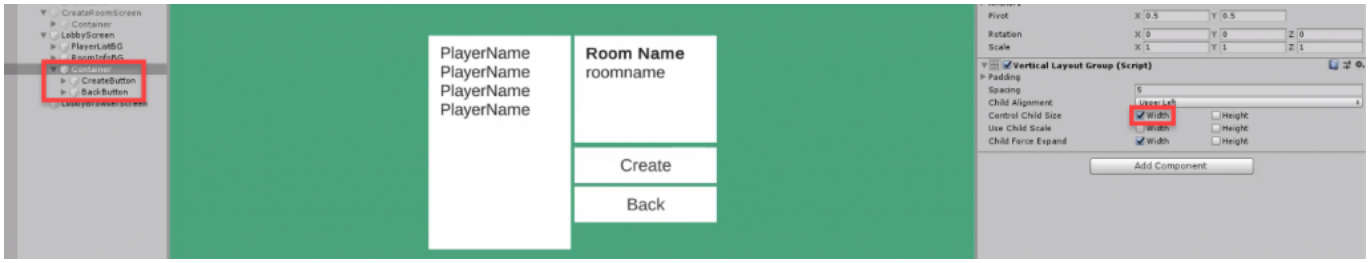


We can then duplicate the text and set it as a child of the room info BG. Call the text **RoomInfoText**.

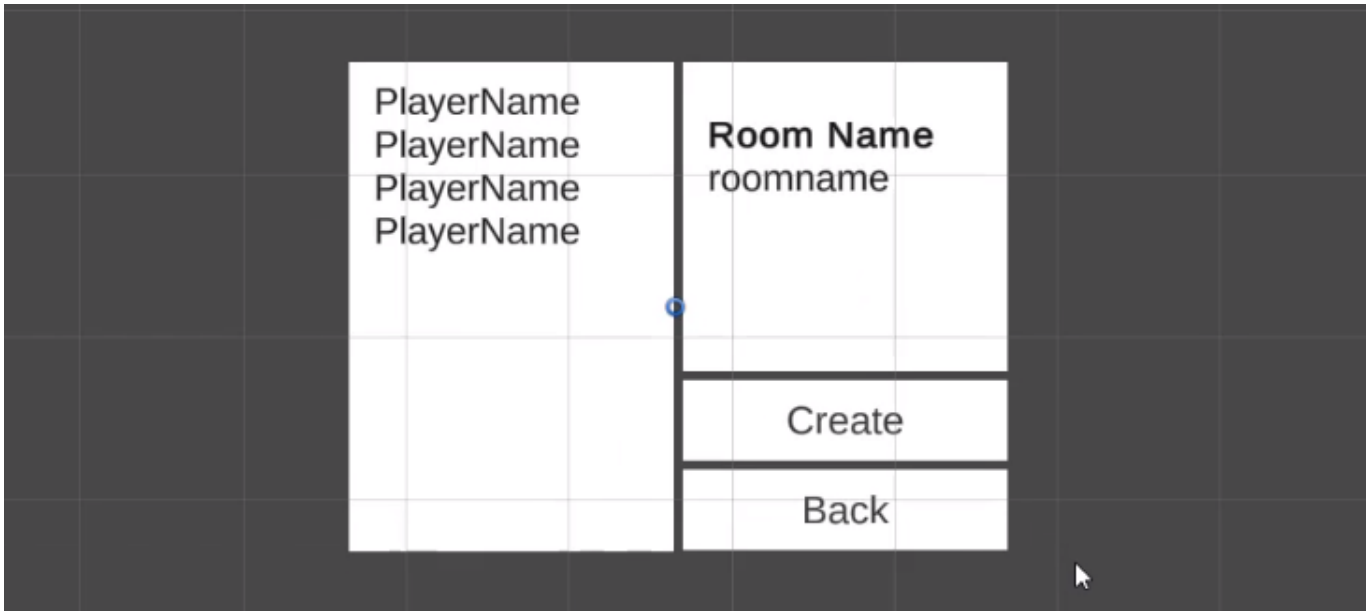


For the buttons, copy over the **Container** from either one of the previous pages.

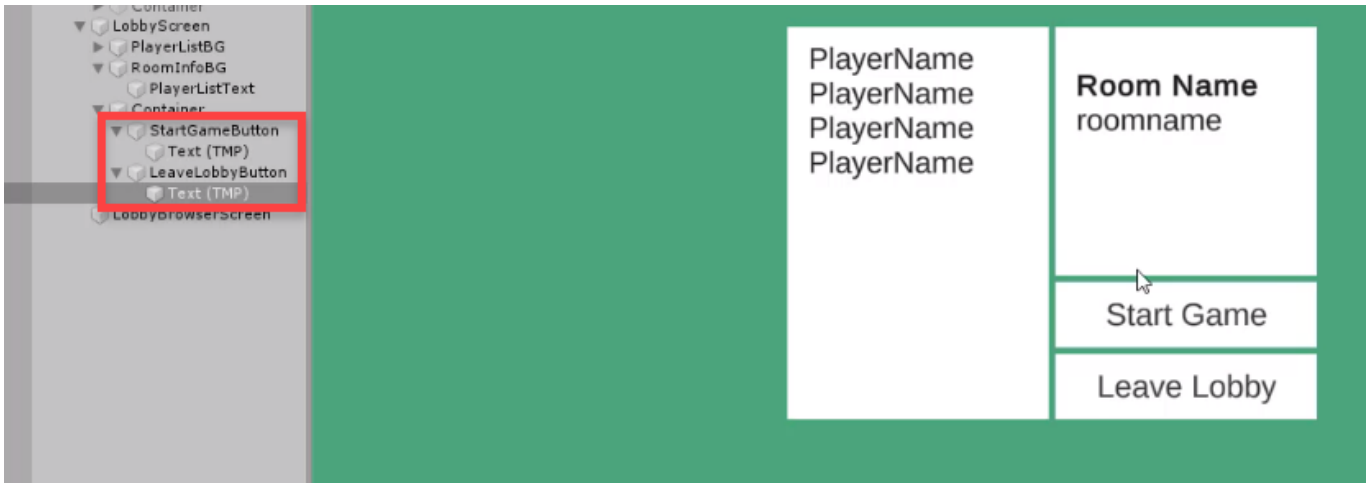
- Delete the **Input** field
- Enable **Control Child Size – Width**



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



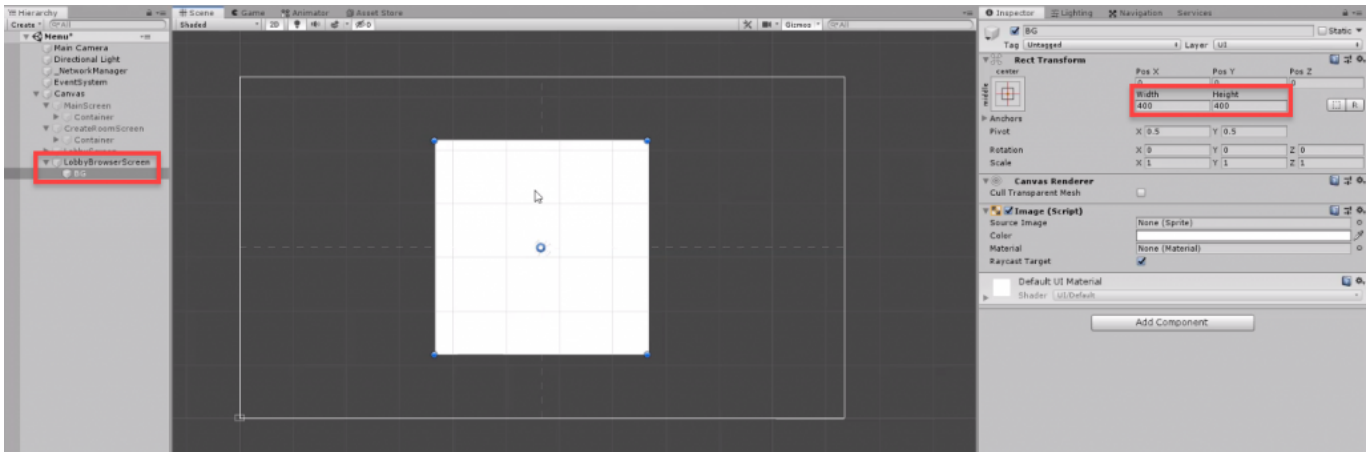
Set the top button to **Start Game** and the bottom button to **Leave Lobby**.



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*

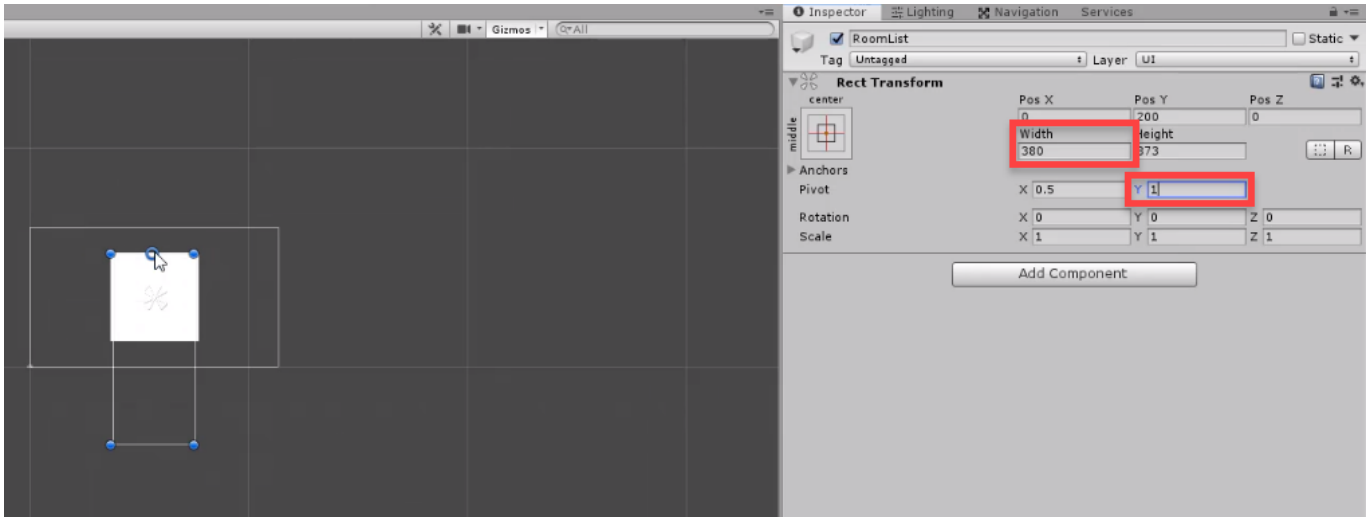


Next create an empty object as a child of BG called **RoomList**. We'll be working on this more next lesson as it's to do with setting up our scroll rect.

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.

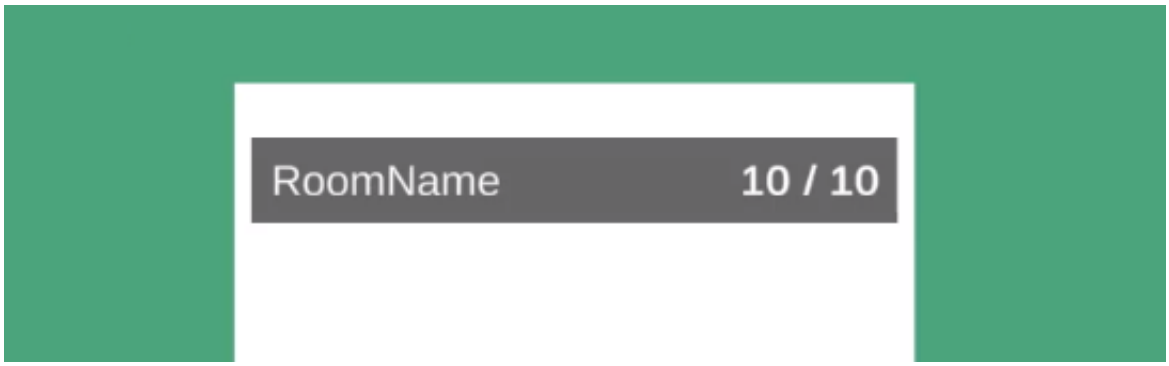


As a child of the **RoomList**, create a new image object and call it **RoomButton**.

