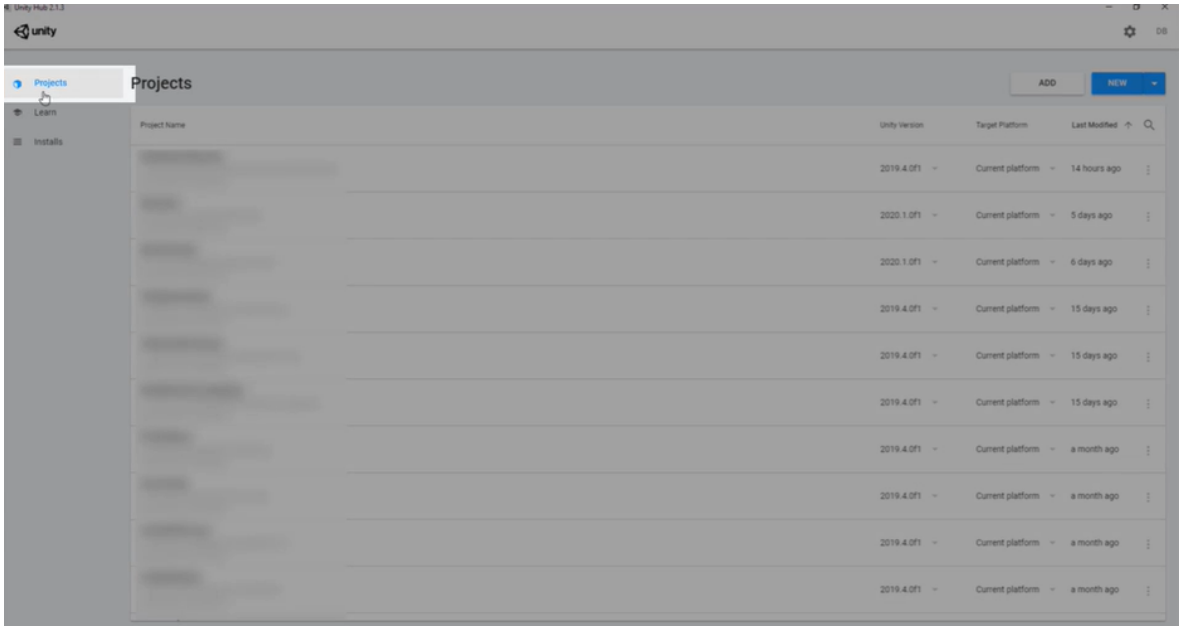
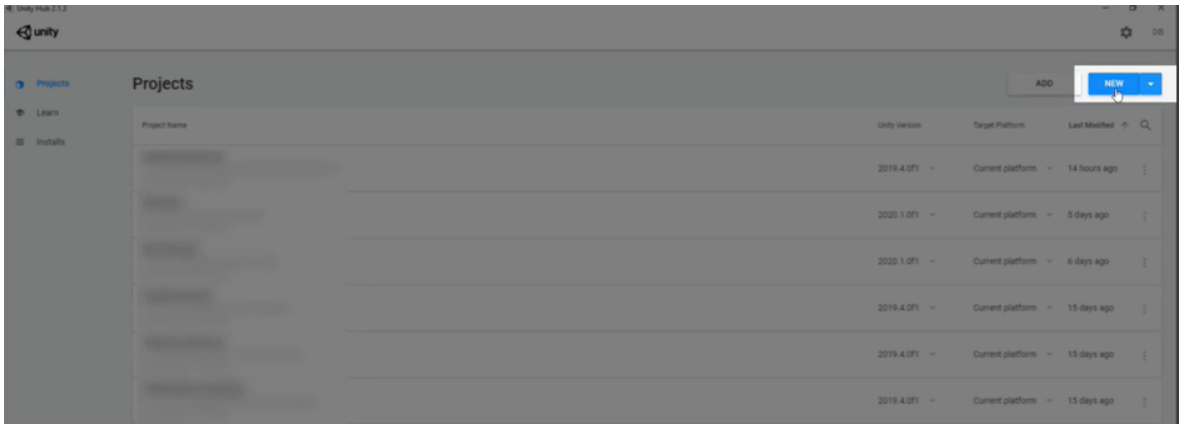
In this lesson, we're going to be setting up the initial project.

You need to have the latest version of **Unity Hub** installed if you haven't already. Download Unity Hub: https://unity3d.com/get-unity/download

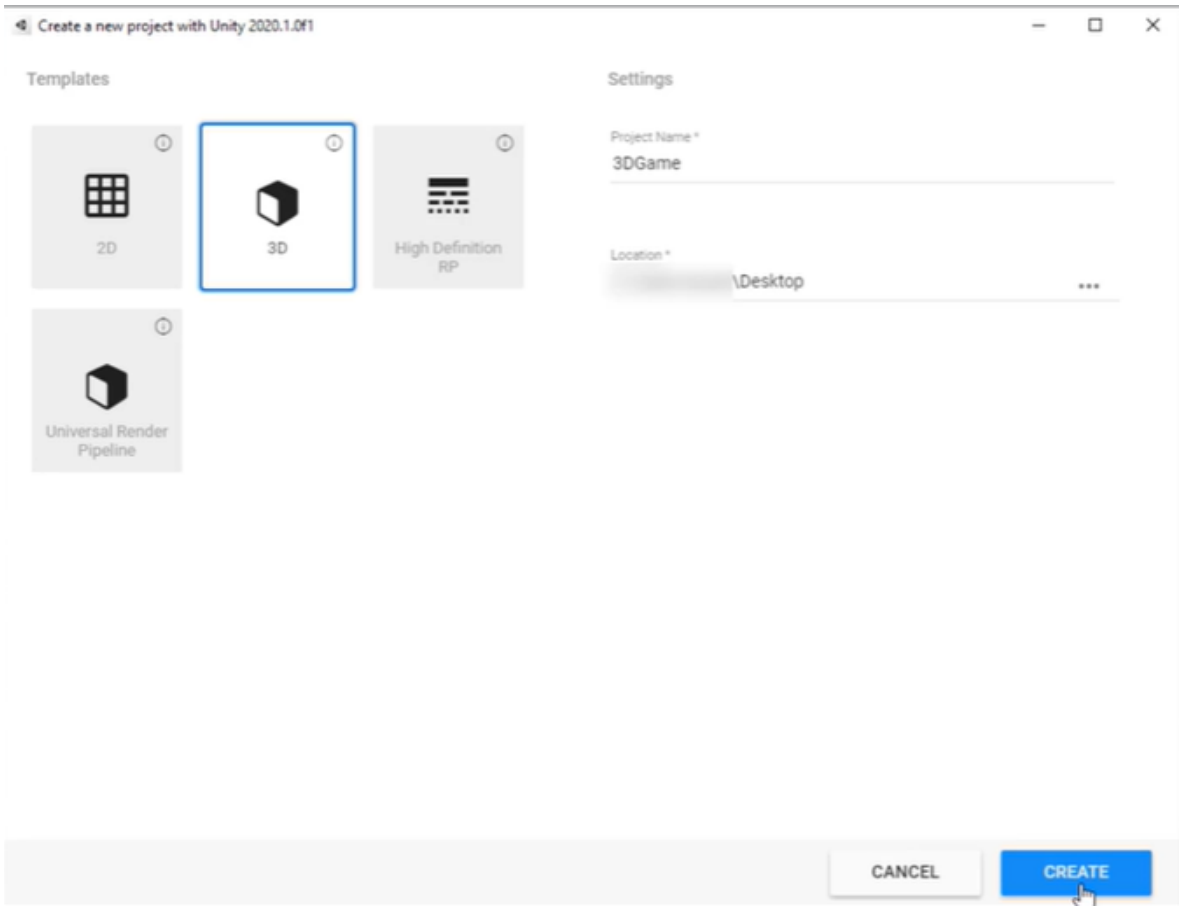
Open up Unity hub, and click on the **Projects** tab:



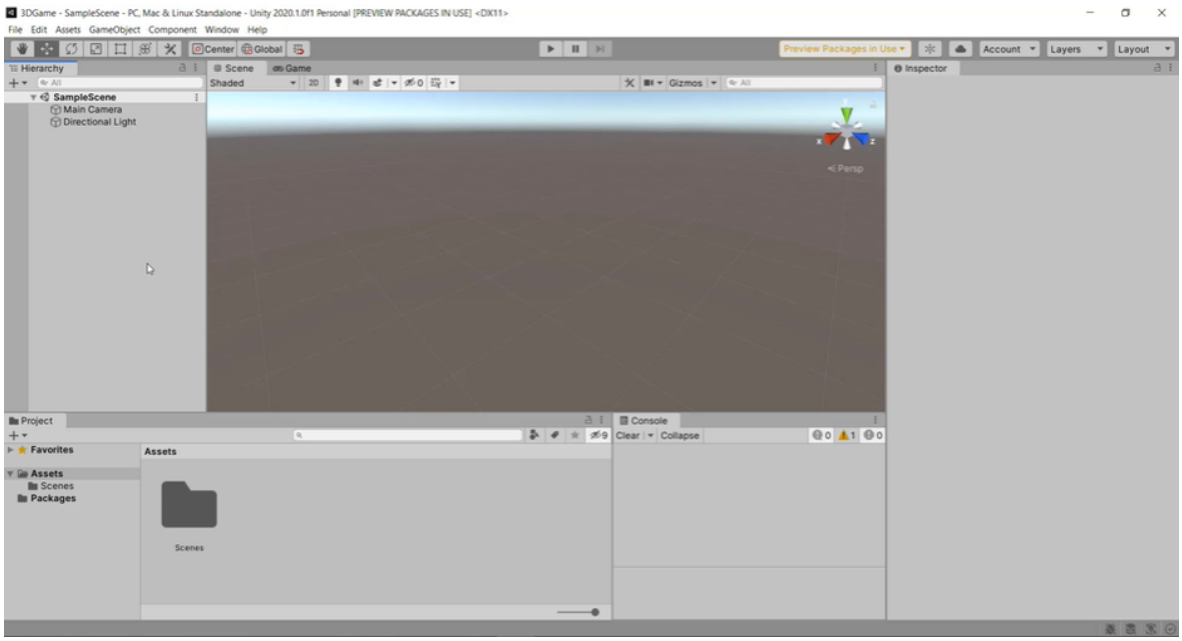
... and click on the **New** button**:**



Select the **3D template**, and set the project's name and location. Then click on the '**Create**' button.



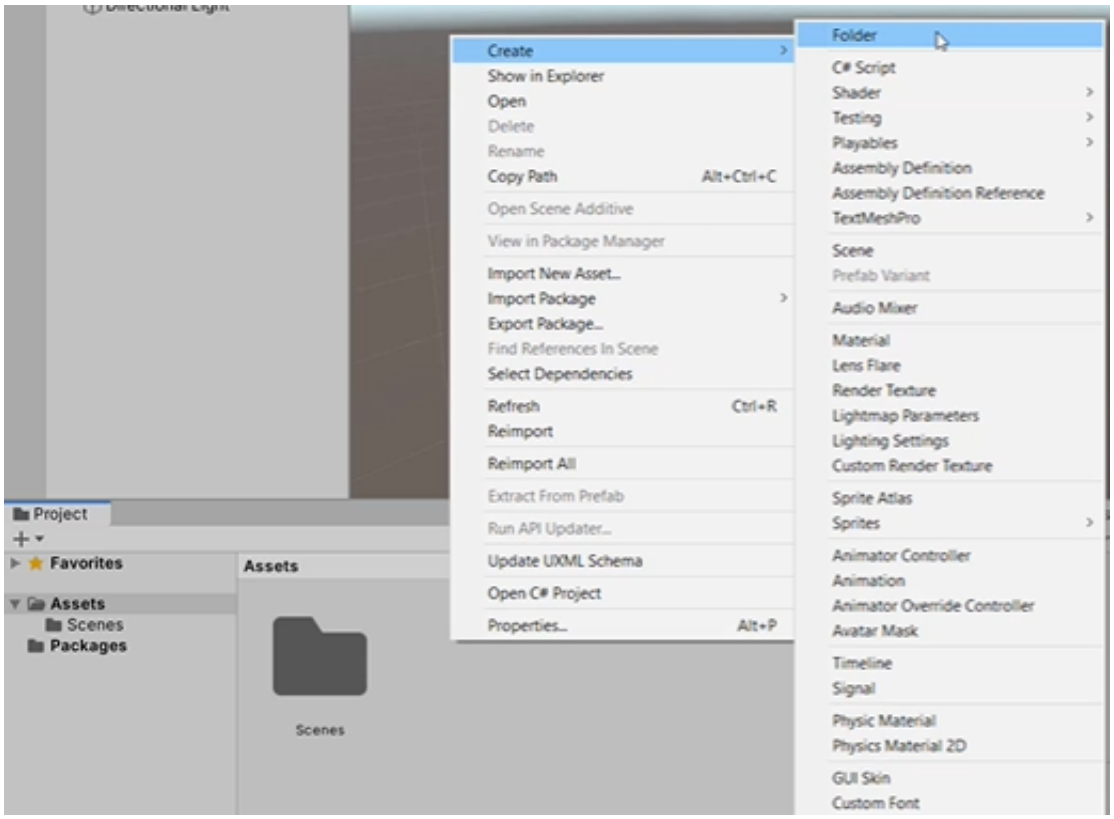
We now have the Unity Editor open inside of our brand new project.



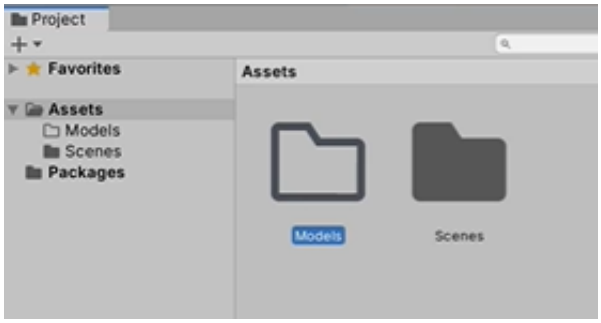
In this lesson, we're going to be importing a number of different 3D models into our project.

First of all, we're going to **create a new folder** to store our 3D models.

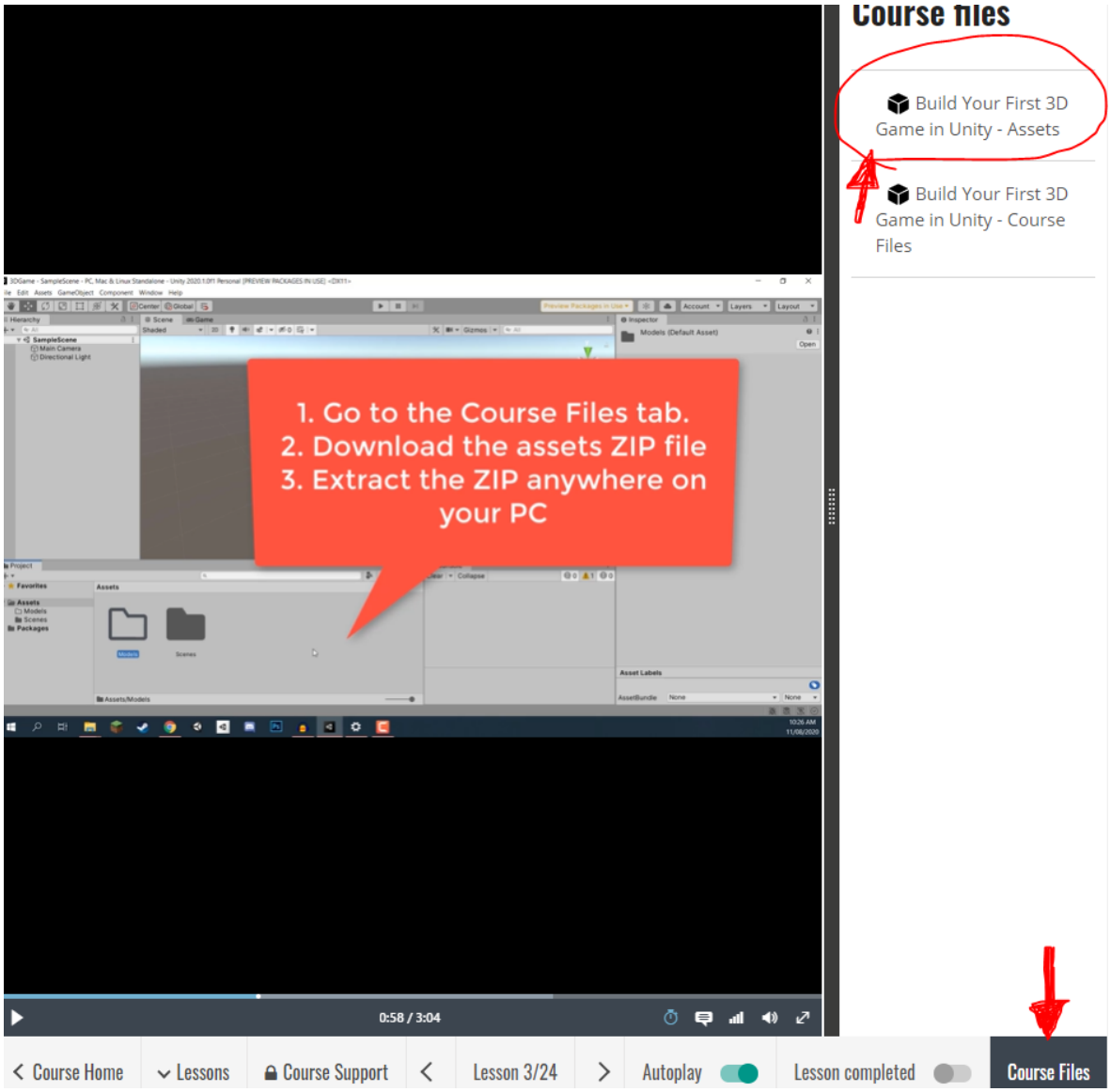
Right-click on the project panel, and go to **Create > Folder**.

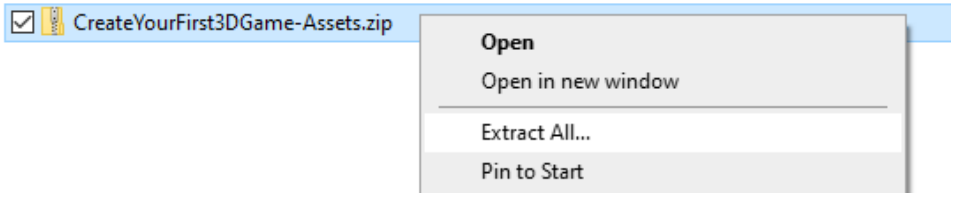


Then we're going to **name** the folder as "Models".

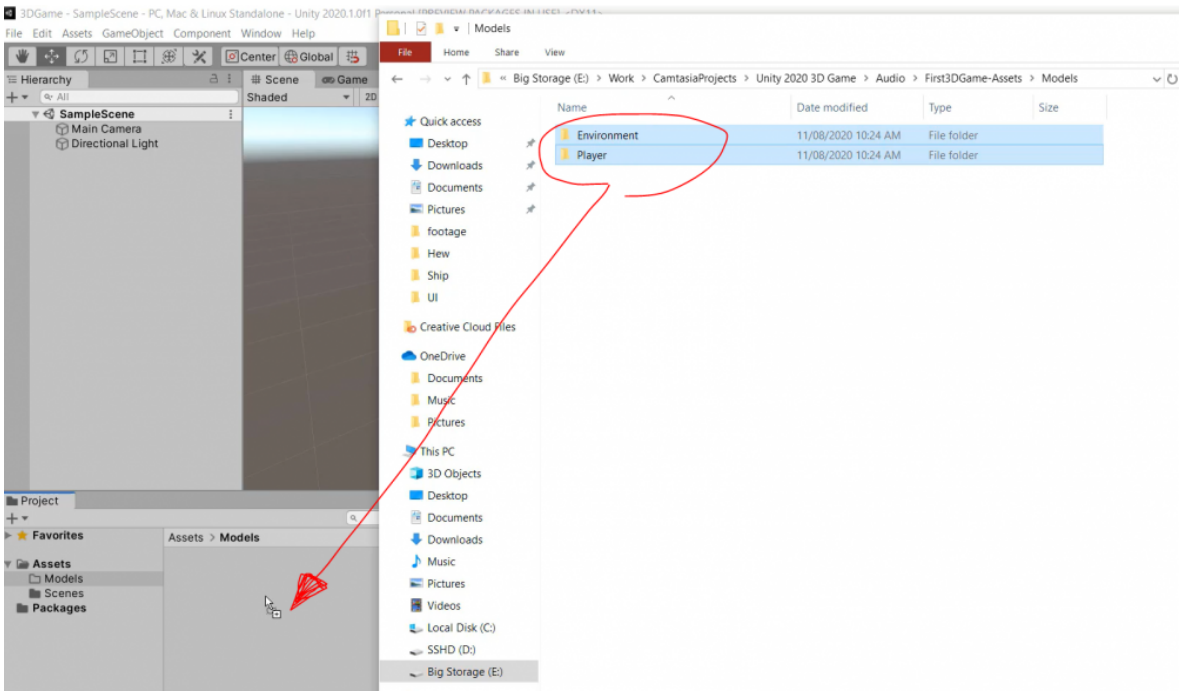


Next, we're going to download the **Assets.zip** file from the **Course Files** tab.



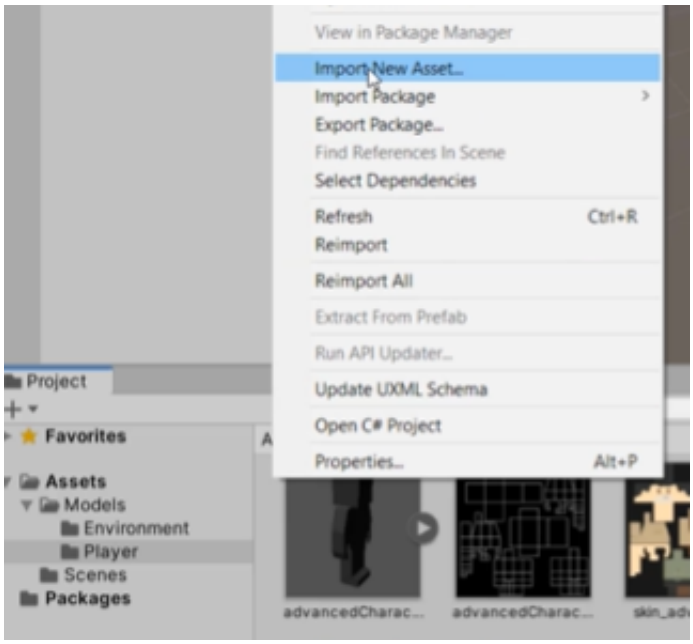


We're going to select the two folders inside the extracted file (**Assets/Models/**) and drag them into our project browser.



Now we've successfully imported the files into our project.

Alternatively, instead of the drag-and-drop method, you can **right-click** in the project window and click on **Import New Asset**.

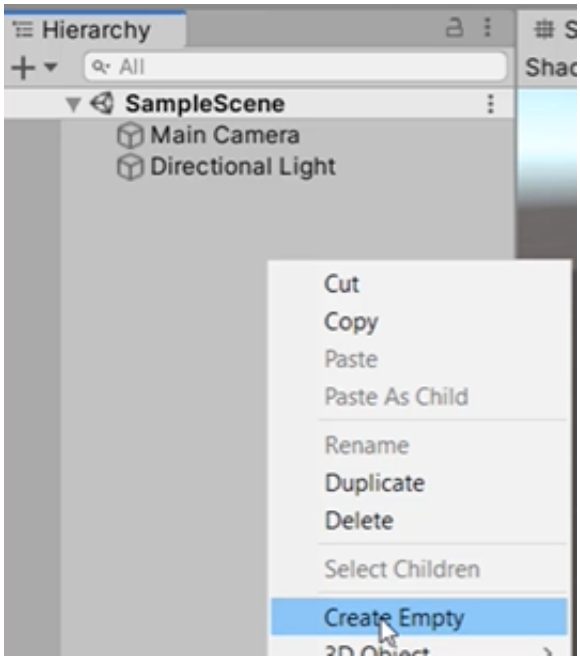


In this lesson, we're going to be creating a Player **GameObject*.**

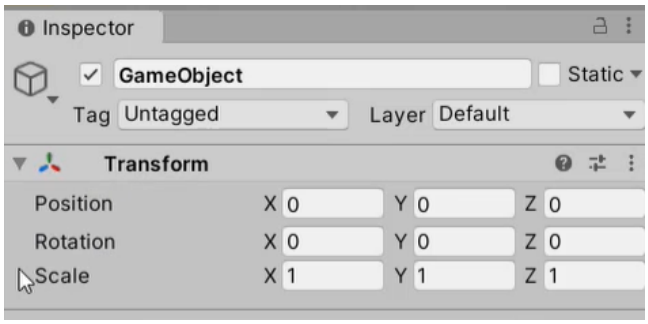
*(A **GameObject** is the base class for all entities in Unity that you can attach components onto.)

Creating A GameObject

We're going to right-click in the **Hierarchy**, and click on **Create Empty**.



This is going to create a brand new GameObject. In the **Inspector** window, you can see that it has a component called **Transform**.



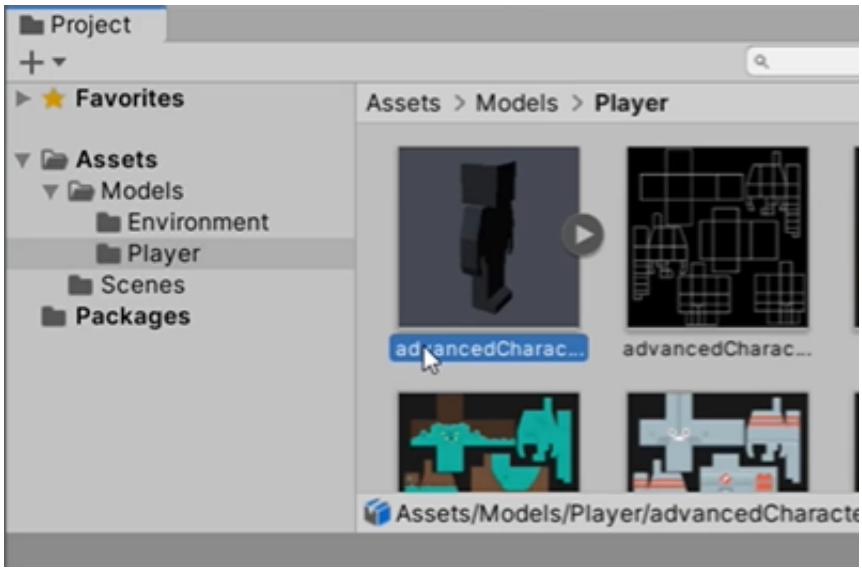
This component defines the **position, rotation** and **scale** of our GameObject in 3D space.

Adding Components (Model)

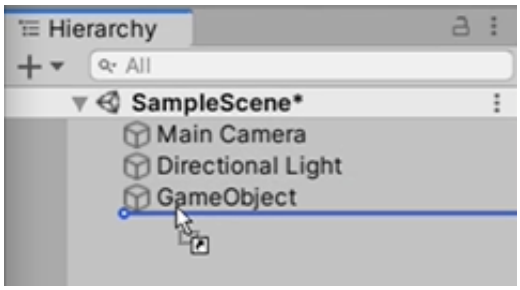
We will add the following components to our Player GameObject:

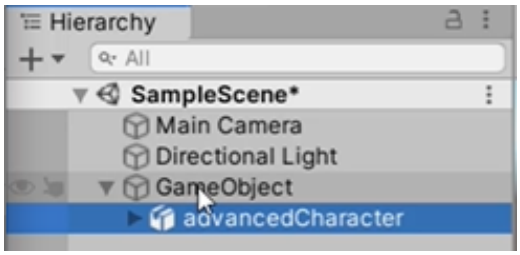
- a 3D model
- a collider
- a script that allows us to jump and collect coins.

For our 3D model, let's go to **Assets > Models > Player:**

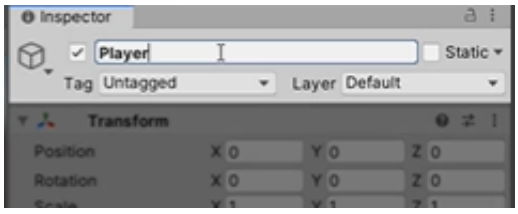


...and drag the model (**advancedCharacter.fbx**) into our GameObject as a child.