- 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



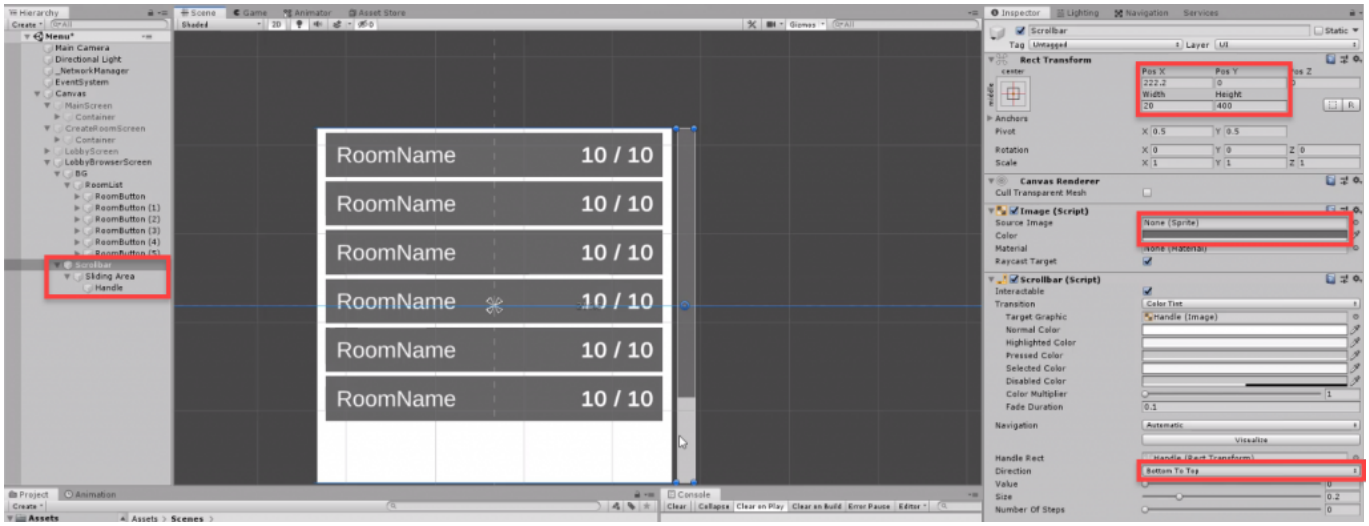
On the room list container, add a **Vertical Layout Group** component.

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

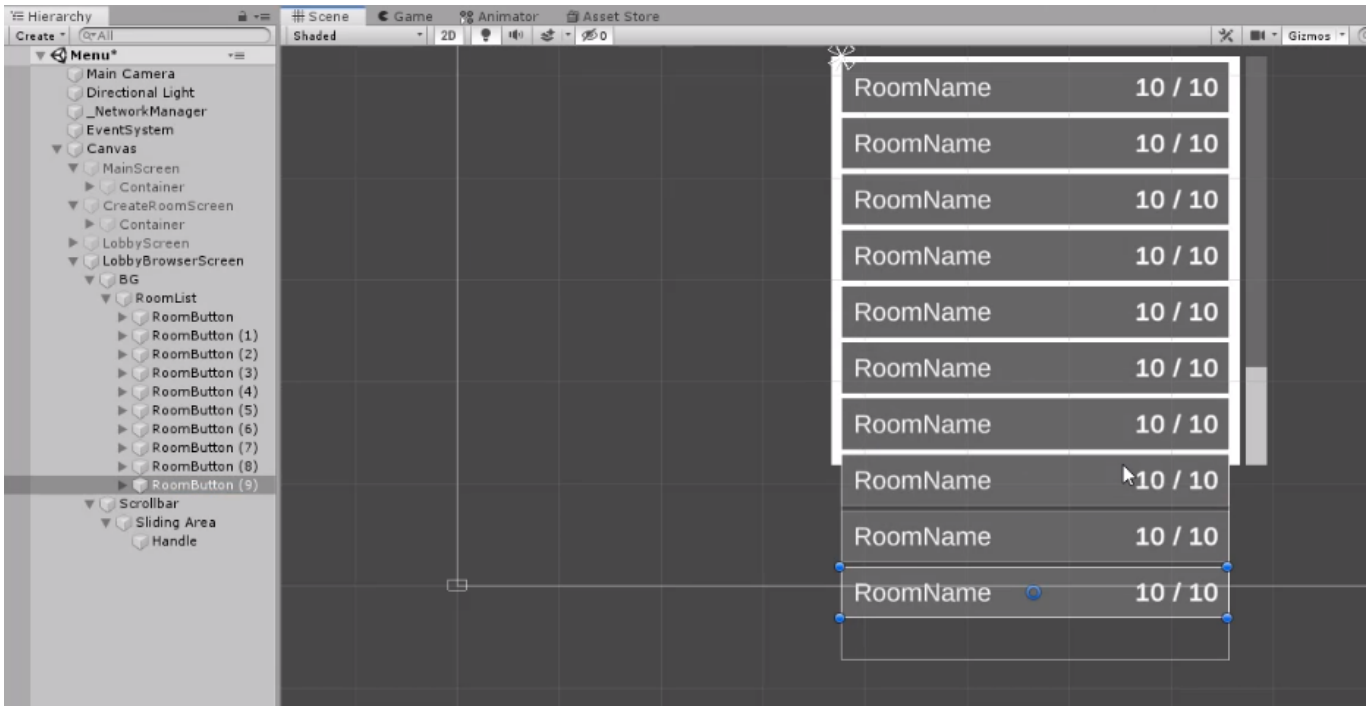
Now we can duplicate the room button and see that they stack up nicely.

Since this is going to be a scroll window, we also need a scroll bar. As a child of the BG, 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*

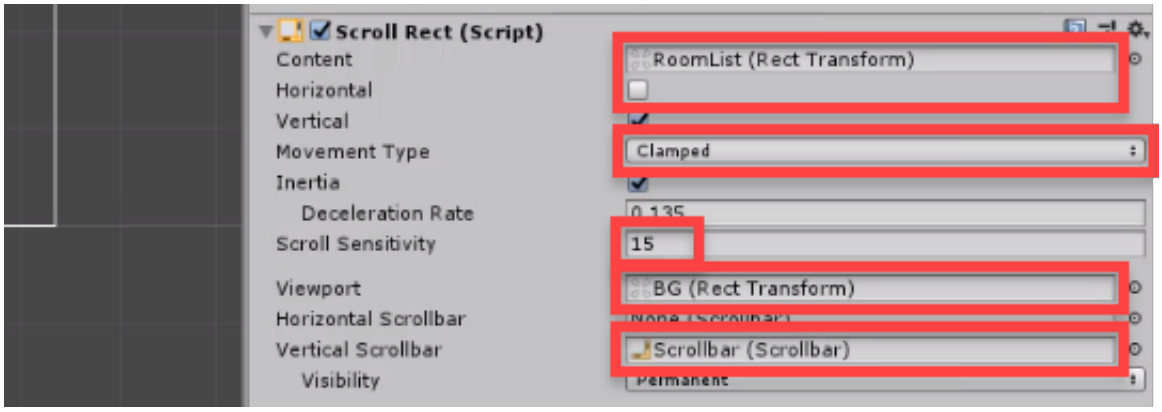


Now let's get the actual scroll window working. First though, we'll duplicate the room button some more. You'll see that they go over the white background and spill downwards. A scroll rect will mask that and allow us to scroll vertically along the room list's height.



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

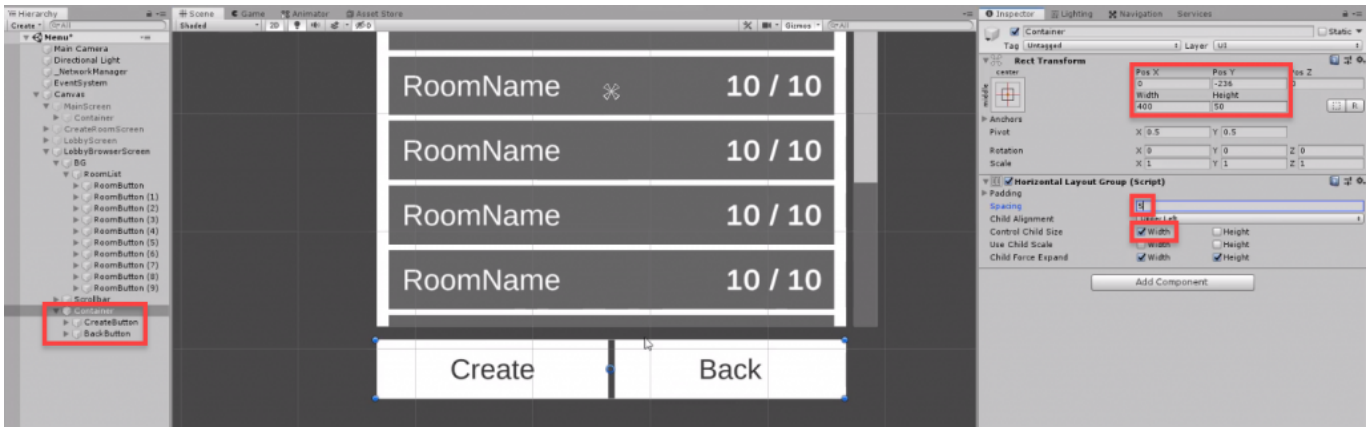


If you press play, you should be able to scroll through the list and see that the BG masks it.

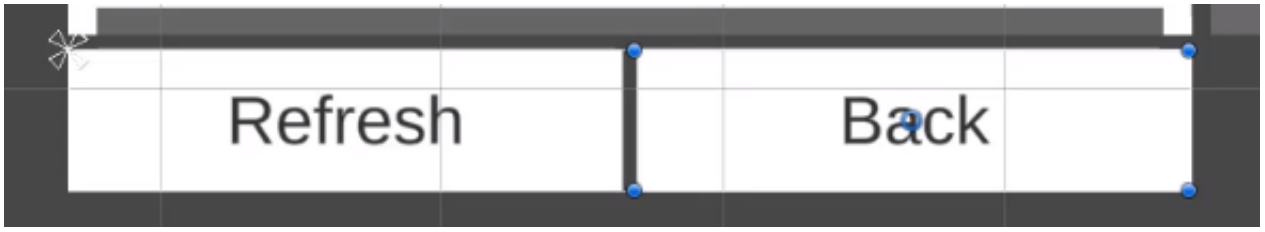
For the buttons, just copy over the previous button's we've used but replace the **Vertical Layout Group** with a **Horizontal Layout Group**.

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

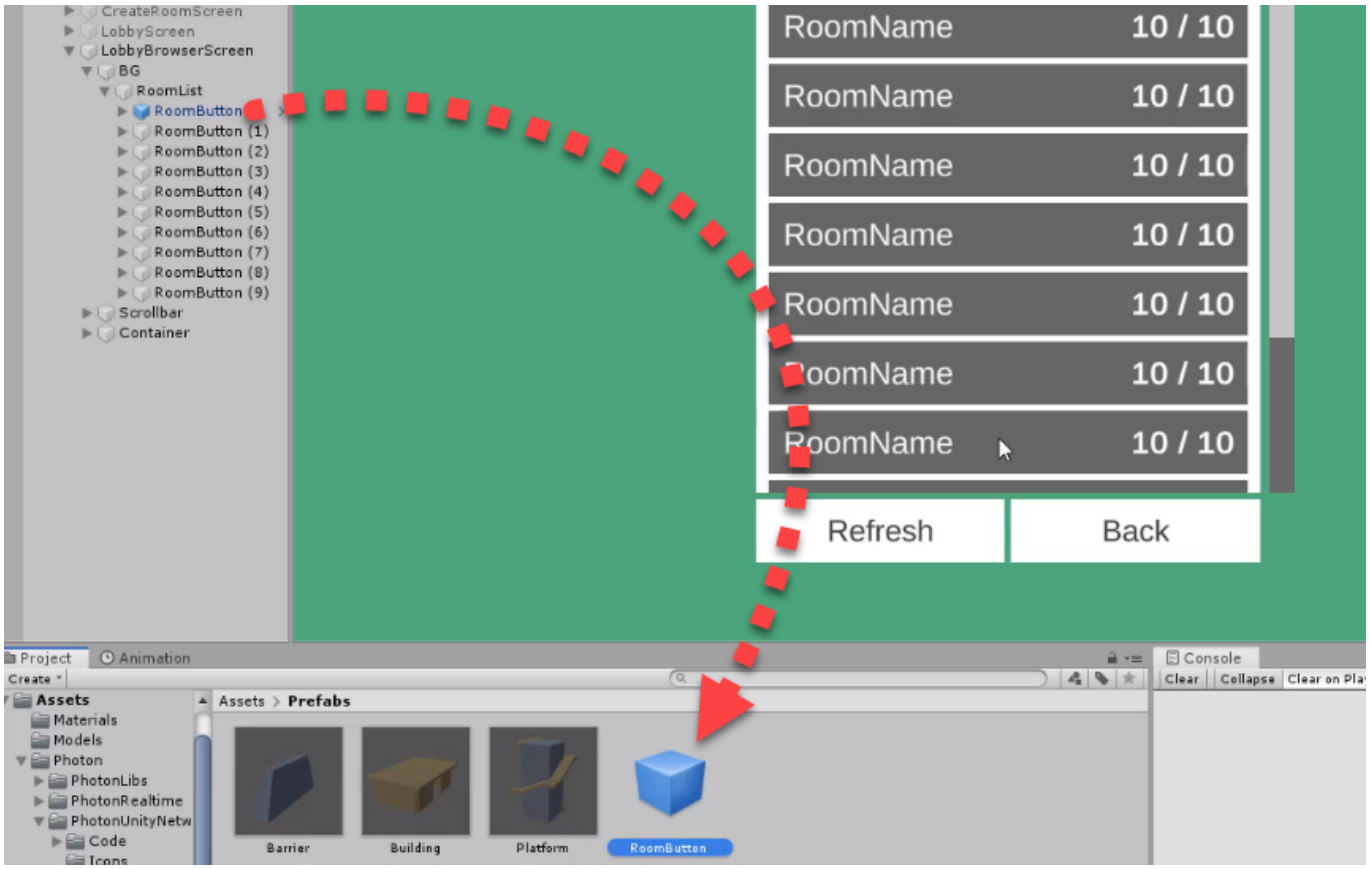
- Set the **Height** to *50*
- Set the **Spacing** to *5*
- Enable **Control Child Size – Width**



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.

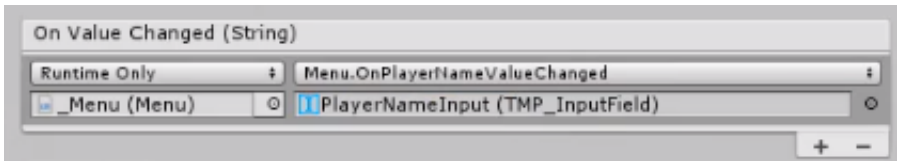
```
// called when the "Create Room" button has been pressed.
public void OnCreateRoomButton ()
{
    SetScreen(createRoomScreen);
}
```

Then for the **Find Room** button.

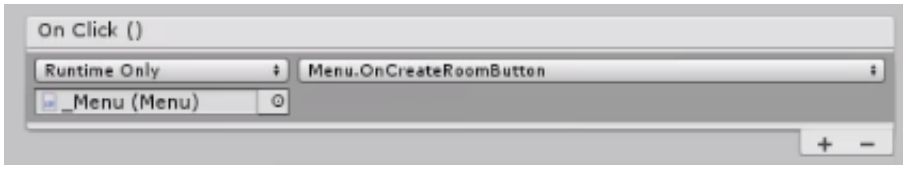
```
// called when the "Find Room" button has been pressed
public void OnFindRoomButton ()
{
    SetScreen(lobbyBrowserScreen);
}
```

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

The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected code.

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);
}
```

The following instructions have been updated, and differs from the video:

In order to change between Scenes, you need to open File->Build Settings and add both Scenes in the “Scenes in Build” area. You can just drag and drop the Scenes there.

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();
```


GameManager Script

Create a new C# script called **GameManager**. This will be where we spawn in the players. In the Hierarchy, create a new empty GameObject called **_GameManager** and attach the script to it. Also add a **PhotonView** component, since we will be receiving RPC's.

- Set the **View ID** to *2*

Now open up the script in Visual Studio so we can begin. First, we need to add our namespaces.

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

Since we want to access our PhotonView component, let's change our inheriting class.

```
public class GameManager : MonoBehaviourPun
```

Next, let's create our variables.

```
[Header("Players")]
public string playerPrefabPath;
public Transform[] spawnPoints;
public float respawnTime;

private int playersInGame;

// instance
public static GameManager instance;

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

In order to spawn in the players, we need to know when everyone has joined the Game scene. Let's create an RPC function which will be called on all the player's computer when someone joins the scene.

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

    if(playersInGame == PhotonNetwork.PlayerList.Length)
        SpawnPlayer();
}
```


In the **Start** function, we can call the **ImInGame** function.

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

Then, the **SpawnPlayer** function will be called for each player, spawning in their player.

```
void SpawnPlayer ()
{
    GameObject playerObj = PhotonNetwork.Instantiate(playerPrefabPath, spawnPoints[Ra
ndom.Range(0, spawnPoints.Length)].position, Quaternion.identity);

    // initialize the player
}
```

Spawn Points

Back in the editor, let's create a few empty GameObject's as a child of the **_GameManager** for the spawn points. Put them all in the house or wherever you want. Then drag them into the **GameManager**'s spawn points array.

Creating the Player

To create the player, let's start by creating a new sprite object (right click Hierarchy > *2D Object* > *Sprite*) and call it **Player**.

- Set the **Sprite** to one of the character sprites
- Set the **Sorting Layer** to *Player*

Now let's attach a few components to the player.

1. **Rigidbody2D**
2. **CapsuleCollider2D**

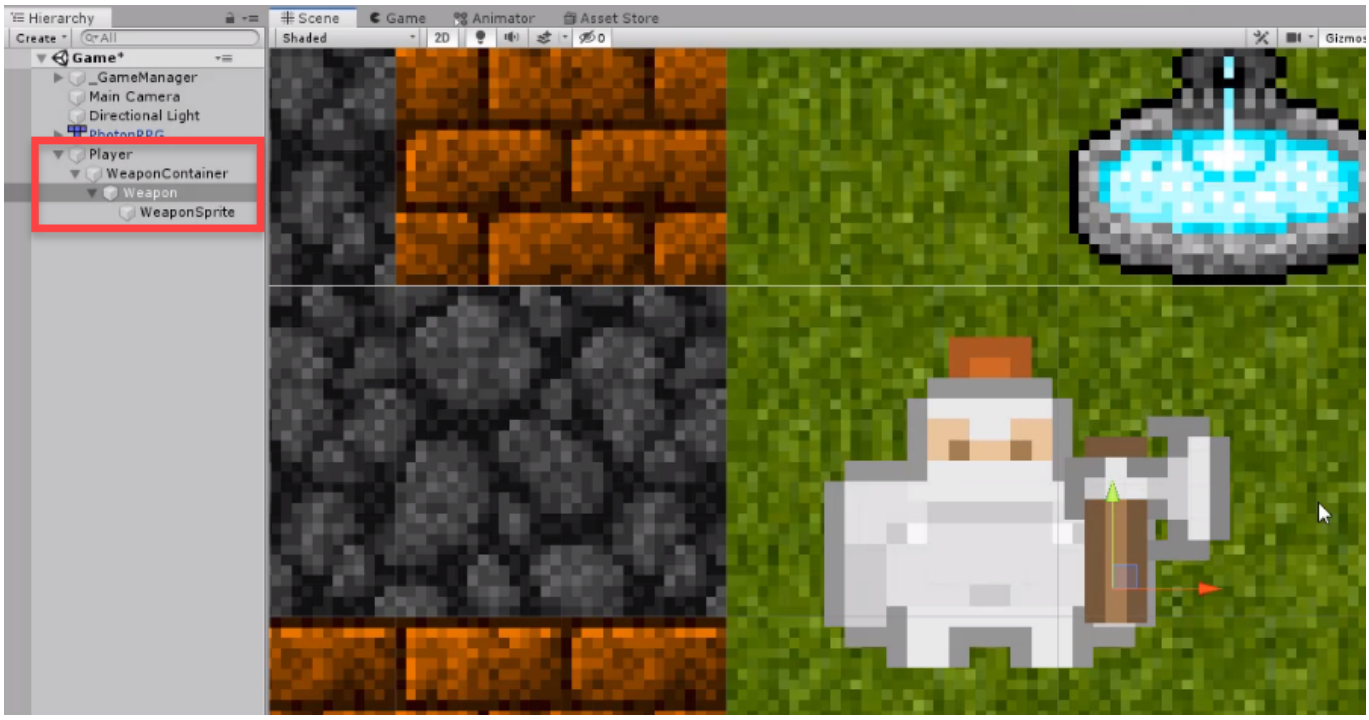
We need to tweak them just a little bit.

- Set **Gravity Scale** to *0*
- Enable **Constraints – Freeze Z Rotation**
- Set collider **Size** to *0.8, 0.8*
- Disable **Is Trigger**

Player Weapon

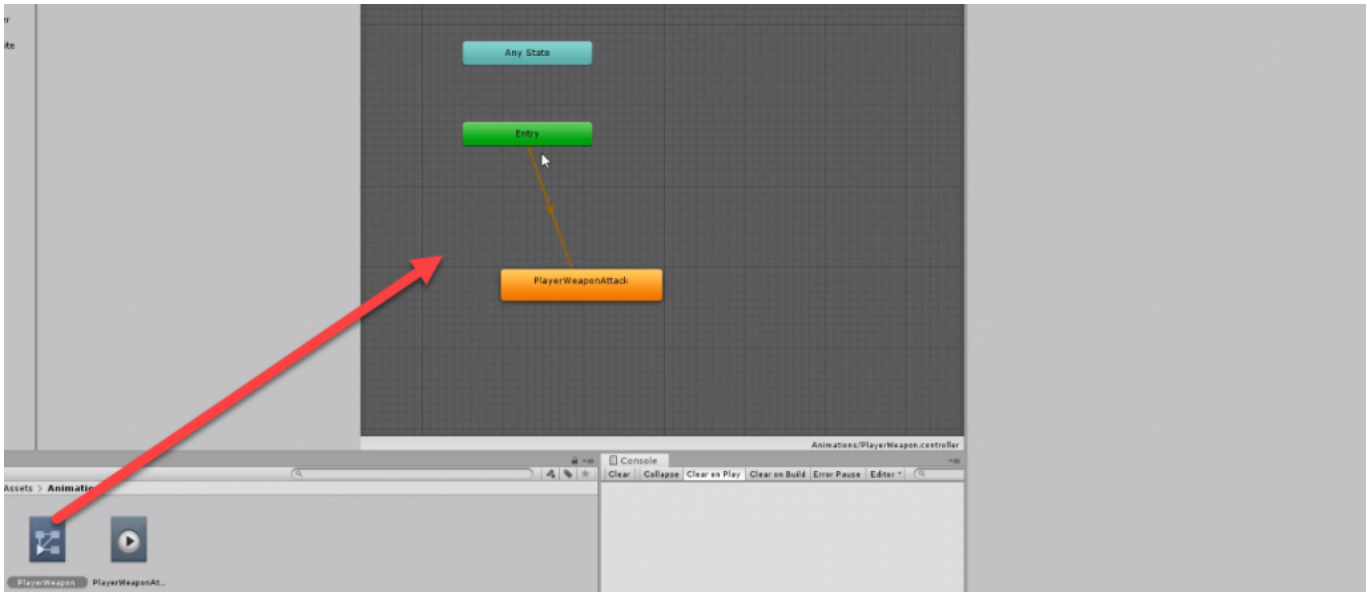
For the player weapon, let's create a new empty GameObject as the child of the player and call it **WeaponContainer**. Then, as a child of that, create a new empty GameObject called **Weapon**. Finally, as a child of that, create a new sprite object and call it **WeaponSprite**.

- Set the **Sprite** to a weapon
- Set the *Weapon* **Position** to *0.37, -0.26*
- Set the *WeaponSprite* **Position** to *0.1, 0.2*



We can not create our weapon animation. In the **Animations** folder, create a new **Animation Controller** called **PlayerWeapon**. Then create a new **Animation** called **PlayerWeaponAttack**.

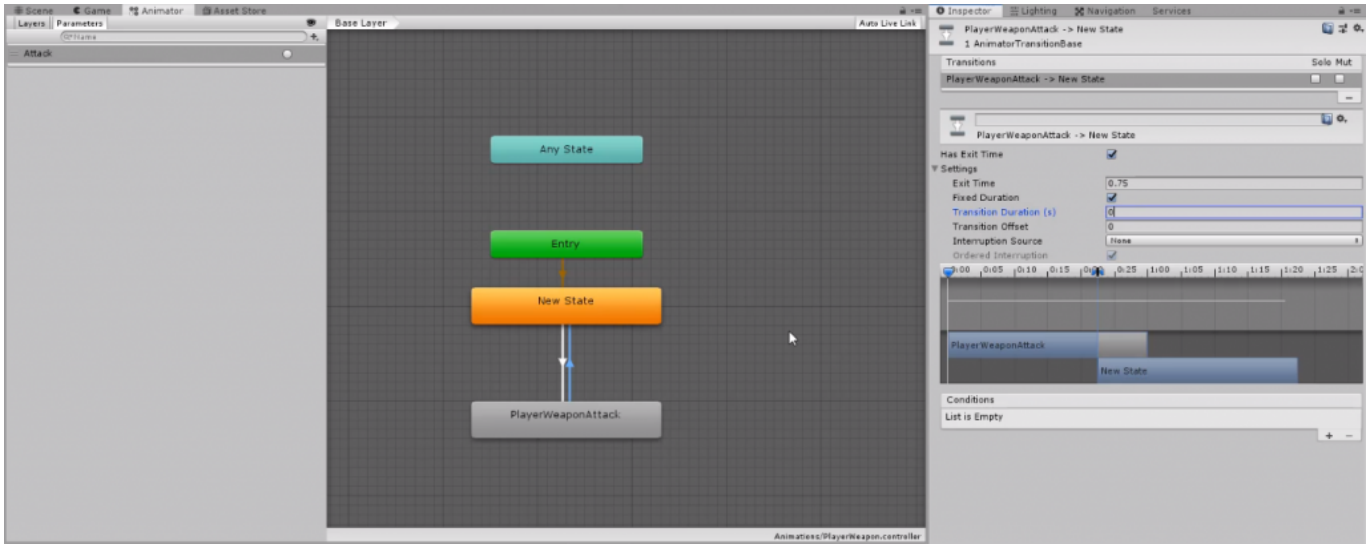
Drag the animation controller onto the **Weapon** object. Then double click the object and drag in the clip.