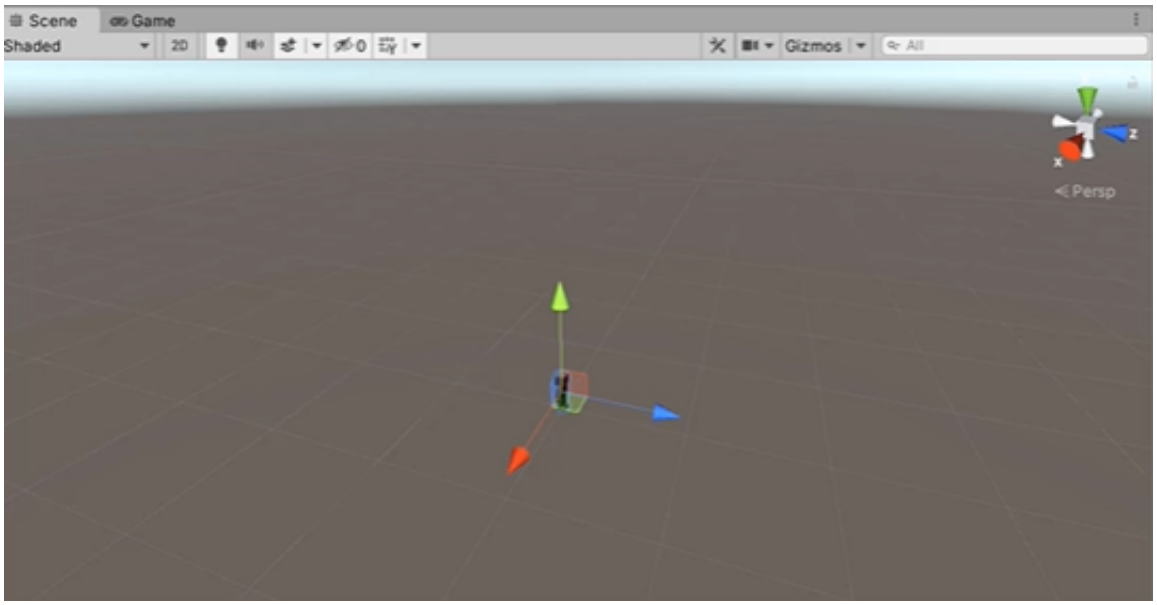




Let's **rename** it to "Player":

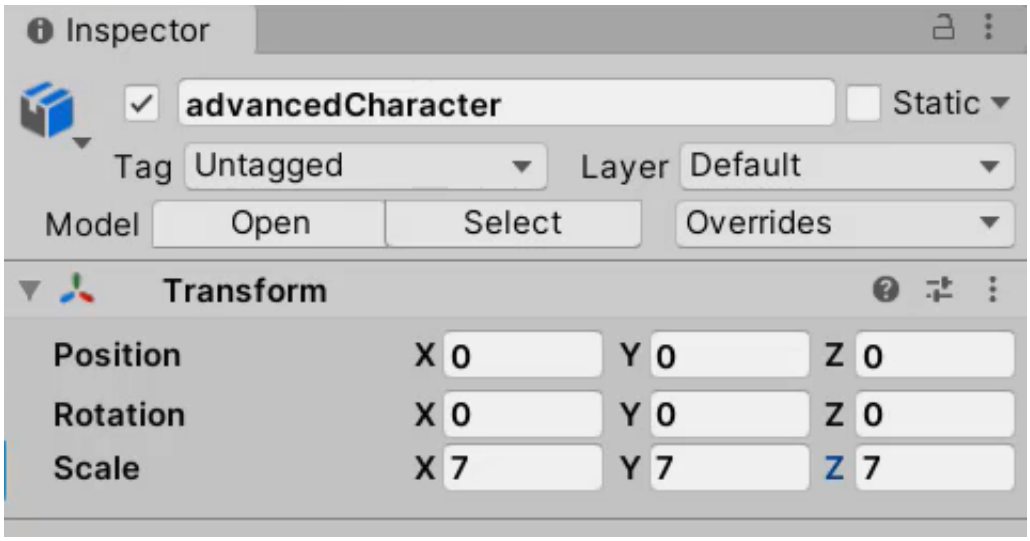


You might have noticed that the current size of the model is too small. (1 Unity unit = 1m)

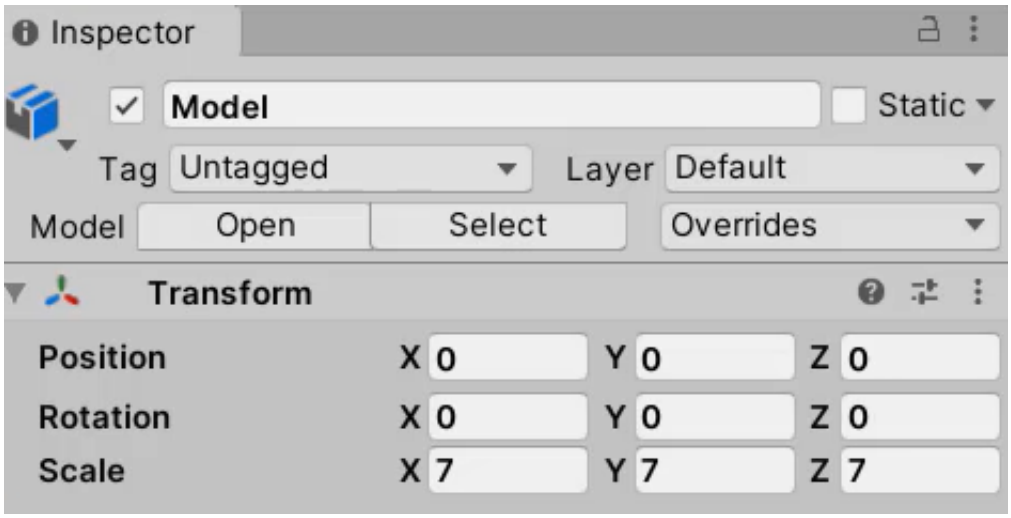


You can press **F** to **zoom in** on the selected GameObject:

Let’s go to the Inspector and change the **Scale** from 1 to 7:



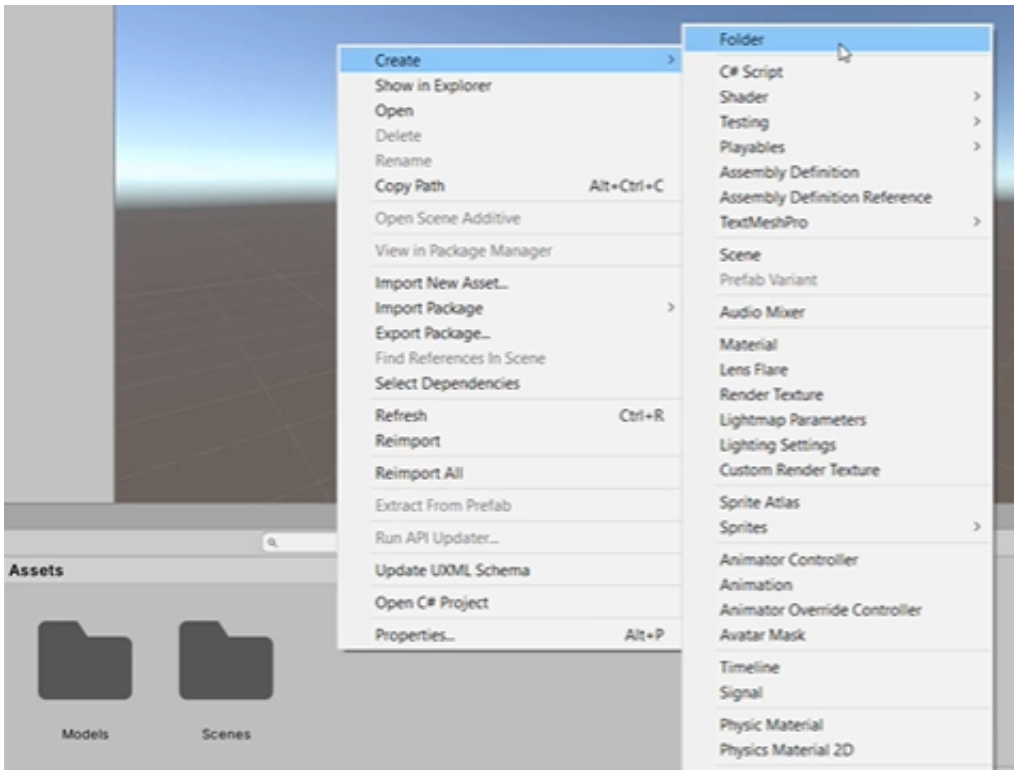
And we're also going to rename it to "Model" while we're at it.



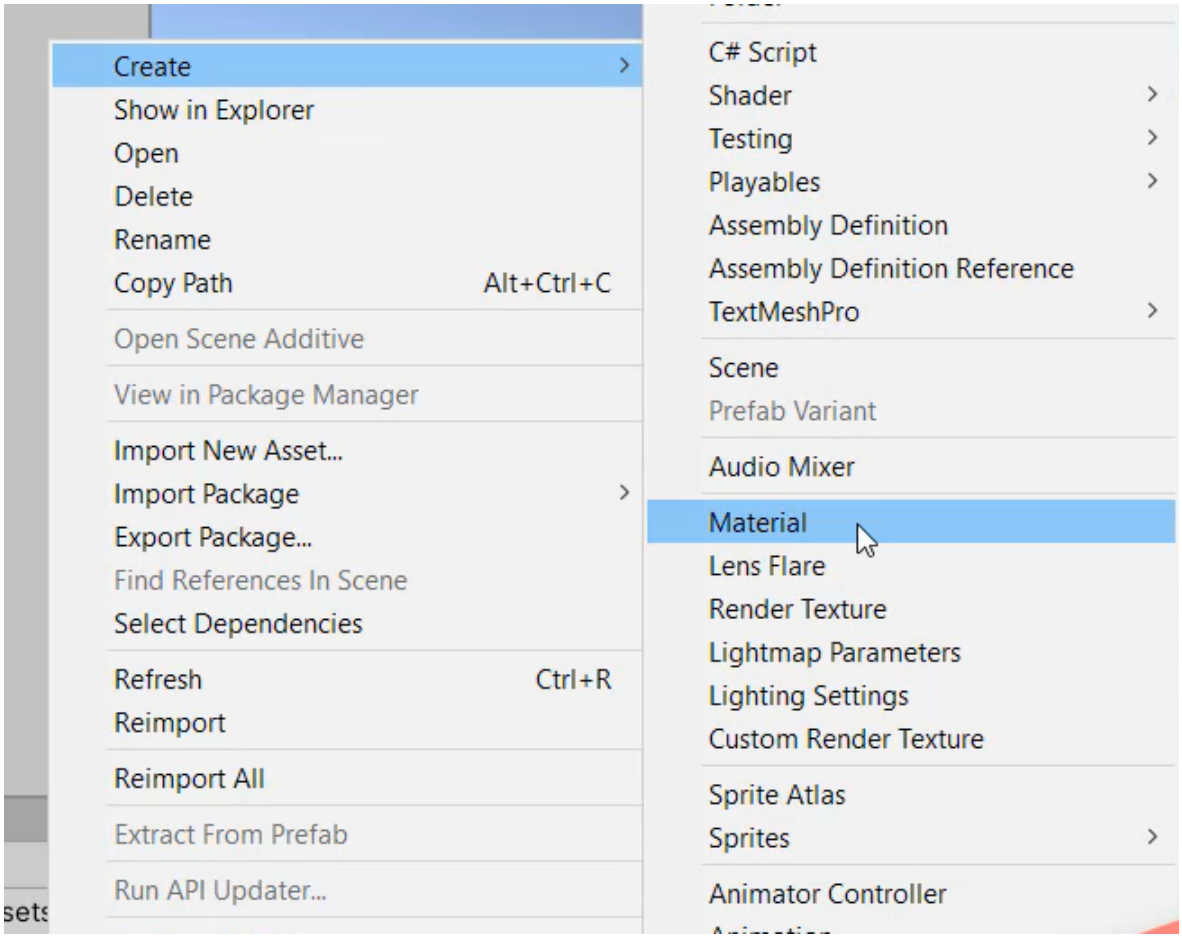
Applying A Texture To 3D Model

Materials define how an object looks in the world.

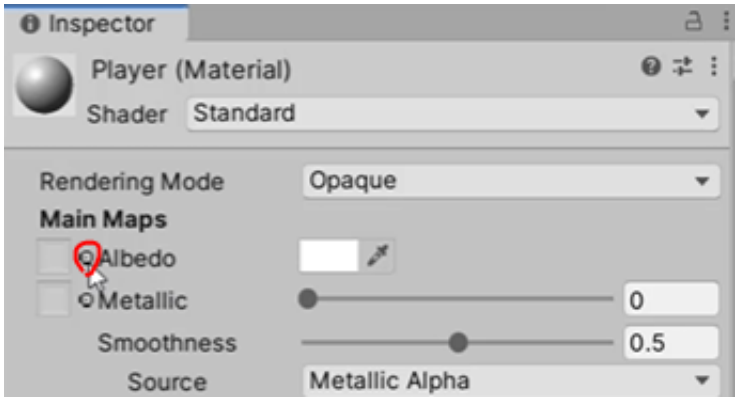
Let's go ahead and create another folder called "Materials" inside the root (Assets) folder.



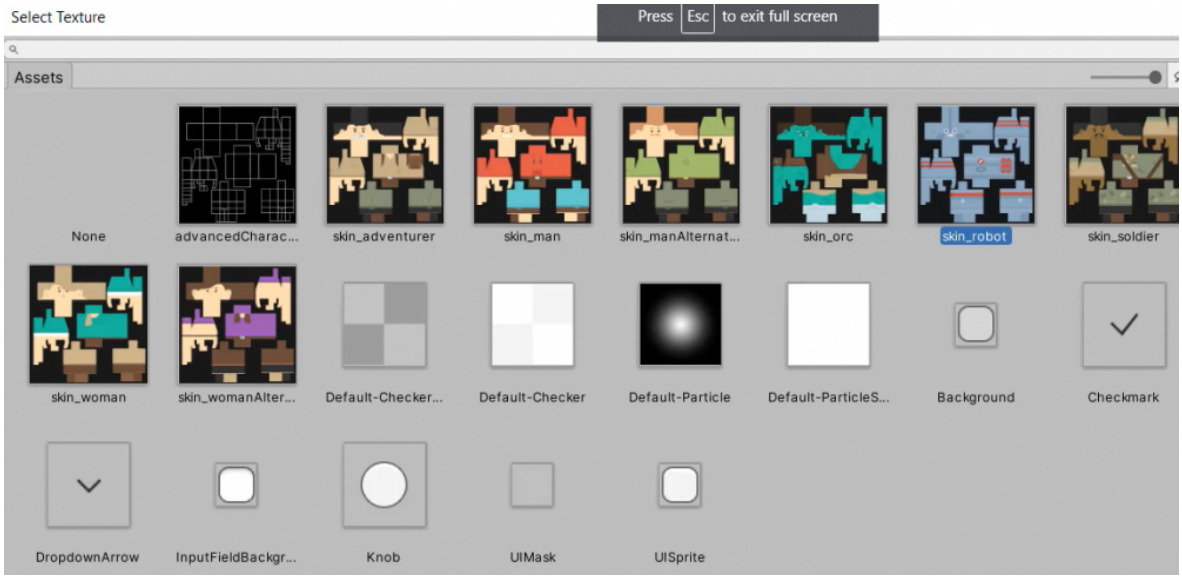
And inside this folder, we're going to **Right-click > Create > Material**:



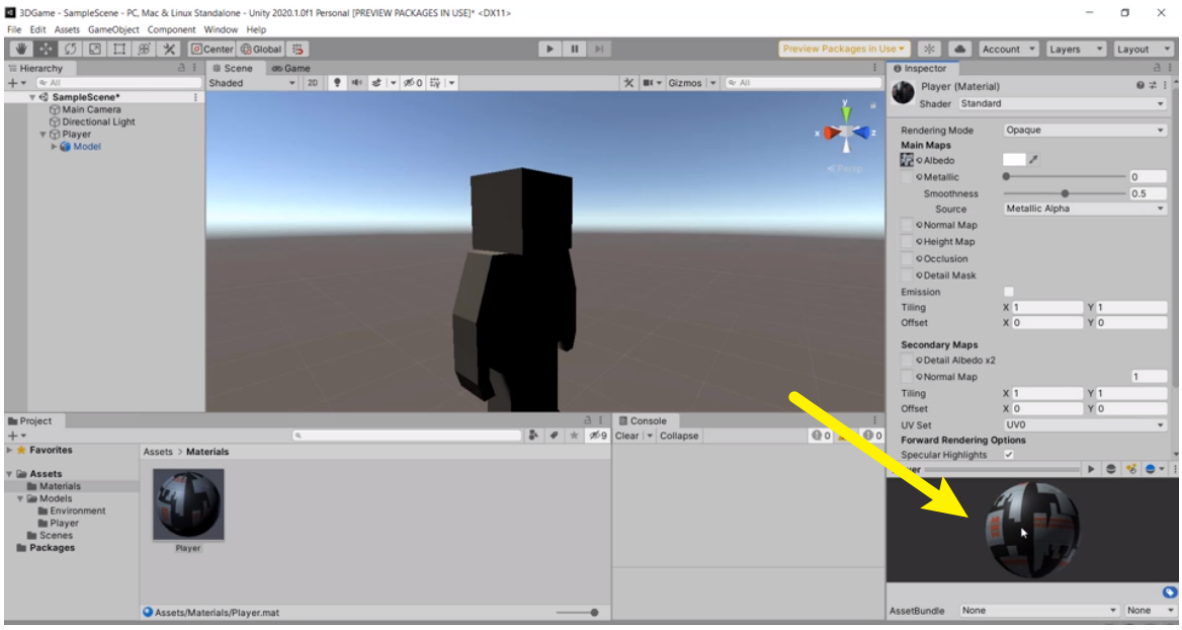
We're going to click on this little circle next to the **Albedo** parameter, which determines the base color of the surface:



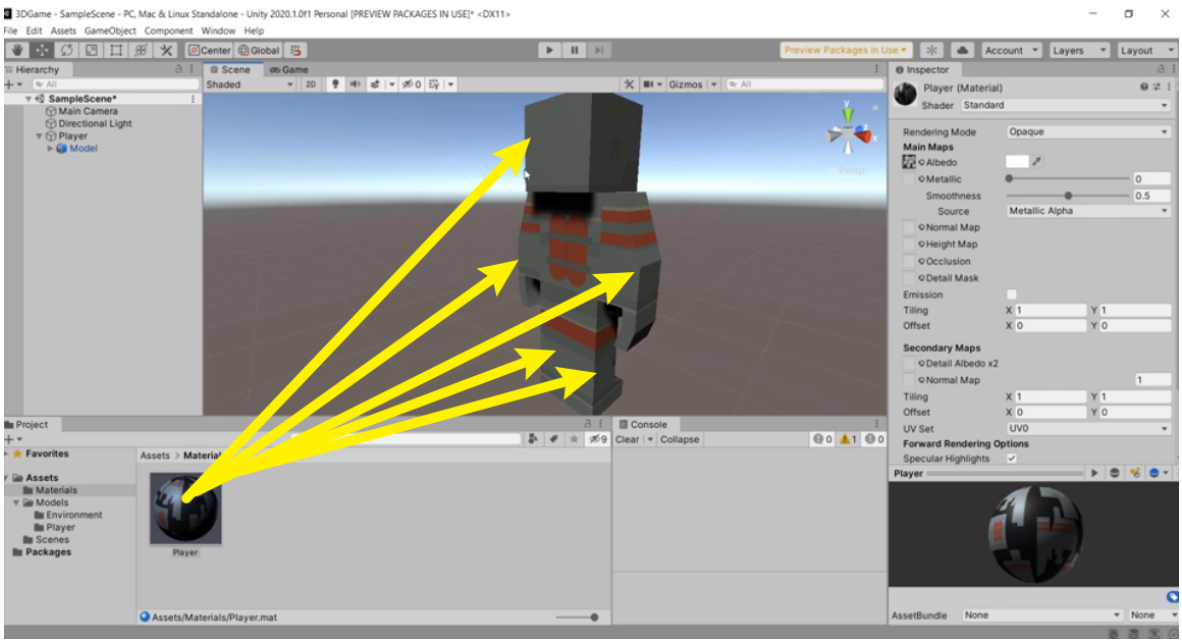
It will pop up the **Texture** window, where we can select a texture to apply to our character. We're going to select **skin_robot**:



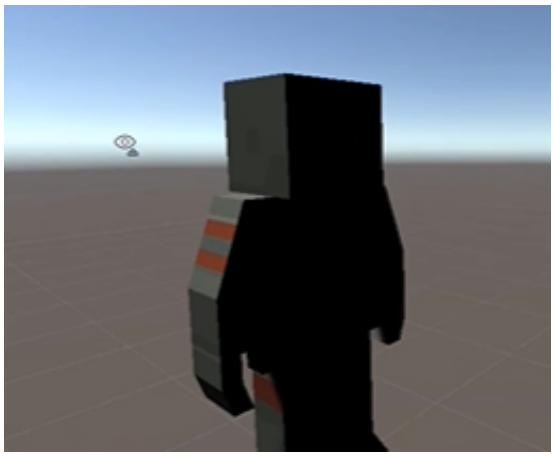
We can now see in the preview window that the texture is applied to this material.



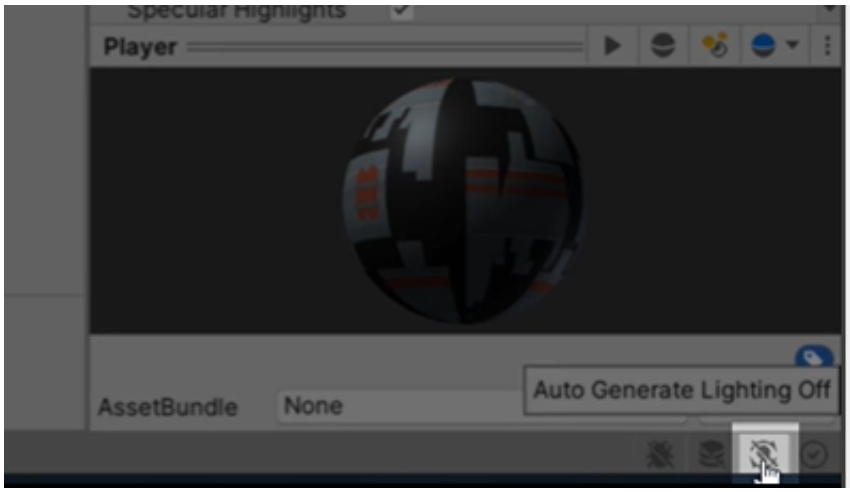
We can then **apply the material** to our model **by dragging** it onto all the different limbs of our Player.

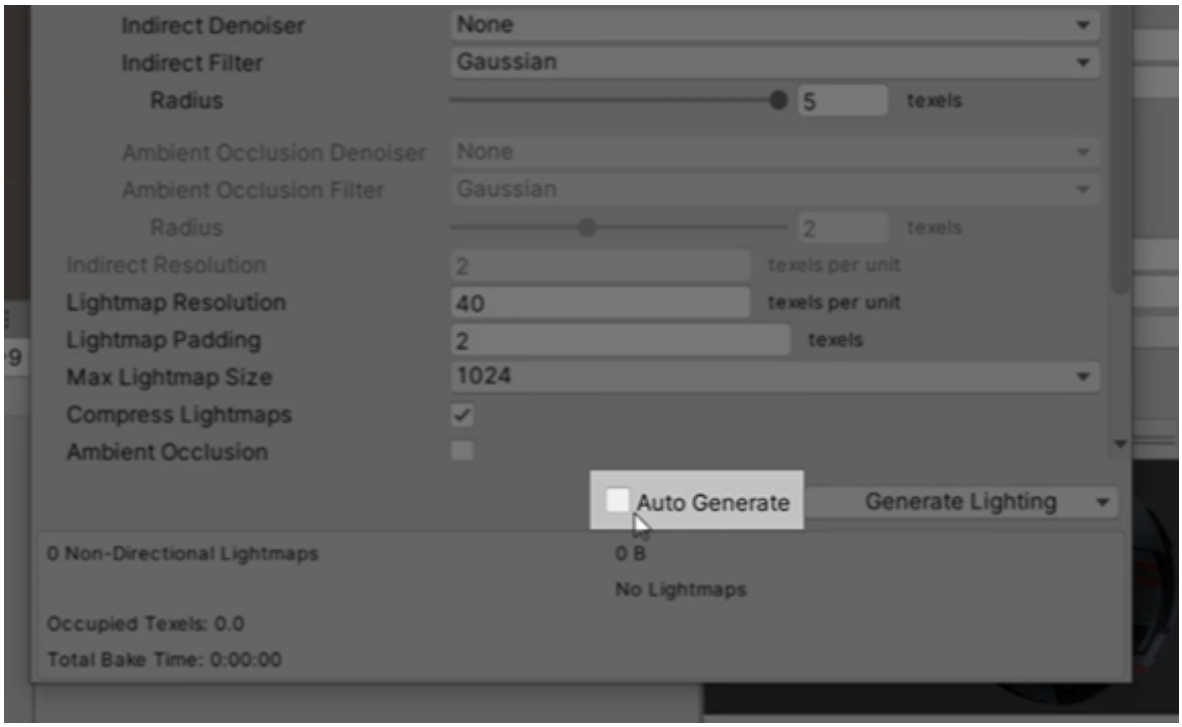


You may also notice that it's pitch-black on one side. This is because the **Auto Generate Lighting** is set to off by default.

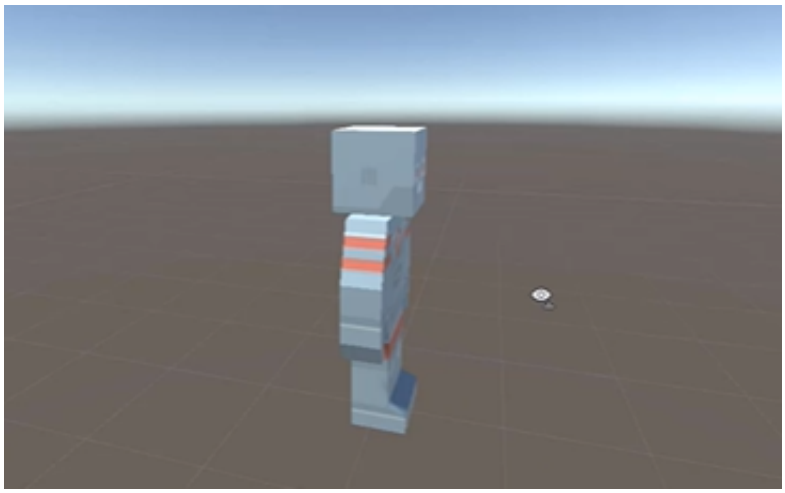


We can enable **Auto Generate Lighting** by clicking on the light bulb icon on the bottom-right of the screen.





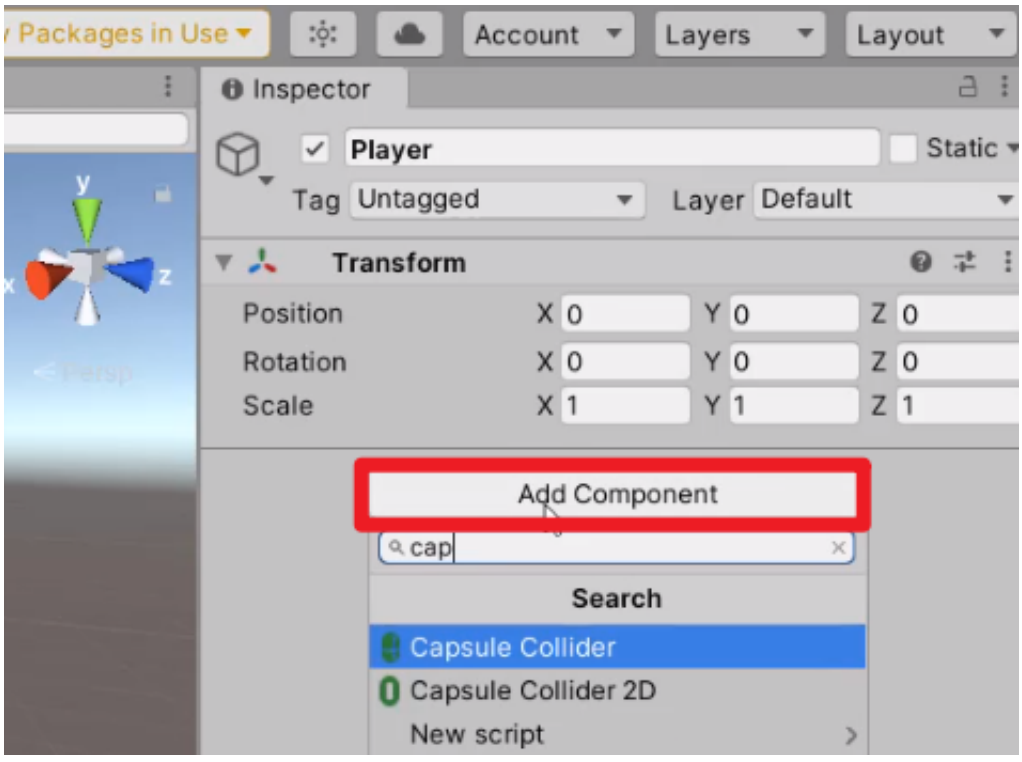
You can see that our model is now lit properly.



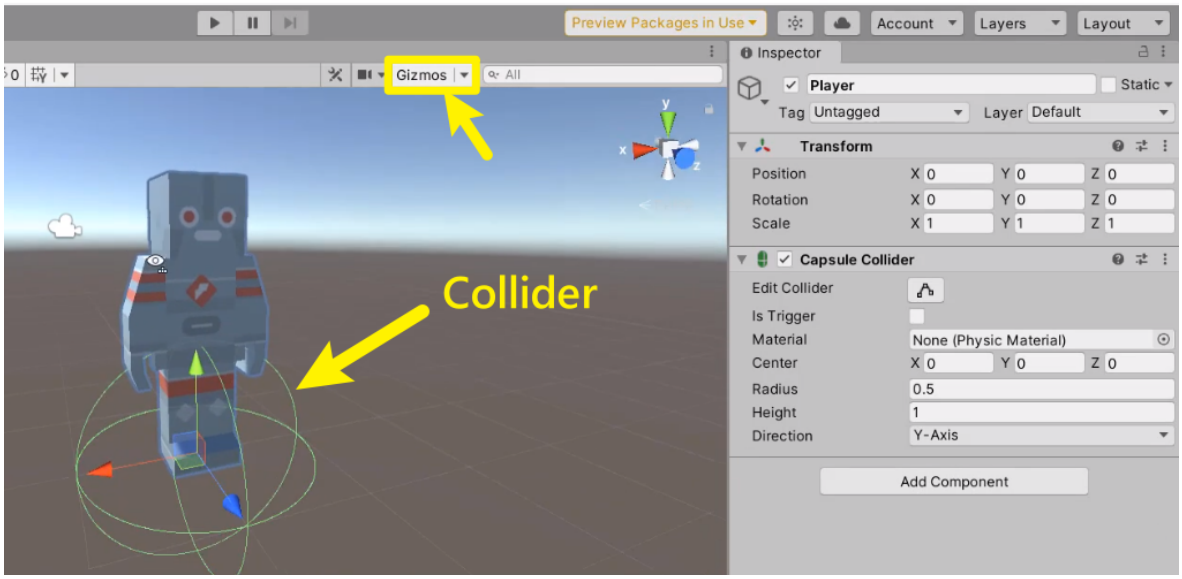
Adding Components (Collider)

Colliders in Unity are components that provide collision detection, which allows our GameObjects to stand on the ground and interact with other objects.