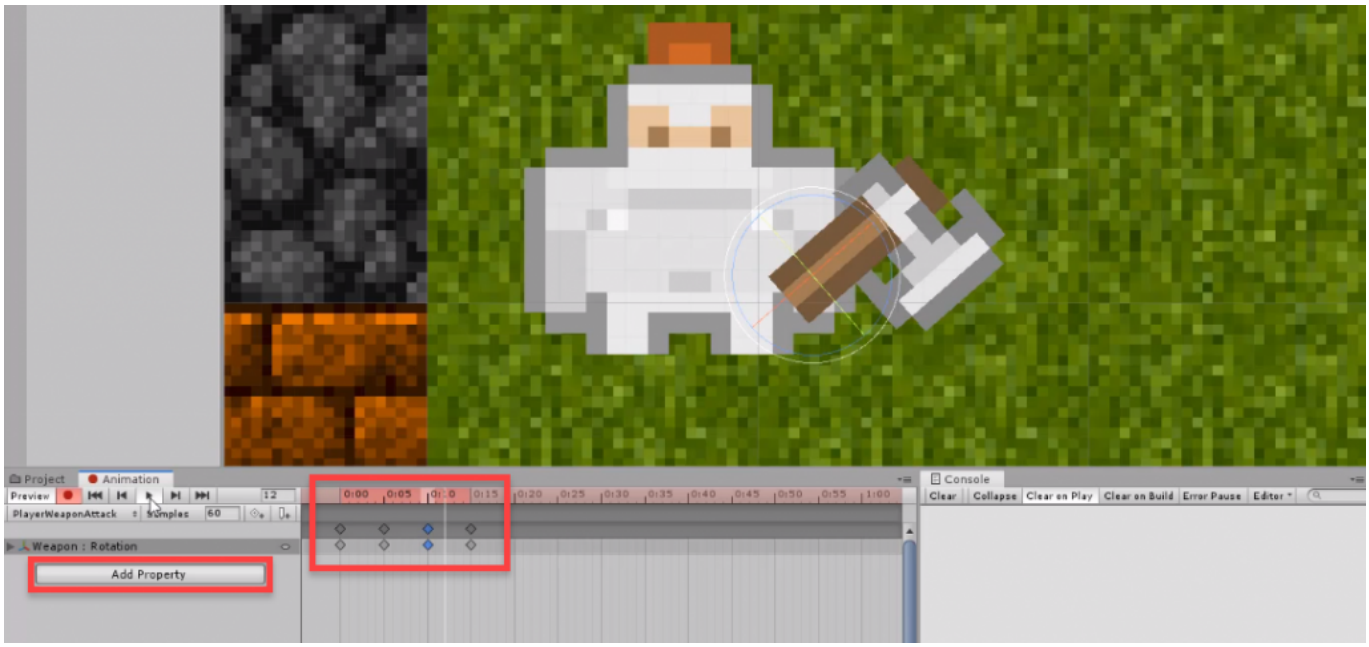
With the animator, we need to set it up so that when we call a trigger, the attack animation will play.

- Right click > *Create State* > *Empty*
- Right click the state and select *Set as Layer Default State*
- Click on the + icon on the side panel and create a new *Trigger* called **Attack**
- Create a transition between the **New State** and **PlayerWeaponAttack**, setting the condition to be the trigger
- Create a transition between the **PlayerWeaponAttack** and **New State**

- For both transitions:
    - Set **Transition Duration** to 0



We can now double click on the **PlayerWeaponAttack** animation clip to begin editing it. The only property we need, is the weapon's **Rotation**. Create a short animation of the weapon attacking.



Player Controller

Create a new C# script called **PlayerController** and attach it to the player. We'll begin scripting in the next lesson.

PlayerController Script

Let's start with our namespaces.

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

Since we're going to be receiving RPC's, let's change our inheriting class.

```
public class PlayerController : MonoBehaviourPun
```

Then we can create our variables.

```
[HideInInspector]
public int id;

[Header("Info")]
public float moveSpeed;
public int gold;
public int curHp;
public int maxHp;
public bool dead;

[Header("Attack")]
public int damage;
public float attackRange;
public float attackRate;
private float lastAttackTime;

[Header("Components")]
public Rigidbody2D rig;
public Player photonPlayer;
public SpriteRenderer sr;
public Animator weaponAnim;

// local player
public static PlayerController me;
```

The first thing we're going to do, is set up the player movement and attacking. In the **Update** function, we'll call these functions.

```
void Update ()
{
    if(!photonView.IsMine)
        return;

    Move();

    if(Input.GetMouseButtonDown(0) && Time.time - lastAttackTime > attackRate)
```



```
        Attack();
}
```

In the **Move** function, we'll set the player's velocity based on the axis input.

```
void Move ()
{
    // get the horizontal and vertical inputs
    float x = Input.GetAxis("Horizontal");
    float y = Input.GetAxis("Vertical");

    // apply that to our velocity
    rig.velocity = new Vector2(x, y) * moveSpeed;
}
```

Let's now work on the **Attack** function. We're not going to complete it now, but fill in most of it.

```
// melee attacks towards the mouse
void Attack ()
{
    lastAttackTime = Time.time;

    // calculate the direction
    Vector3 dir = (Input.mousePosition - Camera.main.WorldToScreenPoint(transform.pos
ition)).normalized;

    // shoot a raycast in the direction
    RaycastHit2D hit = Physics2D.Raycast(transform.position + dir, dir, attackRange);

    // did we hit an enemy?
    if(hit.collider != null && hit.collider.gameObject.CompareTag("Enemy"))
    {
        // get the enemy and damage them
    }

    // play attack animation
    weaponAnim.SetTrigger("Attack");
}
```


Flipping the Player

In the **Update** function, let's set it up so we can flip the player horizontally.

```
float mouseX = (Screen.width / 2) - Input.mousePosition.x;

if(mouseX < 0)
    weaponAnim.transform.parent.localScale = new Vector3(-1, 1, 1);
else
    weaponAnim.transform.parent.localScale = new Vector3(1, 1, 1);
```

Taking Damage and Dying

Let's then create the **TakeDamage** function. This is called when we get hit by an enemy. It's also an RPC function, yet only ever called on the computer of the player who got hit. Since the master client will control the enemies, they'll need to send the RPC directly to the hit player.

```
[PunRPC]
public void TakeDamage (int damage)
{
    curHp -= damage;

    // update the health bar

    if(curHp <= 0)
        Die();
    else
    {
        StartCoroutine(DamageFlash());

        IEnumerator DamageFlash ()
        {
            sr.color = Color.red;
            yield return new WaitForSeconds(0.05f);
            sr.color = Color.white;
        }
    }
}
```

The **Die** function gets called when our health reaches 0.

```
void Die ()
{
    dead = true;
    rig.isKinematic = true;

    transform.position = new Vector3(0, 99, 0);

    Vector3 spawnPos = GameManager.instance.spawnPoints[Random.Range(0, GameManager.i
nstance.spawnPoints.Length)].position;
```



```
    StartCoroutine(Spawn(spawnPos, GameManager.instance.respawnTime));
}

IEnumerator Spawn (Vector3 spawnPos, float timeToSpawn)
{
    yield return new WaitForSeconds(timeToSpawn);

    dead = false;
    transform.position = spawnPos;
    curHp = maxHp;
    rig.isKinematic = false;

    // update the health bar
}
```

Initializing the Player

When the player is first spawned in, we need to call the **Initialize** function for all players.

In the video, 'rig.isKinematic' is set to false. It should be set to true.

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

    // initialize the health bar

    if(player.IsLocal)
        me = this;
    else
        rig.isKinematic = true;
}
```


Connecting the Player

In the **GameManager** script, let's spawn and setup the player there. First, we need to create an array to keep track of all the players.

```
public PlayerController[] players;
```

Then in the **Start** function, we can set the length of the array.

```
players = new PlayerController[PhotonNetwork.PlayerList.Length];
```

Next, we can continue with the **SpawnPlayer** function.

```
// initialize the player
playerObj.GetComponent<PhotonView>().RPC("Initialize", RpcTarget.All, PhotonNetwork.L
ocalPlayer);
```

Back in the PlayerController Script

Back in the **PlayerController** script, we can add ourselves to the players array in the **Initialize** function.

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

Now let's add some other functions which we'll be using later on. The first is **Heal**. This will heal the player a certain amount of health.

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

    // update the health bar
}
```

Next, we have the **GiveGold** function, which does the same as Heal, but with gold.

```
[PunRPC]
void GiveGold (int goldToGive)
{
    gold += goldToGive;

    // update the ui
}
```

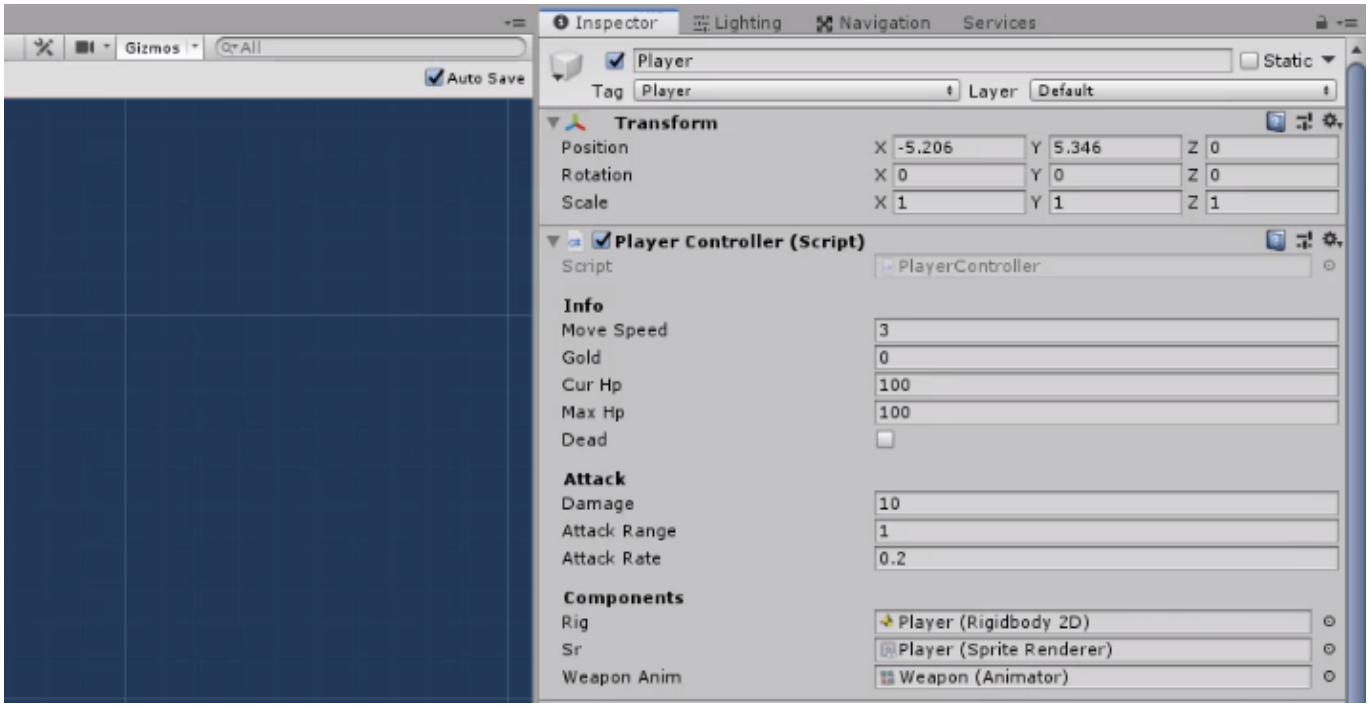

Back in the editor, we can drag the player object into the **Resources** folder and delete the one in the scene.

Select the **_GameManager**.

- Set **Player Prefab Path** to *Player*
- Set **Respawn Time** to *3*

Edit the **Player** prefab.

- Add a **PhotonView** component
- Add a **PhotonTransformView** component and drag it into the PhotonView's *Observed Components*
- Let's then fill in the **Player Controller** properties:
    - Set **Move Speed** to *3*
    - Set **Cur Hp** and **Max Hp** to *100*
    - Set **Damage** to *10*
    - Set **Attack Range** to *1*
    - Set **Attack Rate** to *0.2*
    - Connect up the 3 components



CameraController Script

Create a new C# script called **CameraController** and attach it to the camera. This is going to track the local player.