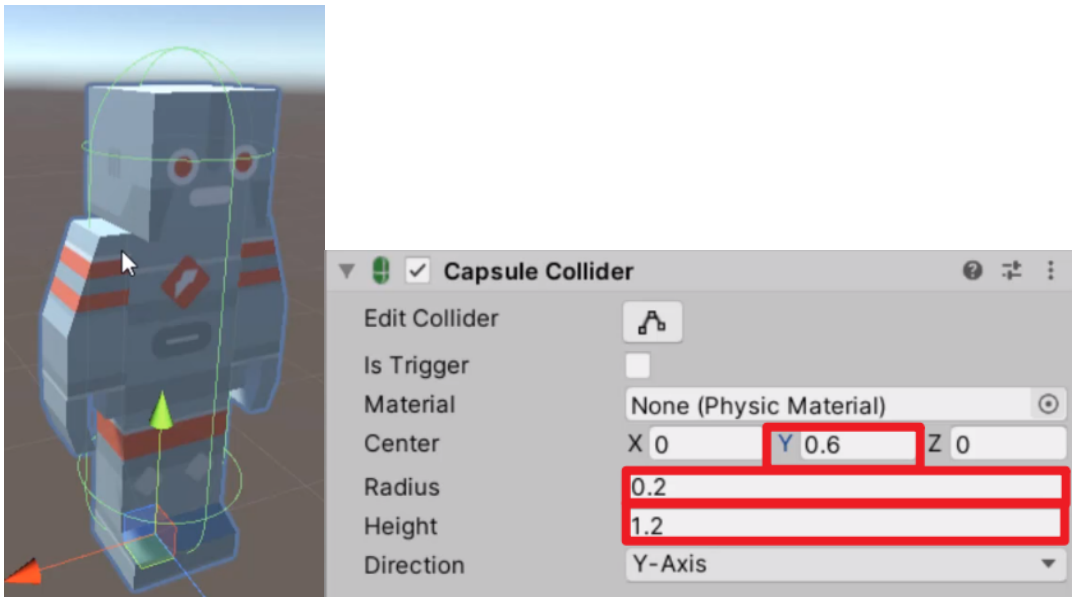
To add a capsule-shaped collider, go to the Inspector and click on **Add Component** > (search) **Capsule Collider**.



Make sure that the '**Gizmos**' button is enabled so you can see the collider (green lines) in the Scene view.



We're going to set the **radius** to **0.2**, the **height** to **1.2**, and the **y-center** value to **0.6**, so it matches the shape of our Player model.

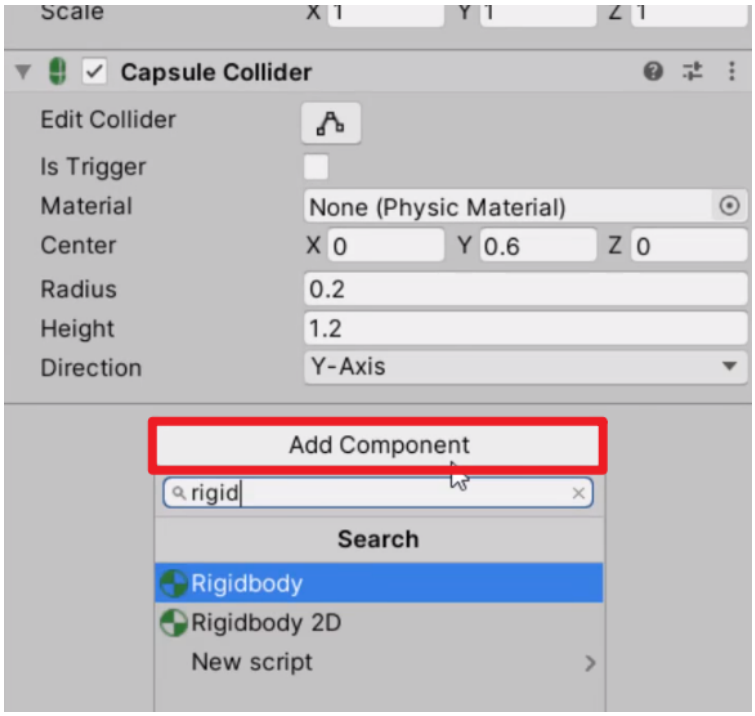


Adding Components (Rigidbody)

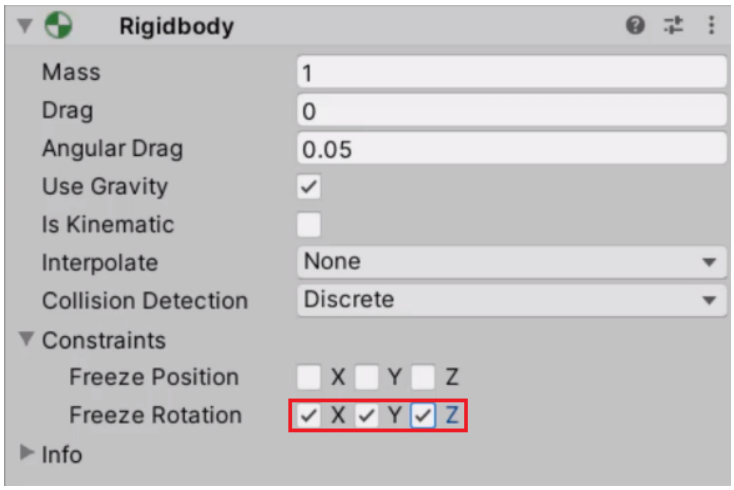
If you want the collider to work, we need a **Rigidbody** component.

Rigidbody is basically a component that applies physics such as gravity, drag, mass, etc.

We're going to click on **Add Component >** (search) **Rigidbody**.



Lastly, we're going to enable everything inside of **Freeze Rotation** inside the **Constraints** drop-down, so our object won't tip over and fall on its side.

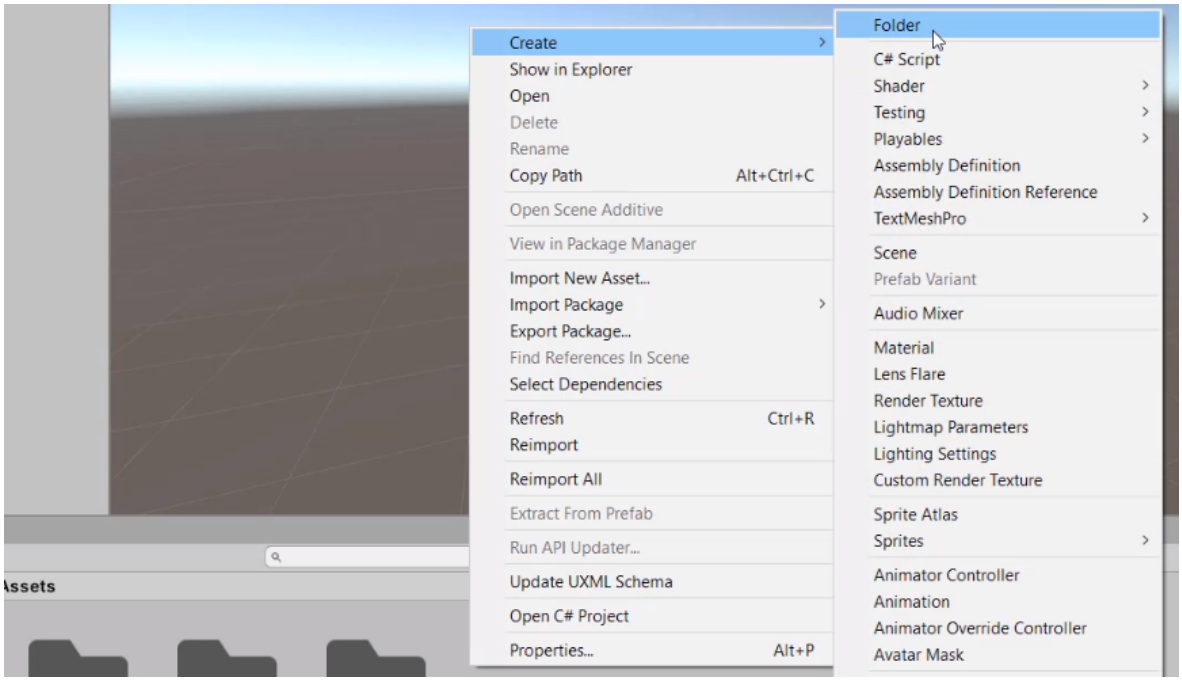


In this lesson, we're going to begin setting up our player movement.

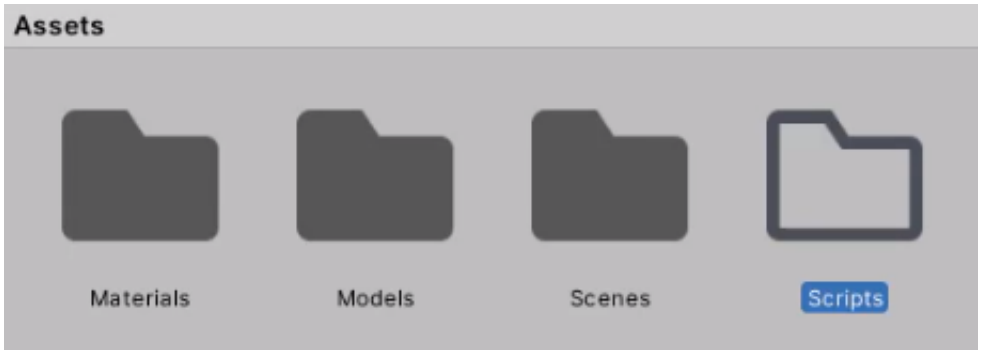
Creating A New Script

Scripts are custom components that can be attached to GameObjects. We're going to be assigning a logic to our player using a C# script in order to get them moving around and jumping.

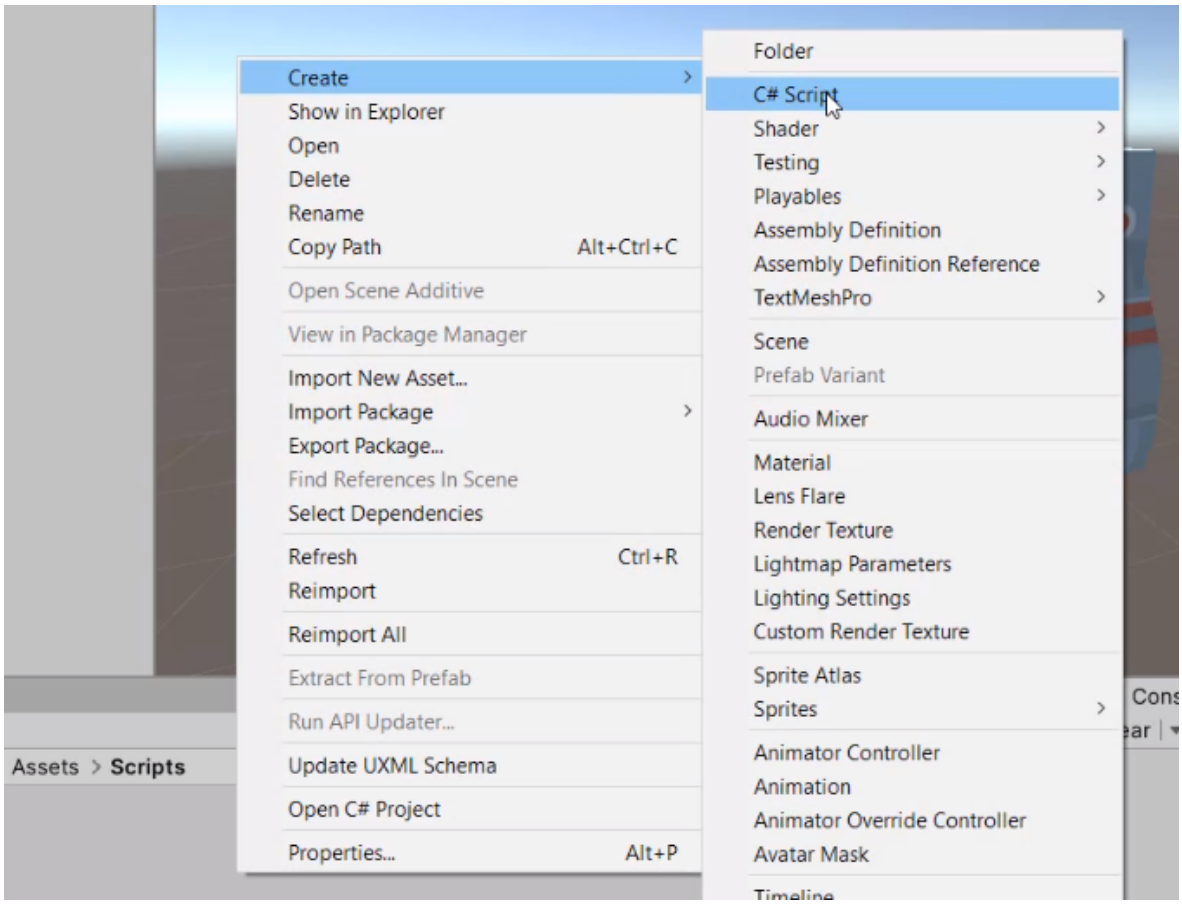
First of all, we're going to **create a folder** by right-clicking in the **Project window > Create > Folder**.



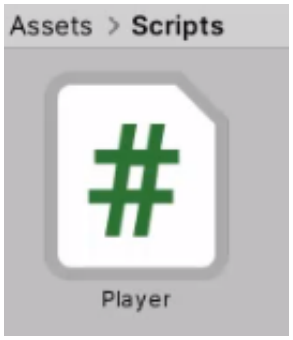
We're going to call this folder "Scripts".