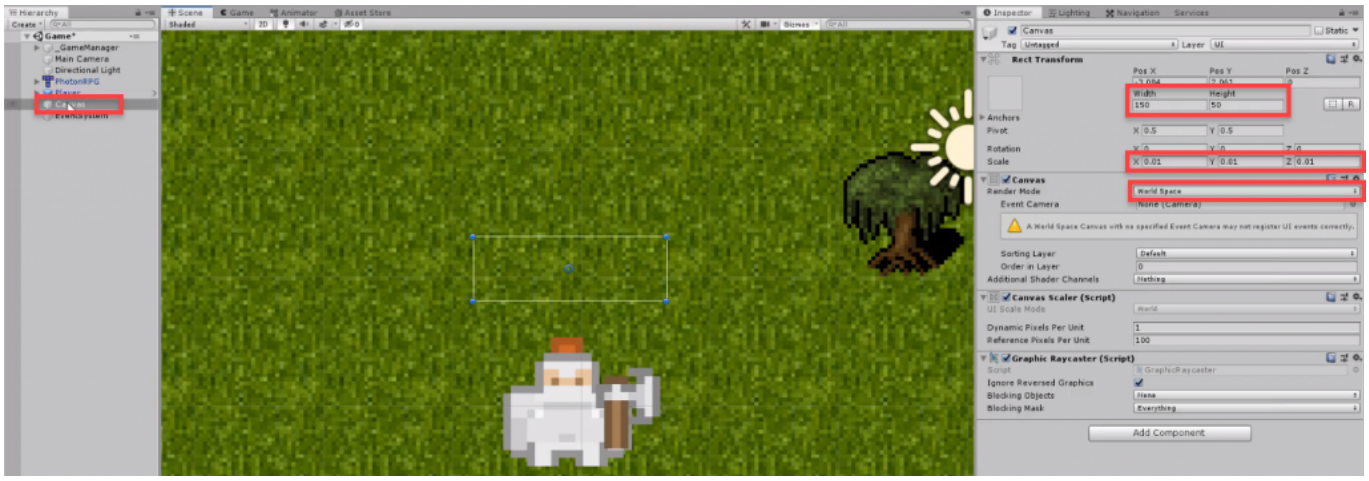
```
void Update ()
{
    // does the player exist?
    if(PlayerController.me != null && !PlayerController.me.dead)
    {
        Vector3 targetPos = PlayerController.me.transform.position;
        targetPos.z = -10;

        transform.position = targetPos;
    }
}
```

You should now be able to play the game and see that the camera tracks the player.

Creating the Header UI

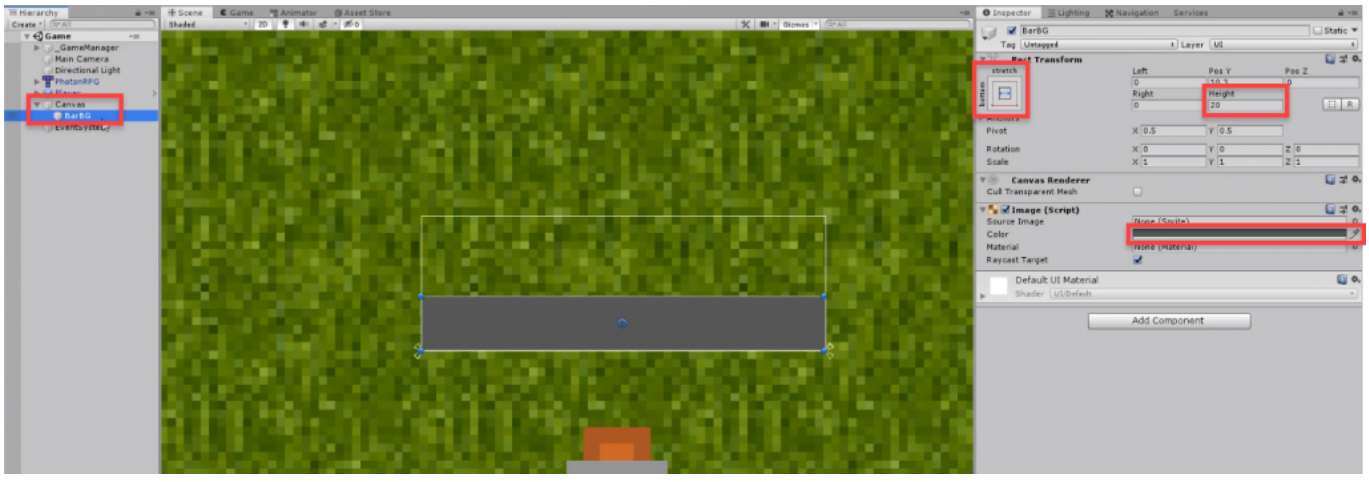
We're going to have a UI info bar on top of the player and enemy's heads. This will display their name and current health. Create a new canvas object and set the **Render Mode** to *World Space*.

- Set the **Width** to *150*
- Set the **Height** to *50*
- Set the **Scale** to *0.01*
- Set the **Sorting Layer** to *UI*



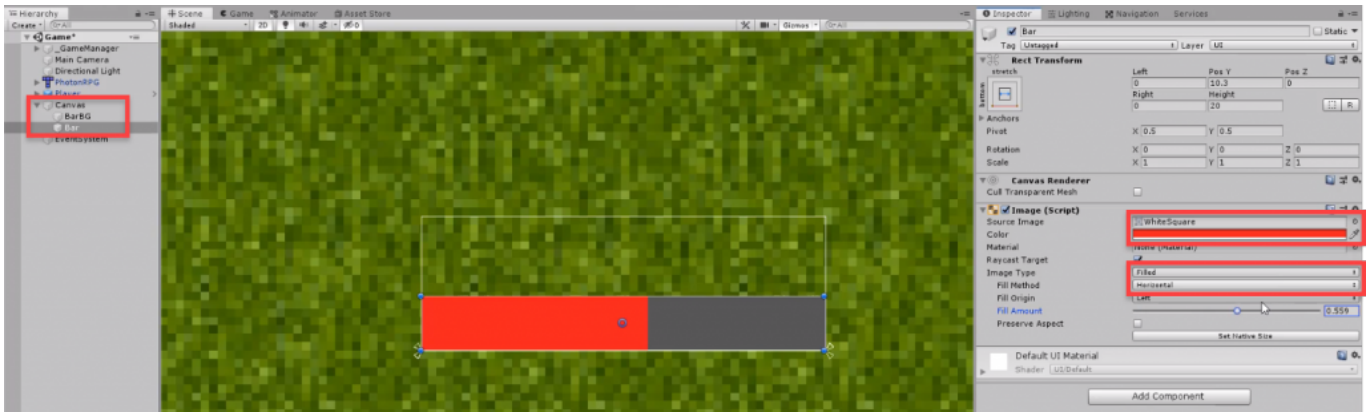
As a child of the canvas, create a new image and call it **BarBG**. This will be the background for our health bar.

- Set the **Height** to *20*
- Set the **Anchoring** to *bottom-stretch*
- Set the **Color** to grey



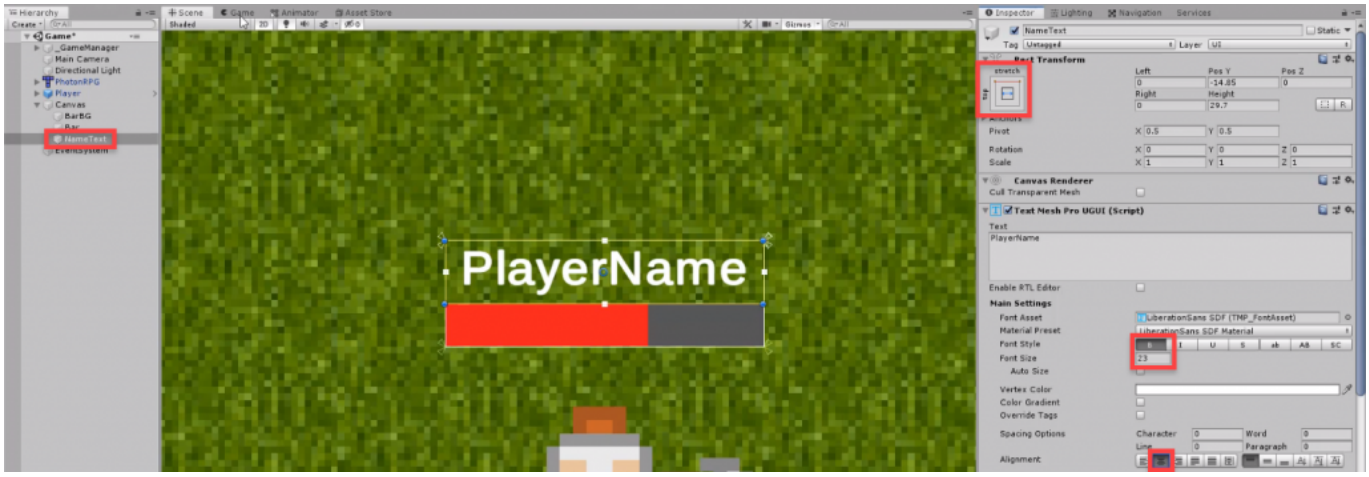
To create the fill of the health bar, duplicate the image and call it **Bar**.

- Set the **Source Image** to *WhiteSquare*
- Set the **Color** to red
- Set the **Image Type** to *Filled*
- Set the **Fill Method** to *Horizontal*



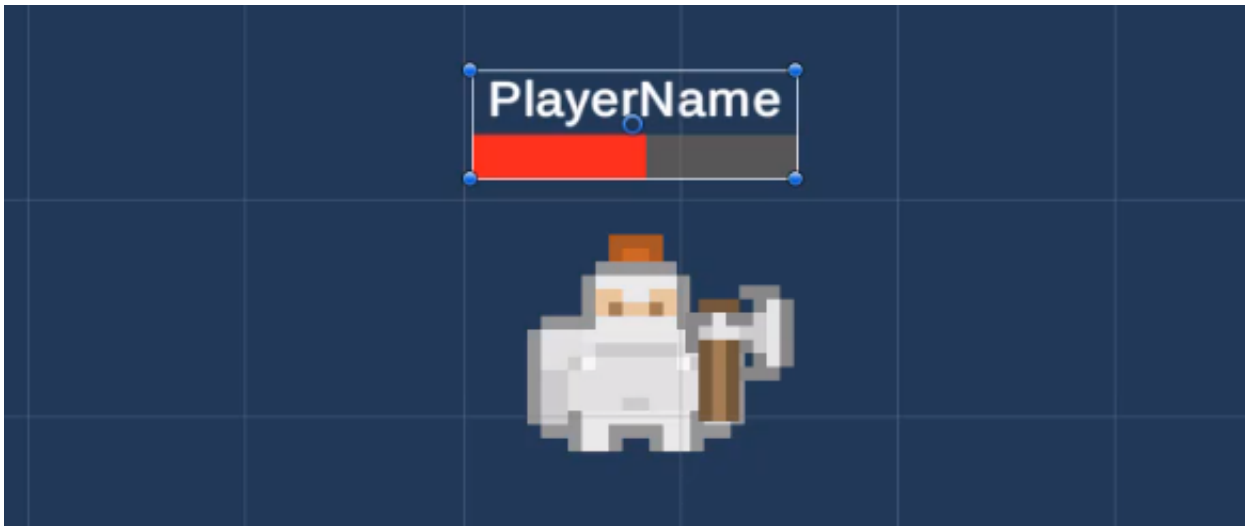
For the name text, create a new **Text Mesh Pro – Text** object and call it **NameText**.

- Set the **Anchoring** to *top-stretch*
- Make the rect fill in all the remaining space
- Set the **Font Style** to ***Bold***
- Set the **Font Size** to *23*
- Set the **Alignment** to *middle-top*



Rename the canvas to **HeaderInfo** and save it as a prefab in the **Prefabs** folder.

Open up the player prefab and drag in a header info. Position this on top of their head.



HeaderInfo Script

Create a new C# script called **HeaderInfo** and attach it to the HeaderInfo prefab. Also attach a **PhotonView** component.

First, we need to add our namespaces.