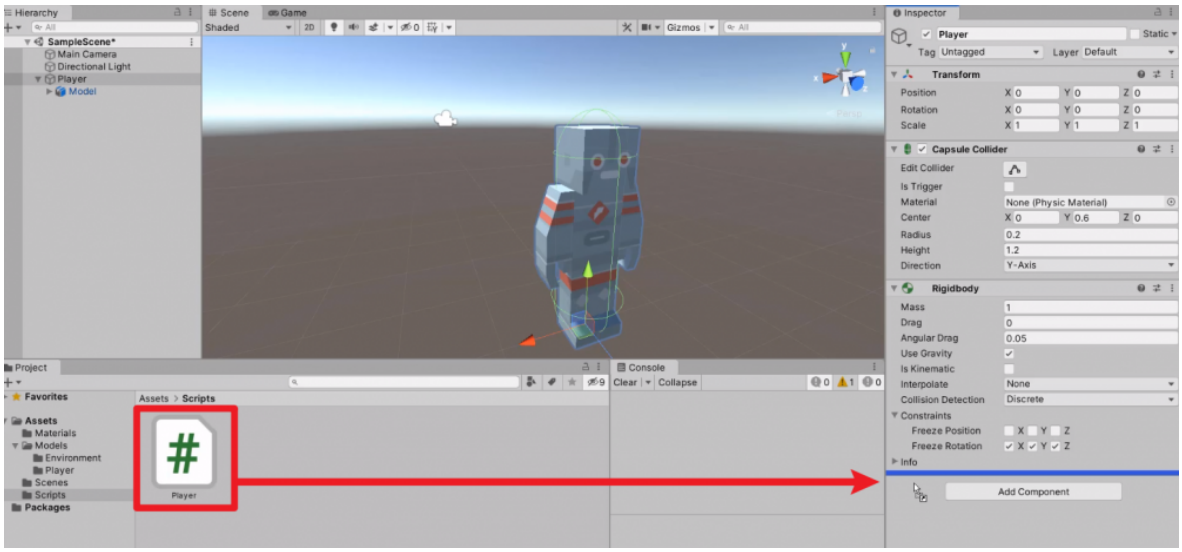
Then, inside the Scripts folder, we're going to create a new script by going to **Create > C# Script**.



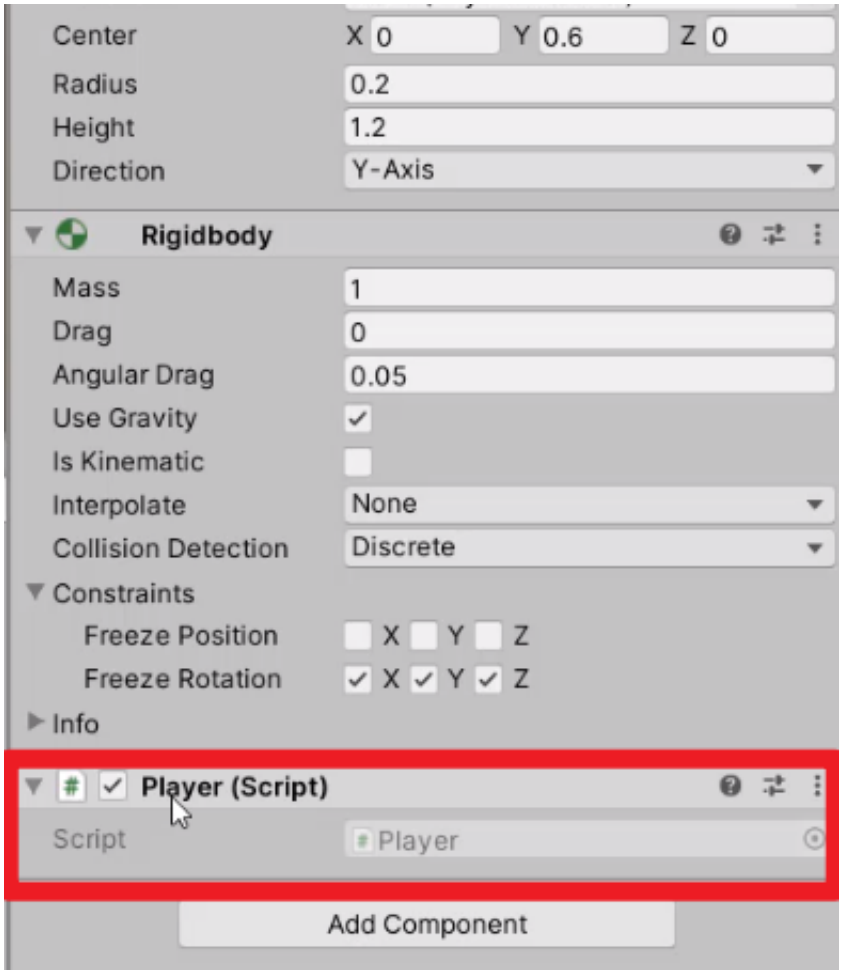
We're going to **rename** the script to be 'Player'.



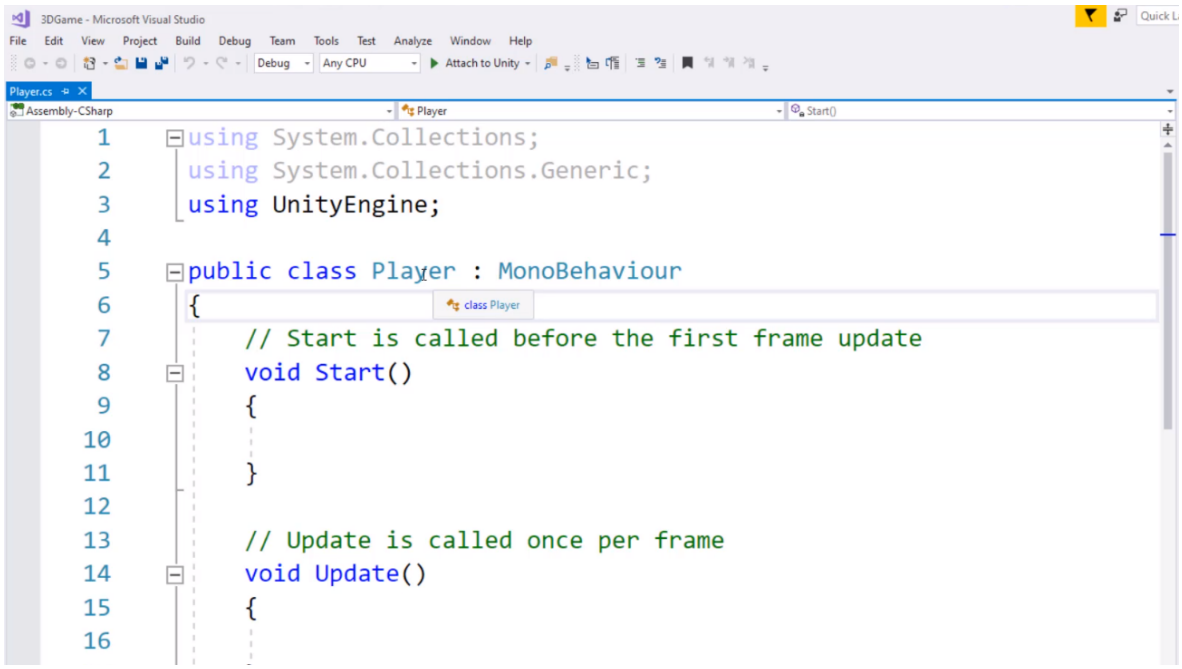
Then, we're going to attach the script to our Player gameObject by **dragging** it onto the **Inspector**.



Now we can see that the Player script is **attached** to our Player GameObject.



Let's **double-click** on the script and open it up in Visual Studio. (Click here to download Visual Studio if you haven't already: [https://visualstudio.microsoft.com/downloads/)](https://visualstudio.microsoft.com/downloads/)



Assigning Player Movement In C#

We now have the script open inside Visual Studio with the default template setup.

The default template comes with two pre-built functions: **Start()** and **Update()**.

```
using UnityEngine;
using System.Collections;

public class MyCustomScript : MonoBehaviour {

 // Use this for initialization
 void Start () {

 }

 // Update is called once per frame
 void Update () {

 }
}
```

The **Update** function gets called every single frame, whereas the **Start** function gets called only once before the first frame is loaded.

Detecting Input

Inside the Update function, we're going to detect whenever a button on a keyboard has been pressed down.

Since we're moving horizontally along the x-axis, we're only going to detect keys assigned for **horizontal** movement (i.e. A, D, ← and →).

For this, we're going to be using **Input.GetAxis()**, which is a pre-built function in UnityEngine that will return a value from -1 to 1, based on keypress on a specified axis (e.g. "Horizontal").

```
public class Player : MonoBehaviour
{
  void Update()
  {
     //Detecting inputs for horizontal movement
     float x = Input.GetAxis("Horizontal");
  }
}
```

By doing this, we're creating a **float** variable **x** every frame, which will have a value of **-1** if the user is pressing ←, **0** if no input is detected, or **1** if the user is pressing →.

Modifying Velocity

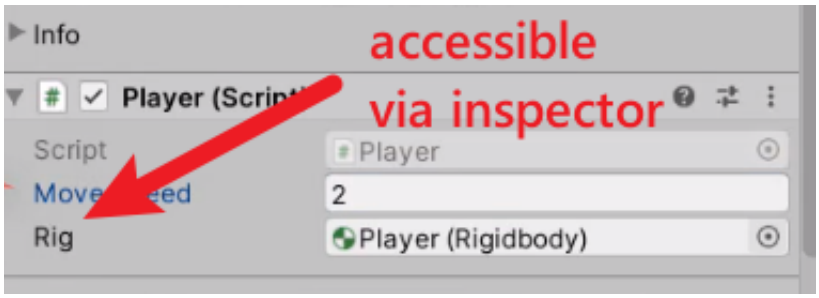
Now we're going to modify our Rigidbody component's **velocity**, which represents our speed and direction as **Vector3(x, y, z)**.

```
public class Player : MonoBehaviour
{
  public float moveSpeed;
  public Rigidbody rig;

  void Update()
  {
     //Detecting inputs for horizontal movement
     float x = Input.GetAxis("Horizontal");

     //Applying the horizontal movement & gravity to the rigidbody's velocity
     rig.velocity = new Vector3(x, rig.velocity.y, 0);
  }

}
```

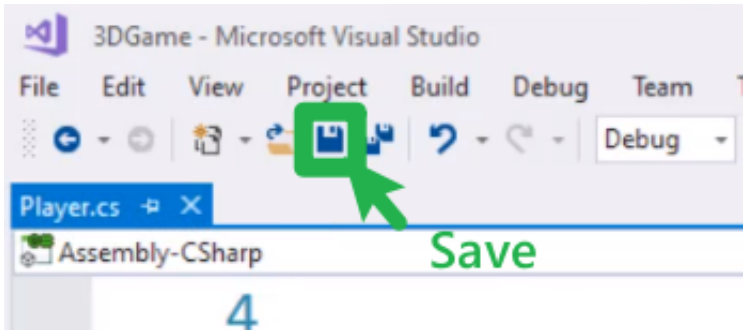
Note that we've created a custom variable "rig" above our Update function as a **public** variable. Unlike a private variable, a public variable can be accessed by other things outside of the script.

```
public Rigidbody rig;
```

This allows us to easily reference any component in the Unity Inspector, or input our own custom value without changing the actual script.

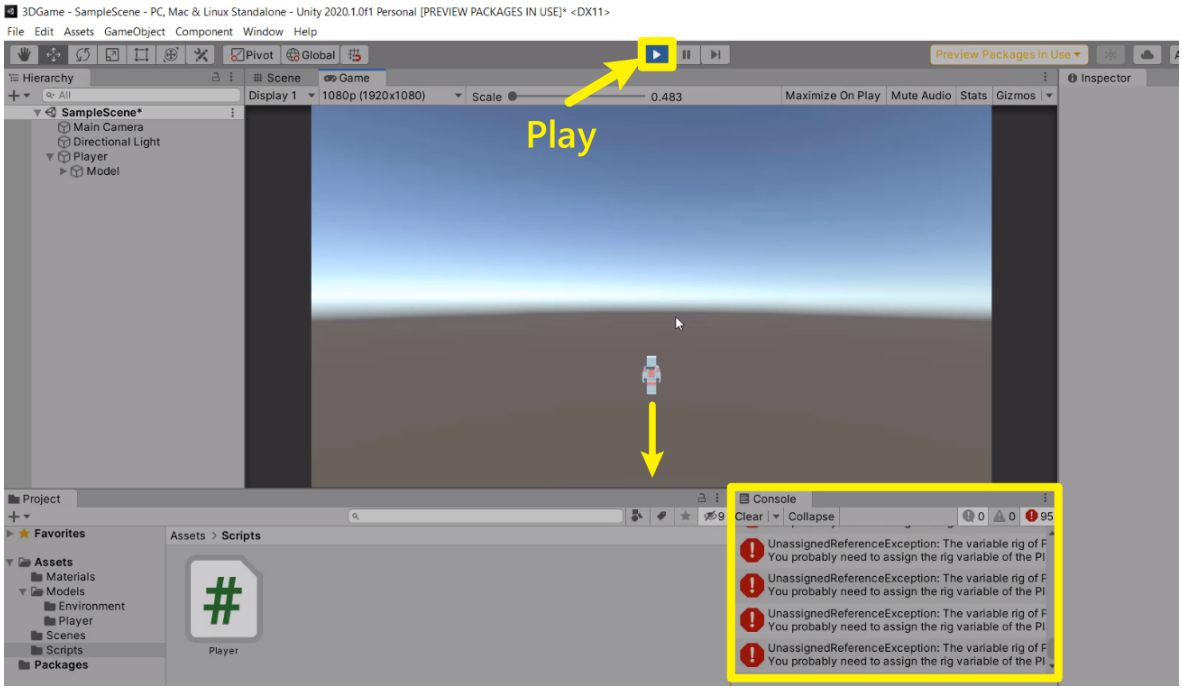


After that, we can hit **Save** (Ctrl+S or ⌘+ S) and move on to the next lesson.

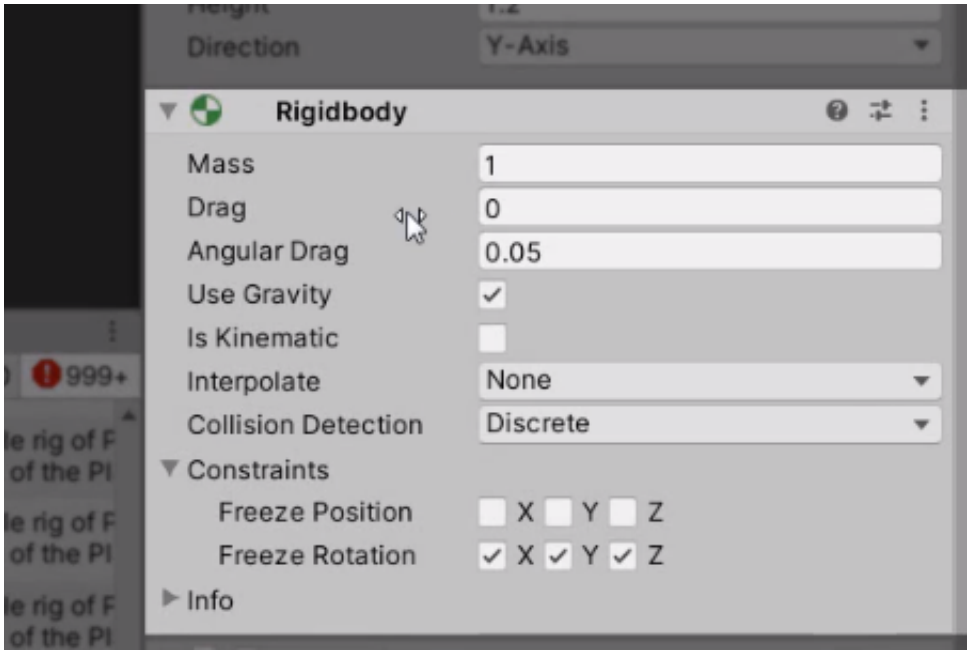


In the previous lesson, we've set up player movement along the horizontal axis.

When we press **Play**, you'll see that our player falls straight down, and we have errors appearing in the console.

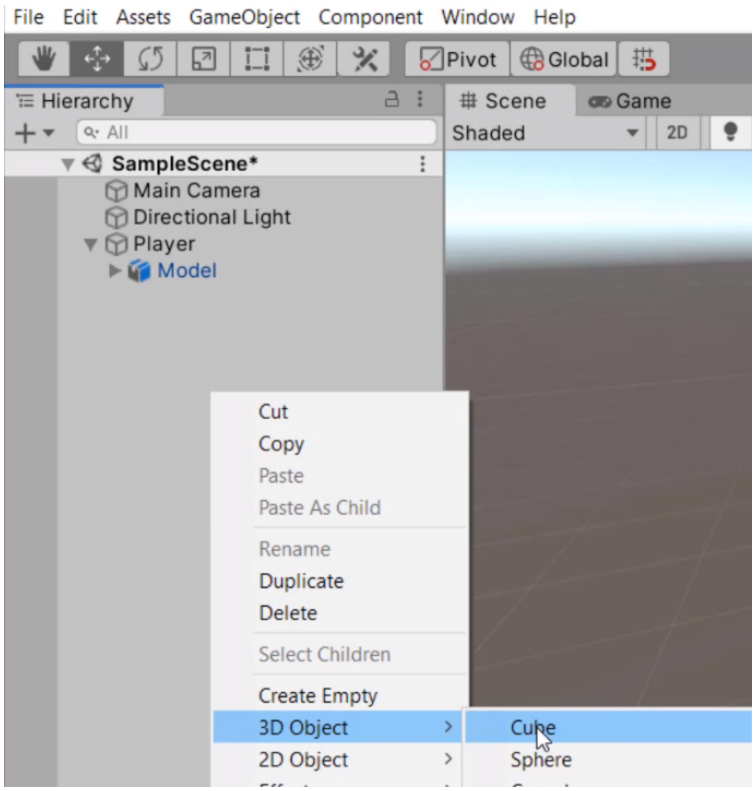


Our player is falling down due to the **Rigidbody** component that we have attached to the game object, which is causing gravity to pull this player downwards.

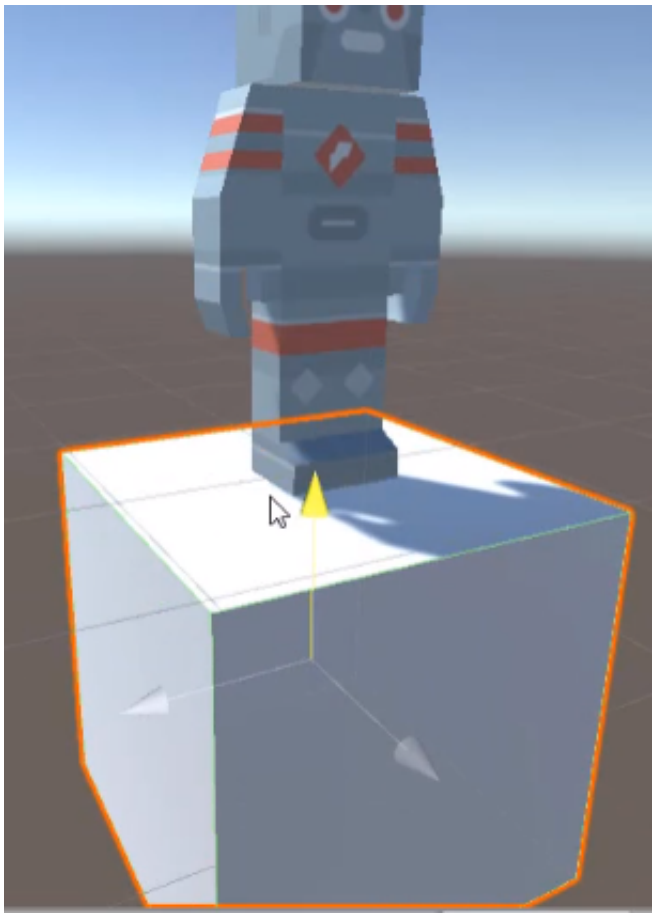


Creating A Ground

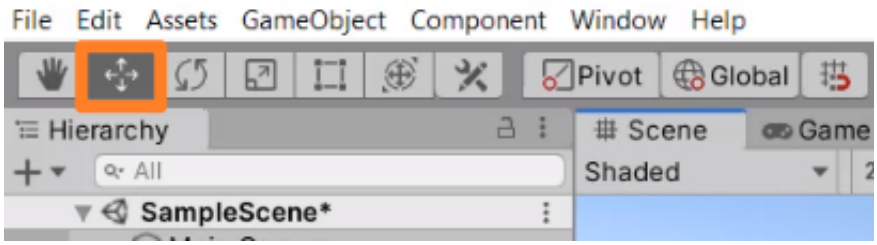
To stop the player from falling down, we're going to **create a platform** under the player. (**Right-click > 3D Object > Cube**)



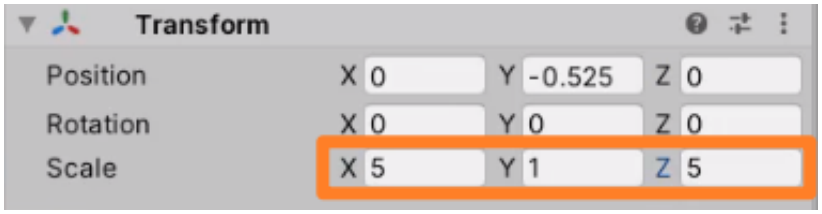
We're going to move the cube down a bit using the **Move** tool.

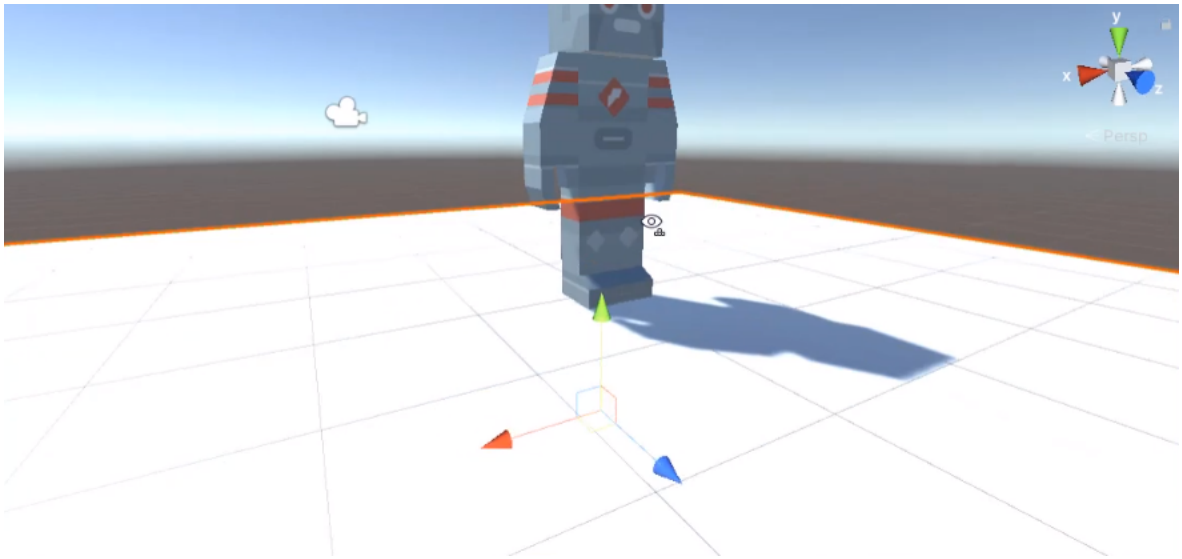


(You can select the **Move** tool by pressing M on the keyboard, or by clicking the arrow icon in the toolbar.)

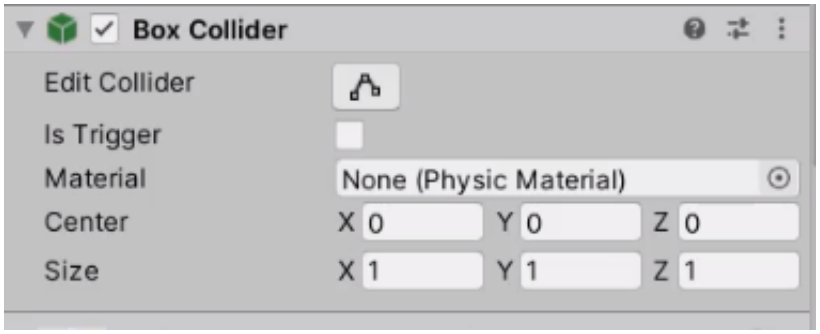


We're also going to set the **Scale** to be (5, 1, 5).

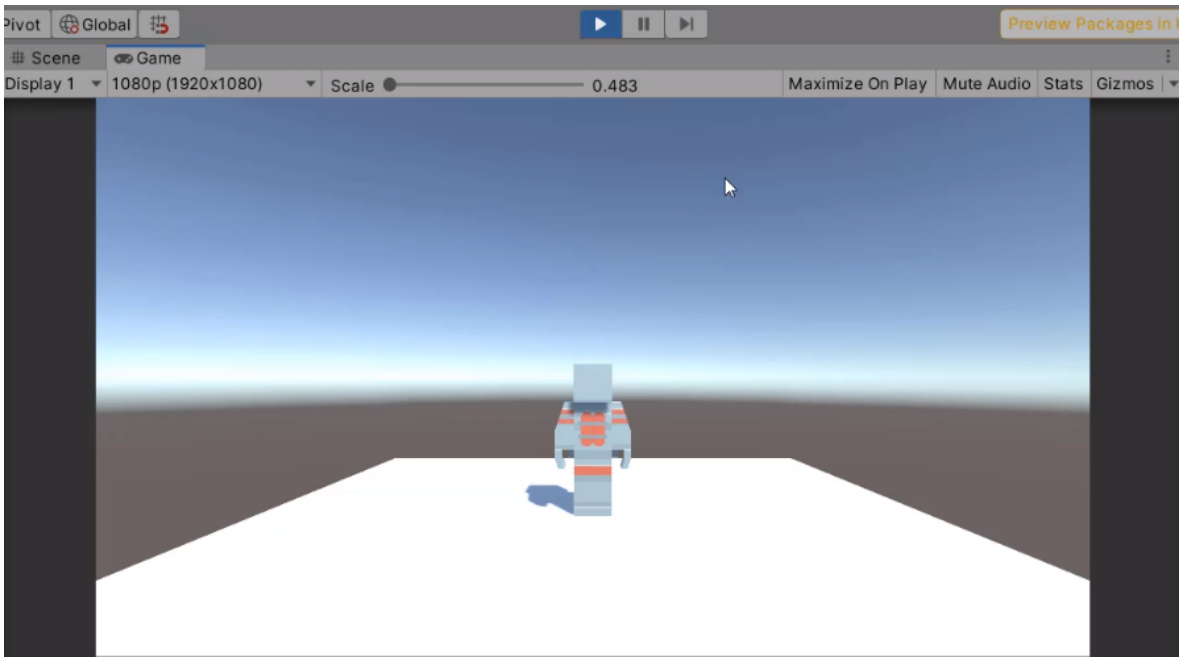




Note that there's a **Box Collider** component attached to the cube. This allows the rigidbody component of our player gameObject to recognize incoming collisions.

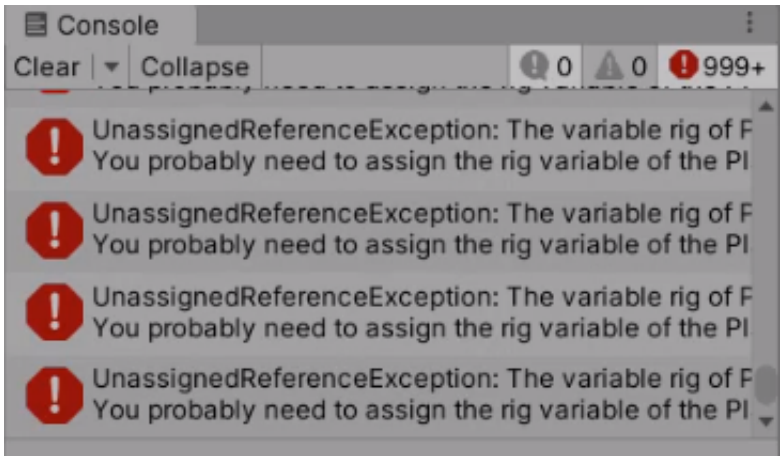


When we hit Play, you'll see that our player can stand on top of the cube.

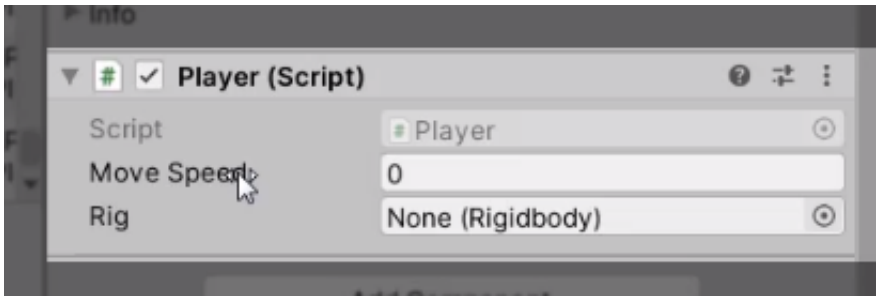


Fixing Error: Unassigned Reference Exception

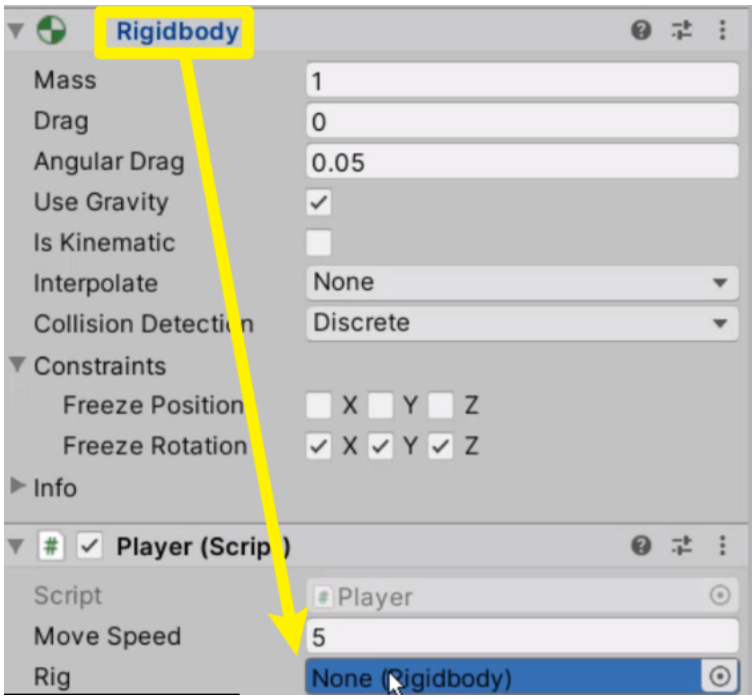
Now, we're going to fix this error: 'UnassignedReferenceException: The variable **rig** of Player has not been assigned.'