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

Our variables are...

```
public TextMeshProUGUI;
public Image bar;
private float maxValue;
```

The **Initialize** function gets called when the parent object is spawned in.

```
public void Initialize (string text, int maxVal)
{
    nameText.text = text;
    maxValue = maxVal;
    bar.fillAmount = 1.0f;
}
```

The **UpdateHealthBar** function gets called when we want to update the bar visual.

```
[PunRPC]
void UpdateHealthBar (int value)
{
    bar.fillAmount = (float)value / maxValue;
```



```
}
```

Connecting it to the Player

In the **PlayerController** script, let's hook up the header info. Create a new variable to reference it.

```
public HeaderInfo headerInfo;
```

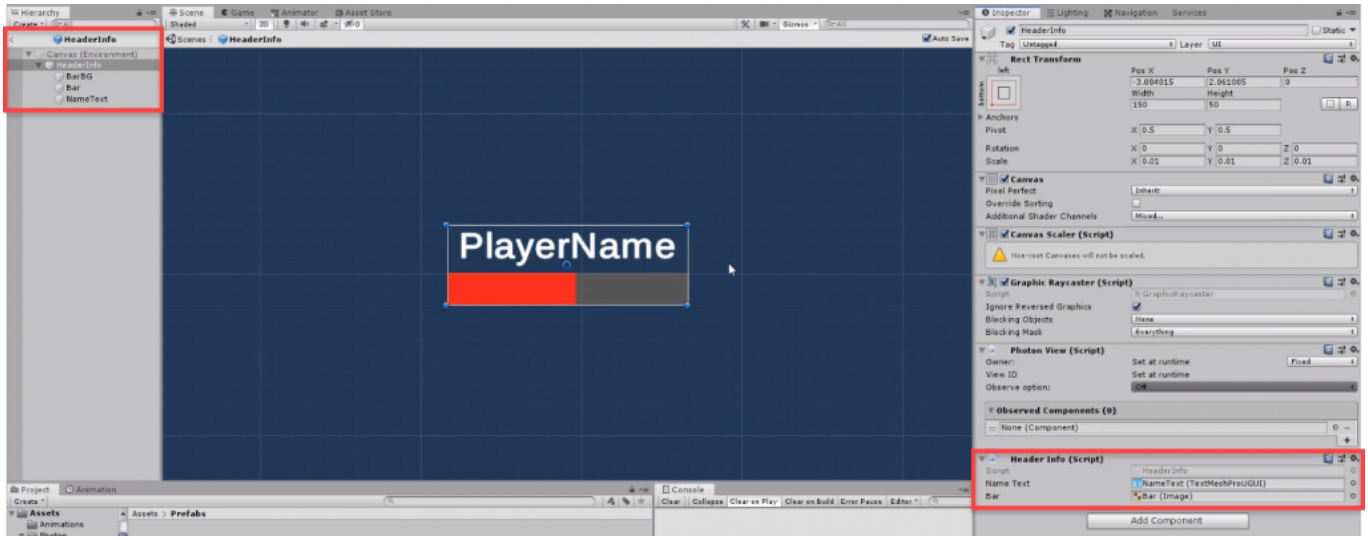
In the **Initialize** function, we'll initialize the header info.

```
// initialize the health bar
headerInfo.Initialize(player.NickName, maxHp);
```

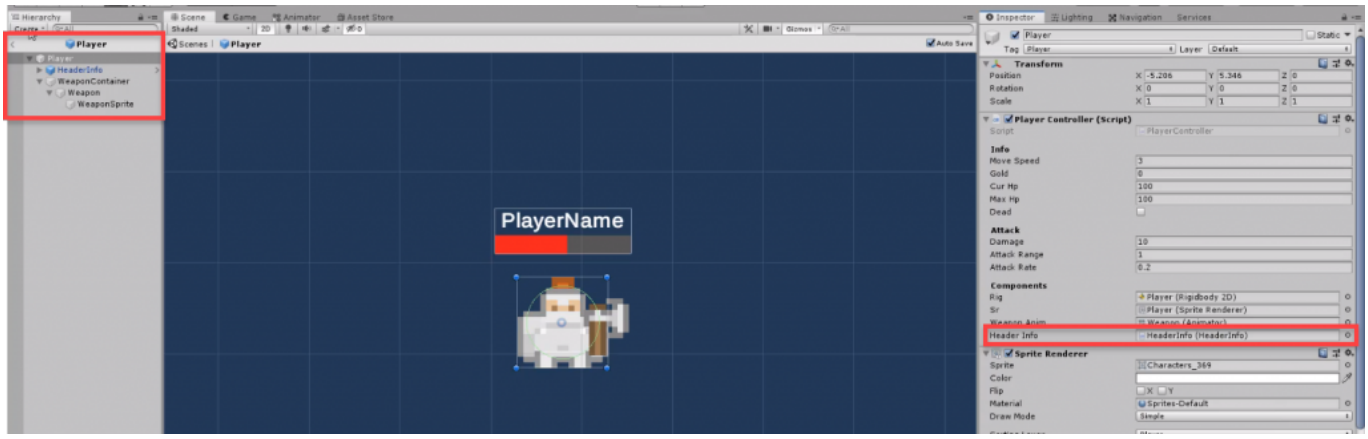
In the **TakeDamage** and **Heal** functions, we'll update the health bar.

```
// update the health bar
headerInfo.photonView.RPC("UpdateHealthBar", RpcTarget.All, curHp);
```

Back in the editor, we can open up the **HeaderInfo** prefab and connect the two public variables.



We can then open the **Player** prefab and connect the header info.

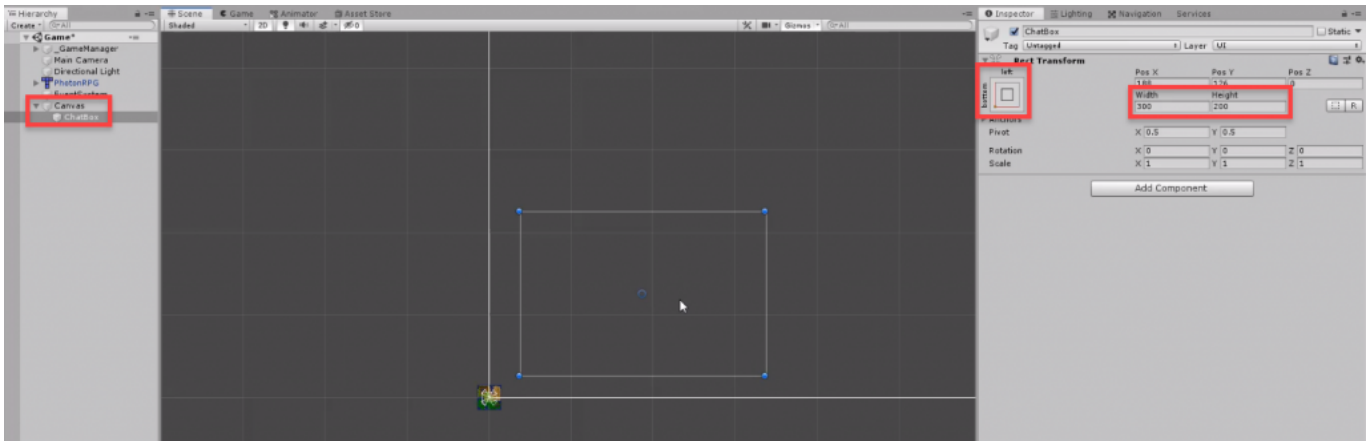


Creating the Chat Box UI

To create our chat box, let's start by creating a new canvas. This is going to be not just for the chat box, but for any other UI we want stuck on the screen.

As a child of the canvas, create a new empty GameObject called **ChatBox**.

- Set the **Width** to *300*
- Set the **Height** to *200*
- Set the **Anchoring** to *bottom-left*

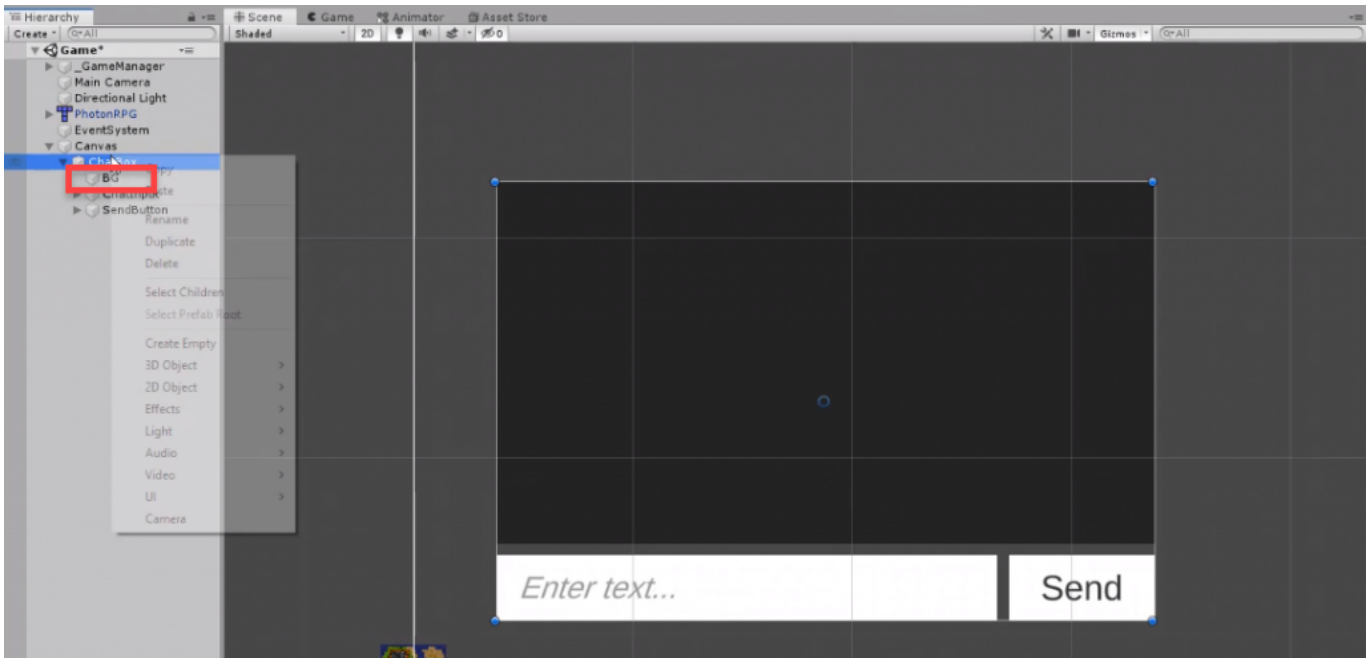


Create a new **Text Mesh Pro – Input Field** and call it **ChatInput**. Then create a new **Text Mesh Pro – Button,** calling it **SendButton**.

- Set the input field's **Source Image** to *none*
- Set the button's text **Font Size** to *16*
- Lay them out as seen below

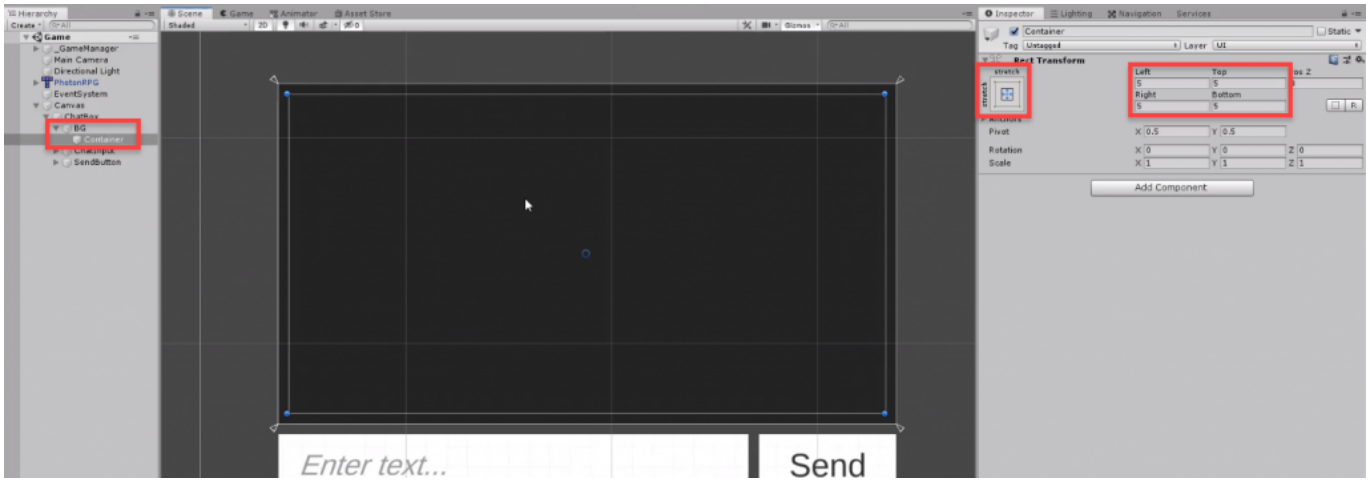
For the text, let's create a new image as a background. Call it **BG**.

- Set the **Color** to black with an alpha value of *20*



To contain the text, let's create a new empty GameObject as a child of BG called **Container**.

- Set the **Anchoring** to *stretch-stretch*
- Set the **Left**, **Top**, **Right** and **Bottom** to *5*



Now let's create the text. Create a new **Text Mesh Pro – Text** object called **ChatLogText** as a child of Container.

- Set the **Font Size** to *18*
- Set the **Anchoring** to *left-bottom*
- Set the **Pivot** to *0.5, 0*
- Set the **Anchoring** to *bottom-stretch*

We also want to move and resize the **Container** because we need to fit in a scrollbar.

Create a new **Scrollbar** object as a child of the **BG**.

- Set the **Direction** to *Bottom To Top*
- Set the **Source Image** to *none*
- Set the **Color** to grey
- Set the Handle's **Source Image** to *none*
- Set the Handle's **Color** to white
- Set the **Width** to *10*

Selecting the **Container**, add two new components: **Scroll Rect** and **Rect Mask 2D**.

- Set the **Content** to *ChatLogText*
- Disable **Horizontal**
- Set the **Scroll Sensitivity** to *15*
- Set the **Viewport** to *Container*
- Set the **Vertical Scrollbar** to *Scrollbar*

In the next lesson, we'll be scripting the chat box.

ChatBox Script

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

Let's fill in our namespaces.

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

Since we'll be receiving RPC's, let's change our inheriting class.

```
public class ChatBox : MonoBehaviourPun
```

Then we can create our variables.

```
public TextMeshProUGUI chatLogText;
public TMP_InputField chatInput;

// instance
public static ChatBox instance;

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

The **OnChatInputSend** function gets called when either the 'Enter' key is pressed or the *Send* button on the chat box is pressed.

```
// called when the player wants to send a message
public void OnChatInputSend ()
{
    if(chatInput.text.Length > 0)
    {
        photonView.RPC("Log", RpcTarget.All, PhotonNetwork.LocalPlayer.NickName, chat
Input.text);
        chatInput.text = "";
    }

    EventSystem.current.SetSelectedGameObject(null);
}
```

The **Log** function is an RPC that gets sent when a player sends a message. All players receive this RPC call.

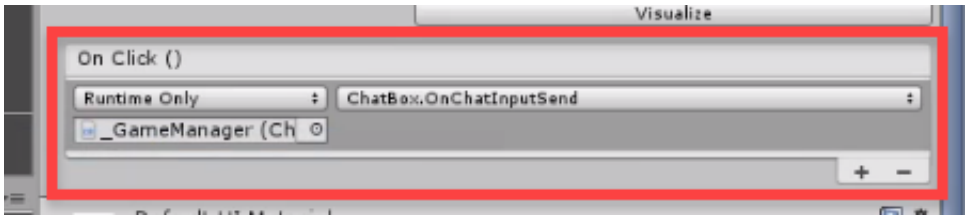

```
[PunRPC]
void Log (string playerName, string message)
{
    chatLogText.text += string.Format("<br>{0}:</b> {1}", playerName, message);

    chatLogText.rectTransform.sizeDelta = new Vector2(chatLogText.rectTransform.sizeD
elta.x, chatLogText.mesh.bounds.size.y + 20);
}
```

In the **Update** function, we can check for the player to press the 'Enter' key. If they're typing, send the message, otherwise focus on the input field.

```
void Update ()
{
    if(Input.GetKeyDown(KeyCode.Return))
    {
        if(EventSystem.current.currentSelectedGameObject == chatInput.gameObject)
            OnChatInputSend();
        else
            EventSystem.current.SetSelectedGameObject(chatInput.gameObject);
    }
}
```

Back in the editor, we can fill in the two properties. Then selecting the **SendButton**, add a new event listener to the **OnClick** event.

- Set this to **ChatBox.OnChatInputSend()**

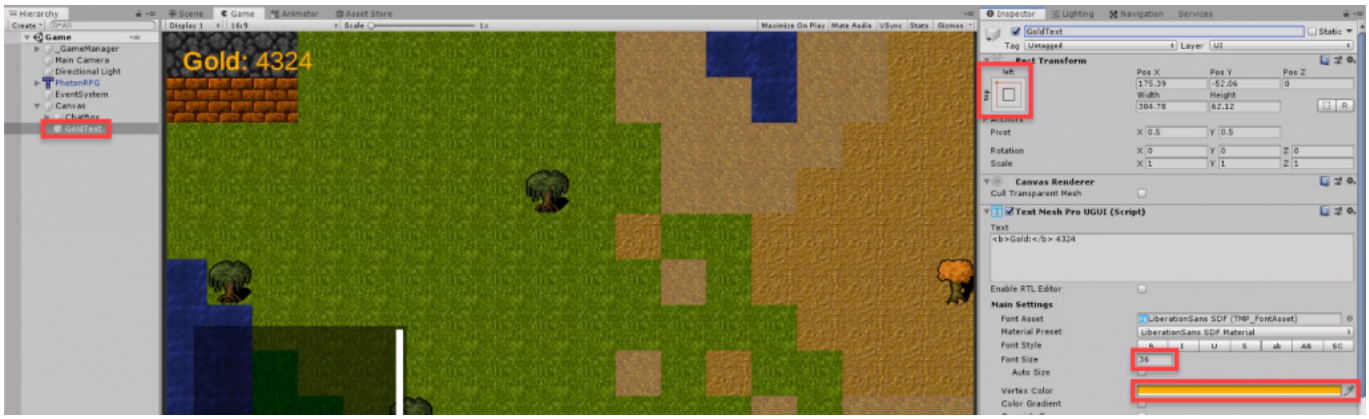


You can now try it our in-game!

Gold Text

Create a new **Text Mesh Pro – Text** object as a child of the canvas called **GoldText**. Position it at the top left corner of the screen.

- Set the **Anchoring** to *top-left*
- Set the **Font Size** to *36*
- Set the **Vertex Color** to gold



GameUI Script

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

First, our namespaces.

```
using UnityEngine.UI;
using TMPro;
```

Then we can create our only variable.

```
public TextMeshProUGUI goldText;

// instance
public static GameUI instance;

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

The **UpdateGoldText** function gets called when we want to update the gold text.

```
public void UpdateGoldText (int gold)
{
    goldText.text = "<b>Gold:</b> " + gold;
}
```


We can call this function over in the **PlayerController** script, down in the **GiveGold** function.

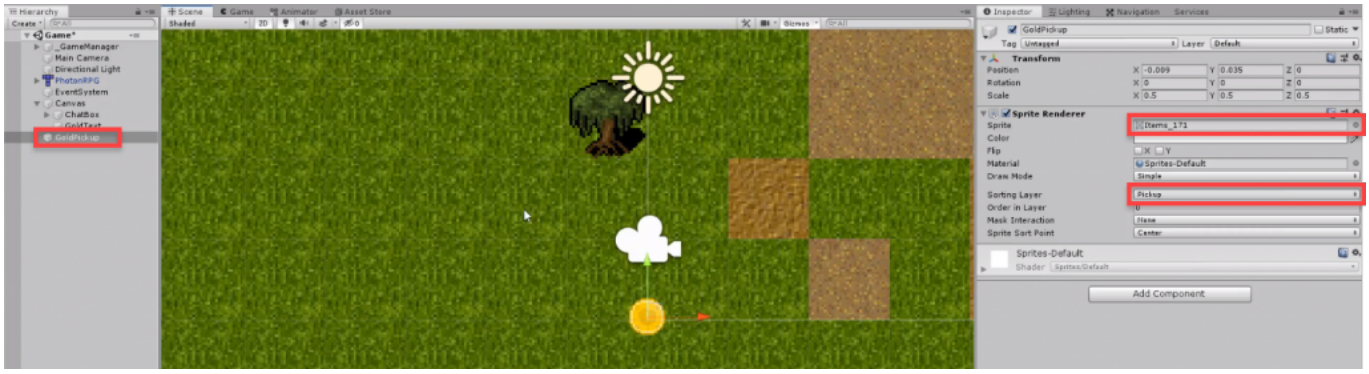
```
// update the ui
GameUI.instance.UpdateGoldText(gold);
```

Back in the editor, connect the text to the script.

Creating the Pickups

Create a new sprite object and call it **GoldPickup**.

- Set the **Sprite** to be a gold coin
- Set the **Sorting Layer** to *Pickup*
- Set the **Scale** to *0.5*



Now we need to add a few components.

- **CircleCollider2D**
    - Enable **Is Trigger**
- **PhotonView**
    - Set the **View ID** to something like 30

Pickup Script

Create a new C# script called **Pickup** and attach it to the pickup.

First, our namespaces.

```
using Photon.Pun;
```

Then changing our inheriting class.

```
public class Pickup : MonoBehaviourPun
```

Just above the class, let's create an enumerator called **PickupType** which will keep track of the different pickup types.

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

Then back in the class, we can enter in our variables.


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

To detect if a player has picked it up, we'll check this in the **OnTriggerEnter2D** function. The master client will check this and send the respective RPC to the player who entered the trigger.

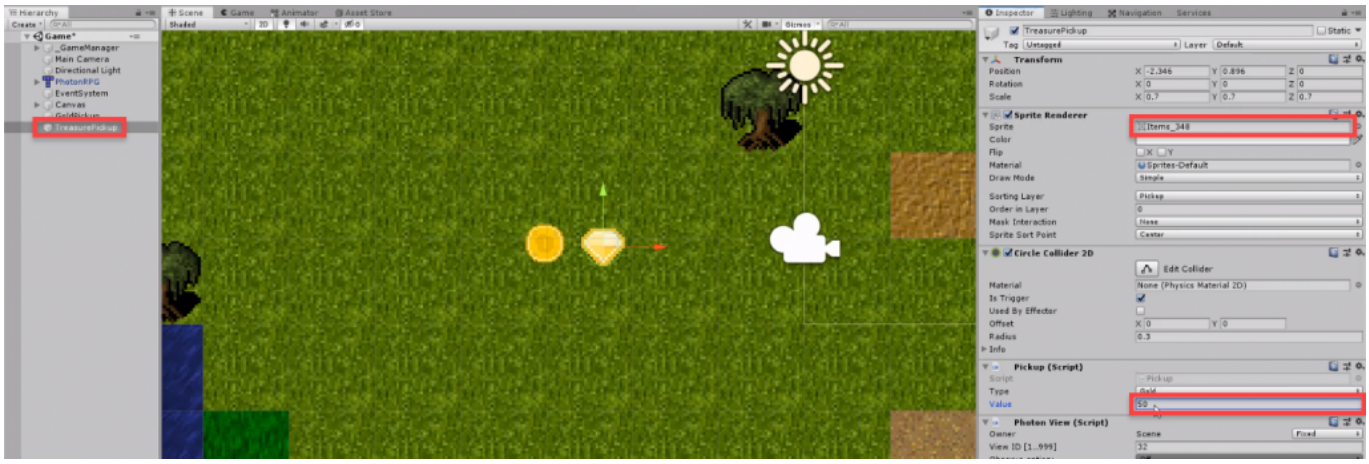
```
void OnTriggerEnter2D (Collider2D collision)
{
    if(!PhotonNetwork.IsMasterClient)
        return;

    if(collision.CompareTag("Player"))
    {
        PlayerController player = collision.gameObject.GetComponent<PlayerController>
();

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

        PhotonNetwork.Destroy(gameObject);
    }
}
```

Back in the editor, we can set the value for our gold pickup.

You should now be able to test this out in-game.

Creating the Other Pickups

Duplicate the **GoldPickup** and call it **TreasurePickup**. This is going to be like the gold, but give more.



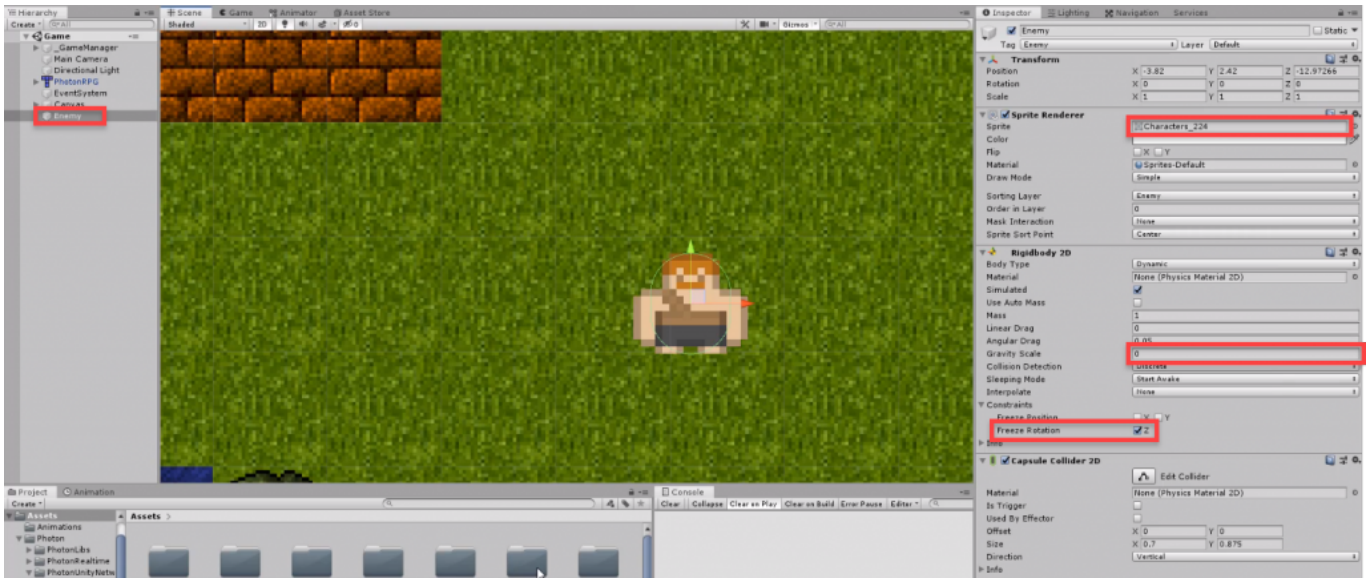
We can then duplicate again and call it **HealthPickup**. This will give health instead of gold.

Drag all three of these into the Resources folder.

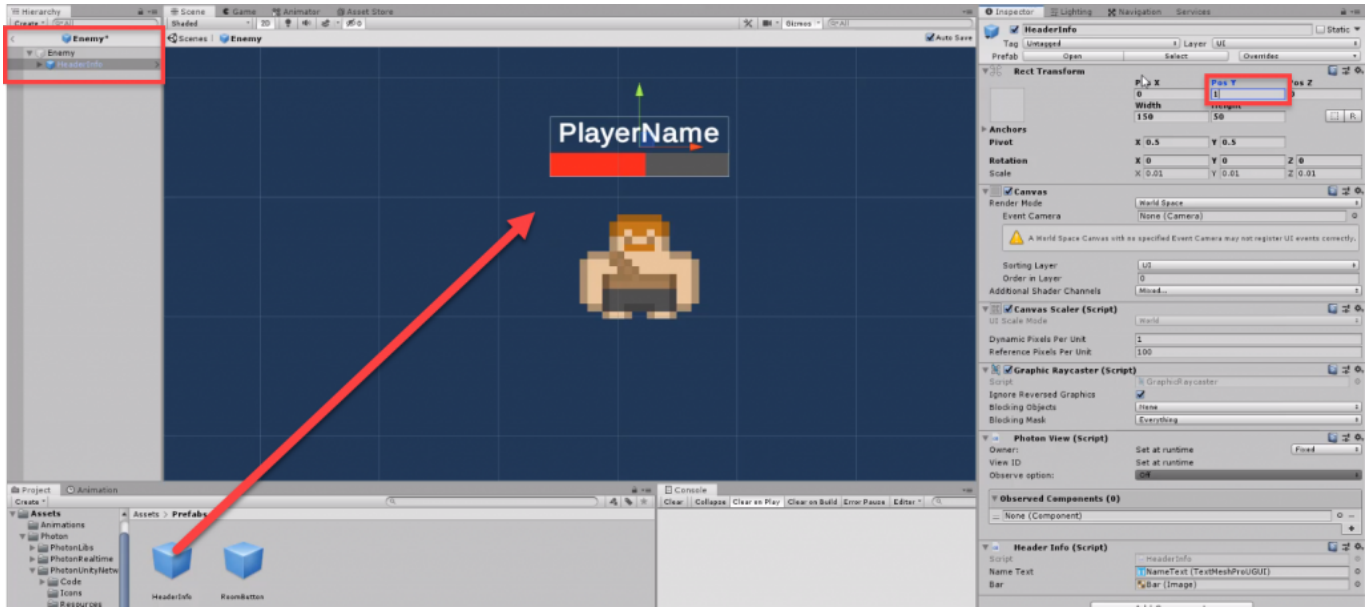
Enemy Object

Create a new sprite object called **Enemy**. Set the sprite to be a character sprite. Add the following components.

- **Rigidbody2D**
  - Set the **Gravity Scale** to *0*
  - Enable **Constraints – Freeze Z Rotation**
- **CapsuleCollider2D**
  - Set the **Size** to *0.7, 0.875*



Save the enemy as a prefab in the Resources folder. Then, we can add a header info the enemy.



Enemy Script

Create a new C# script called **Enemy** and attach it to the enemy. First, we'll setup the namespaces.

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

Then change the inheriting class.

```
public class Enemy : MonoBehaviourPun
```

Here's the variables we're going to have.

```
[Header("Info")]
public string enemyName;
public float moveSpeed;

public int curHp;
public int maxHp;

public float chaseRange;
public float attackRange;

private PlayerController targetPlayer;

public float playerDetectRate = 0.2f;
private float lastPlayerDetectTime;

public string objectToSpawnOnDeath;

[Header("Attack")]
public int damage;
public float attackRate;
private float lastAttackTime;

[Header("Components")]
public HeaderInfo healthBar;
public SpriteRenderer sr;
public Rigidbody2D rig;
```

In the **Start** function, we'll initialize the health bar.

```
void Start ()
{
    healthBar.Initialize(enemyName, maxHp);
}
```

Back in the editor, attach the **PhotonView** and **PhotonTransformView** components to the **Enemy** prefab. Drag the transform view into the photon view's *Observable Components* list.

Enemy Script

In the **Update** function, we'll first make sure that only the master client can control the enemy.

```
void Update ()
{
    if(!PhotonNetwork.IsMasterClient)
        return;

    if(targetPlayer != null)
    {
        // calculate the distance
        float dist = Vector3.Distance(transform.position, targetPlayer.transform.posi
tion);

        // if we're able to attack, do so
        if(dist < attackRange && Time.time - lastAttackTime >= attackRange)
            Attack();
        // otherwise, do we move after the player?
        else if(dist > attackRange)
        {
            Vector3 dir = targetPlayer.transform.position - transform.position;
            rig.velocity = dir.normalized * moveSpeed;
        }
        else
        {
            rig.velocity = Vector2.zero;
        }
    }

    DetectPlayer();
}
```

The **Attack** function will attack the target player.

```
// attacks the targeted player
void Attack ()
{
    lastAttackTime = Time.time;
    targetPlayer.photonView.RPC("TakeDamage", targetPlayer.photonPlayer, damage);
}
```

The **DetectPlayer** function will check if a player is within the chase range. If so, target them.

```
// updates the targeted player
void DetectPlayer ()
{
    if(Time.time - lastPlayerDetectTime > playerDetectRate)
    {
        lastPlayerDetectTime = Time.time;
```



```
        // loop through all the players
        foreach(PlayerController player in GameManager.instance.players)
        {
            // calculate distance between us and the player
            float dist = Vector2.Distance(transform.position, player.transform.positi
on);

            if(player == targetPlayer)
            {
                if(dist > chaseRange)
                    targetPlayer = null;
            }
            else if(dist < chaseRange)
            {
                if(targetPlayer == null)
                    targetPlayer = player;
            }
        }
    }
}
```


Enemy Script

The **TakeDamage** function is going to be called when the enemy gets attacked by a player.

```
[PunRPC]
public void TakeDamage (int damage)
{
    curHp -= damage;

    // update the health bar
    healthBar.photonView.RPC("UpdateHealthBar", RpcTarget.All, curHp);

    if(curHp <= 0)
        Die();
    else
    {
        photonView.RPC("FlashDamage", RpcTarget.All);
    }
}
```

FlashDamage is a function which gets called when the enemy is hit. It flashes the enemy across the network.

```
[PunRPC]
void FlashDamage ()
{
    StartCoroutine(DamageFlash());

    IEnumerator DamageFlash ()
    {
        sr.color = Color.red;
        yield return new WaitForSeconds(0.05f);
        sr.color = Color.white;
    }
}
```

The **Die** function gets called when the enemy's health reaches 0.

```
void Die ()
{
    if(objectToSpawnOnDeath != string.Empty)
        PhotonNetwork.Instantiate(objectToSpawnOnDeath, transform.position, Quaternio
n.identity);

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

PlayerController Script

In the **PlayerController** script, we're going to set up attacking enemies and some other minor things. First, in the **Spawn** function, we can update the health bar.

```
// update health bar
headerInfo.photonView.RPC("UpdateHealthBar", RpcTarget.All, curHp);
```

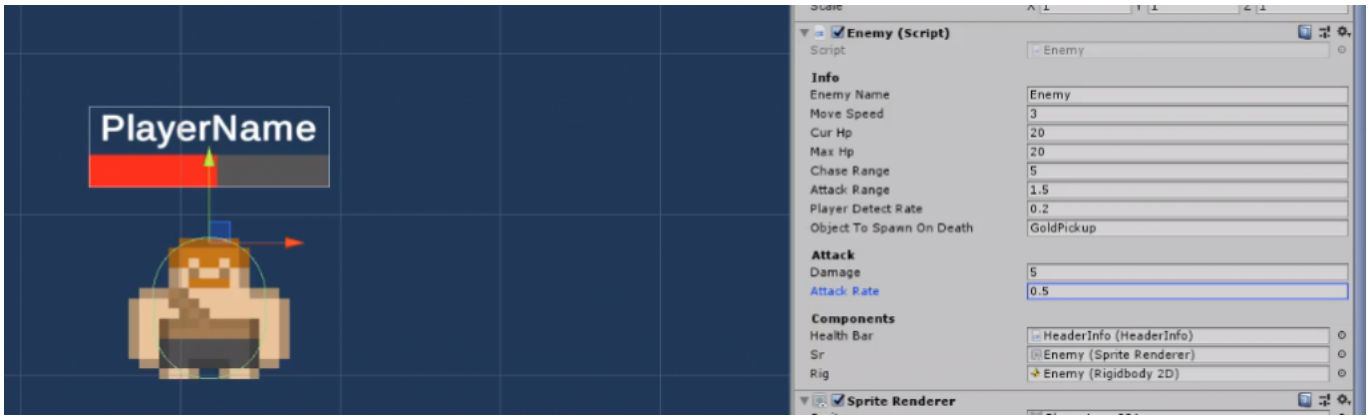
In the **Attack** function, we can damage the hit enemy inside of the *if* statement.

```
// get the enemy and damage them
Enemy enemy = hit.collider.GetComponent<Enemy>();
enemy.photonView.RPC("TakeDamage", RpcTarget.MasterClient, damage);
```

Enemy Prefab

In the enemy prefab, let's set the values.

- Set **Enemy Name** to *Enemy*
- Set **Move Speed** to *3*
- Set **Cur Hp** and **Max Hp** to *10*
- Set **Chase Range** to *5*
- Set **Attack Range** to *0.5*
- Set **Object To Spawn On Death** to *GoldPickup*
- Set **Damage** to *5*
- Set **Attack Rate** to *0.5*



EnemySpawner Script

Create a new C# script called **EnemySpawner** and attach it to a new GameObject called **EnemySpawner**. These are going to be individual spawners we can place wherever.

Let's start with our single namespace.

```
using Photon.Pun;
```

And change our inheriting class.

```
public class EnemySpawner : MonoBehaviourPun
```

We can then create our variables.

```
public string enemyPrefabPath;
public float maxEnemies;
public float spawnRadius;
public float spawnCheckTime;

private float lastSpawnCheckTime;
private List<GameObject> curEnemies = new List<GameObject>();
```

In the **Update** function, we can check the spawn condition.

```
void Update ()
{
    if(!PhotonNetwork.IsMasterClient)
        return;

    if(Time.time - lastSpawnCheckTime > spawnCheckTime)
    {
        lastSpawnCheckTime = Time.time;
        TrySpawn();
    }
}
```

The **TrySpawn** function will be called every 'spawnCheckTime' seconds and will check if we're able to spawn a new enemy.

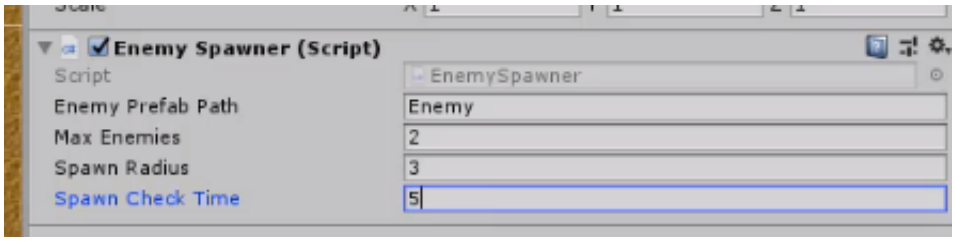
```
void TrySpawn ()
{
    // remove any dead enemies from the curEnemies list
    for(int x = 0; x < curEnemies.Count; ++x)
    {
        if(!curEnemies[x])
            curEnemies.RemoveAt(x);
```



```
    }

    // if we have maxed out our enemies, return
    if(curEnemies.Count >= maxEnemies)
        return;

    // otherwise, spawn an enemy
    Vector3 randomInCircle = Random.insideUnitCircle * spawnRadius;
    GameObject enemy = PhotonNetwork.Instantiate(enemyPrefabPath, transform.position
+ randomInsideCircle, Quaternion.identity);
    curEnemies.Add(enemy);
}
```

Back in the editor, we can fill in the spawner properties.

- Set **Enemy Prefab Path** to *Enemy*
- Set **Max Enemies** to *2*
- Set **Spawn Radius** to *3*
- Set **Spawn Check Time** to *5*



Congratulations on Finishing the Course!

Let’s have a look at what we learned and made.

- We learned how to use **Photon** inside of Unity
- We created a **player controller** which can move, attack, get gold and heal
- **Pickups** which can give the player health or gold
- A **chat box** which can allow players to talk to each other
- A **lobby system** for players to team up together before entering the game
- This was all done using Unity’s **UI system**