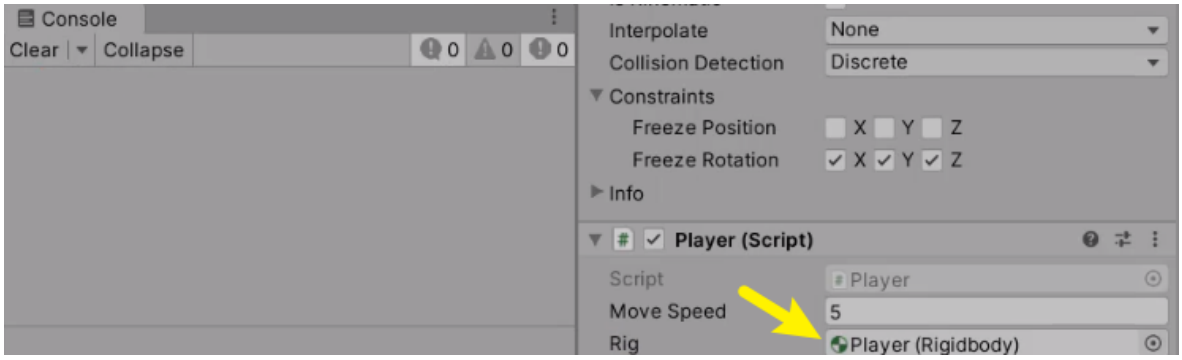
When we select the Player Gameobject, you'll see that our **Move speed** and **Rig** variables, which we created and set to public in the previous lesson.



We're going to set the **move speed** value to 5, and drag our player's **Rigidbody** component into the **Rig** field.



Now when you press ‘Play’, you should see that no more errors are appearing.



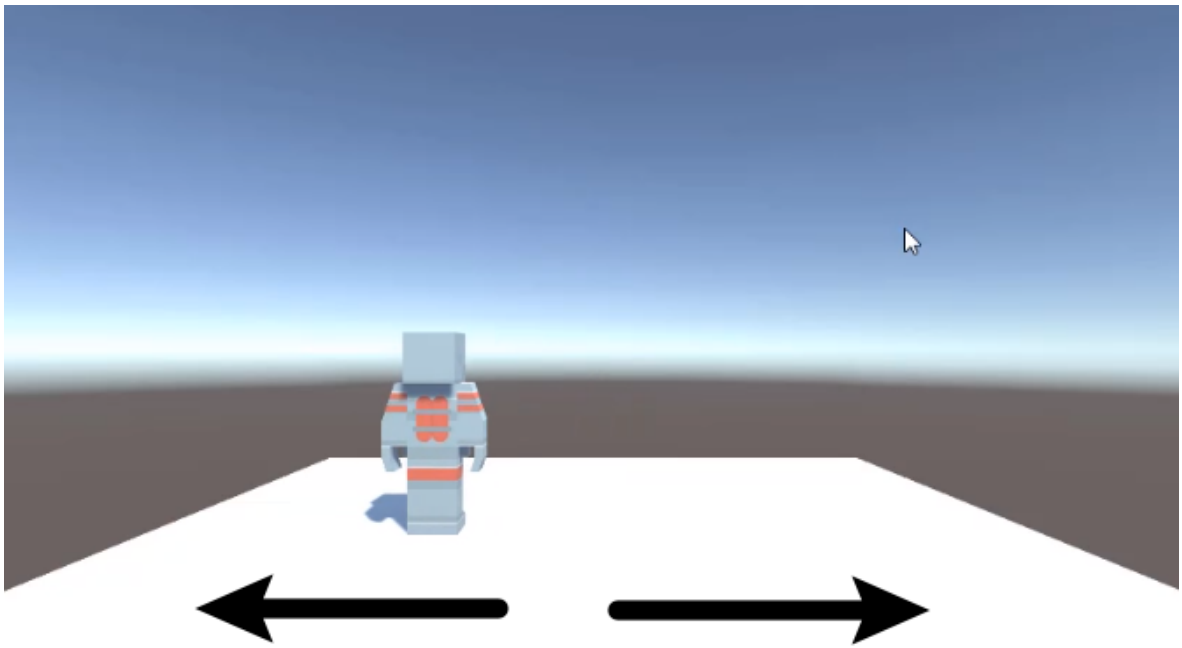
Applying Move Speed To Input

Currently, the variable **moveSpeed** that we created is not actually used anywhere in the script.

To apply our move speed, we’re going to multiply it with the horizontal input axis:

```
float x = Input.GetAxis("Horizontal") * moveSpeed;
```

We can move our player with arrow keys (or A and D) now. Feel free to adjust the speed in the inspector and test it out while in Play mode.



Note that any changes made to GameObjects while in Play mode are not saved.

Challenge: Implement Moving Forward and Backward

Let's try and implement moving forward and backward movement!

Remember, we've created a local variable (x) using the input axis of "Horizontal", and assigned that to the rigid body's velocity.

Hint: Input.GetAxis("Vertical") will return a value in the range between -1 and 1 based on the vertical input axis (e.g. ↑, ↓, W or S)

Solutions:

- Create a new local variable z;

```
void Update() {
...

    //Detecting inputs for vertical movement
    float z =
```

- Get the vertical input axis;

```
void Update() {
...
```



```
    //Detecting inputs for vertical movement
    float z = Input.GetAxis("Vertical");
```

- Multiply it by the move speed variable;

```
void Update() {
...

    //Detecting inputs for vertical movement
    float z = Input.GetAxis("Vertical") * moveSpeed;
```

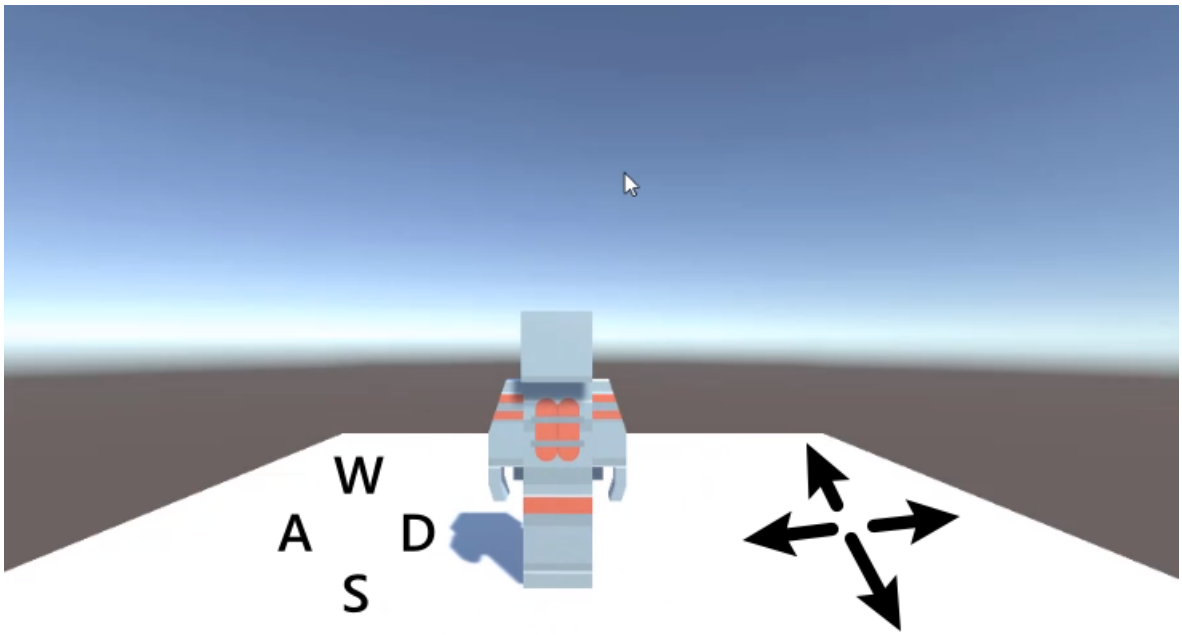
- Replace the 0 inside the new Vector3 value with the z variable.

```
void Update()
{

...
   //Detecting inputs for vertical movement
   float z = Input.GetAxis("Vertical") * moveSpeed;

   //Applying the horizontal movement & gravity to the rigidbody's velocity
   rig.velocity = new Vector3(x, rig.velocity.y, 0);
}
```

Result:

```
public class Player : MonoBehaviour
{
  public Rigidbody rig;

  void Update()
  {
     //Detecting inputs for horizontal movement
     float x = Input.GetAxis("Horizontal") * moveSpeed;

     //Detecting inputs for vertical movement
     float z = Input.GetAxis("Vertical") * moveSpeed;

     //Applying the horizontal movement & gravity to the rigidbody's velocity
     rig.velocity = new Vector3(x, rig.velocity.y, z);
  }
}
```

Now, if you press Play, you should be able to move the player in all four directions with the **W,A,S, D** keys or the arrow keys.



However, our player is always facing the same direction.

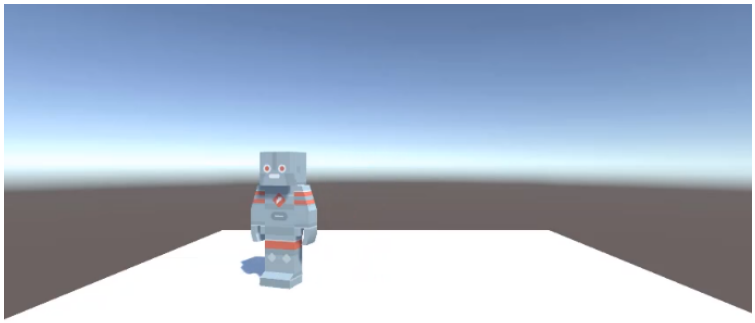
Facing Direction of Movement

We're going to make our player rotate to face the direction that they're moving in.

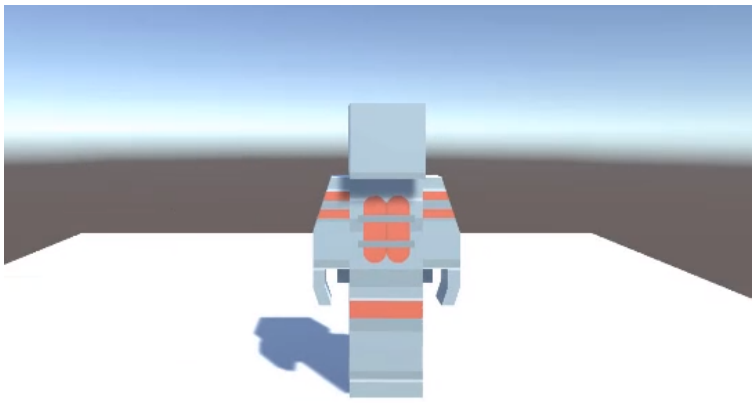
First of all, we're going to fetch the **forward** direction from our player's transform component:

```
transform.forward = rig.velocity;
```

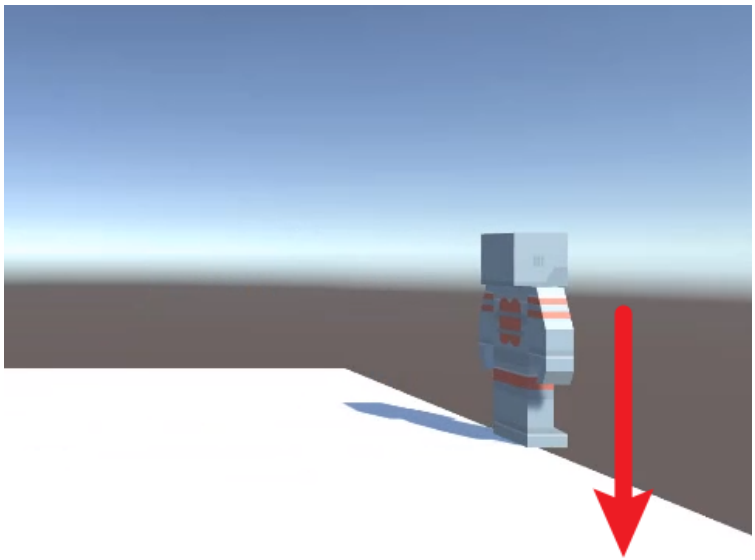
Note that transform.forward is in a form of **Vector3**, so we can assign our rigid body's velocity to it.



Now our player's looking at the direction they're moving in, but their forward direction will snap back to the default as soon as we let go of our button.



They'll also face downwards if they fall off the edge.



To fix this, we're going to create a temporary variable **vel** to store a copy of our velocity Vector3.

```
Vector 3 vel = rig.velocity;
```

This allows us to modify our velocity without changing the original.

We're also going to **fix the y velocity to 0** because we don't want to be facing downwards when we fall.

```
Vector 3 vel = rig.velocity;
vel.y = 0;
```

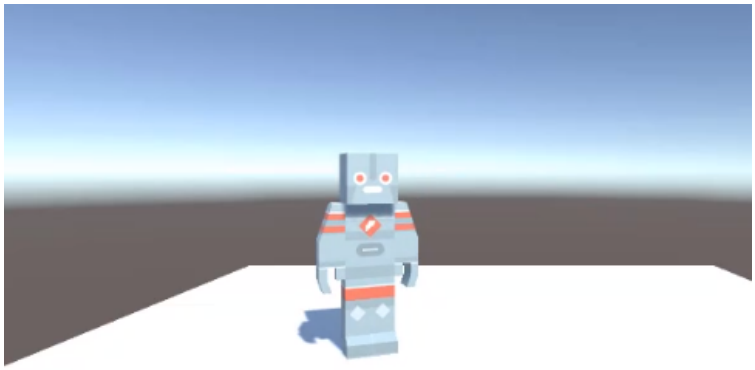
And we're going to replace the **rig.velocity** (original) with our **vel** (temporary).

```
transform.forward = vel;
```


Lastly, to stop our player's rotation from snapping back to the original rotation when we're not moving, we're going to create an **if statement**:

```
void Update() {

  ...

  Vector 3 vel = rig.velocity;
  vel.y = 0;

    //Are we moving? Check if our x-velocity or z-velocity is NOT equal to 0
    if(vel.x != 0 || vel.z != 0)
    {

      //Only rotate if we're moving
      transform.forward = vel;
    }

}
```

Press Play, and you should see that we now rotate to face the direction we're moving in without any issue.



In this lesson, we're going to be setting up player jumping.

First of all, we're going to create a new **public float** variable called "jumpForce".

```
public float jumpForce
```

Detecting Jump Input

Let's say we want to press the **Space** bar in order to jump. To do that, we can check a condition using **Input.GetKeyDown** in the Update function:

```
if(Input.GetKeyDown(KeyCode.Space))
{
}
```

The **GetKeyDown** function will return true on the frame that the requested key is pressed down– in this case, the key is **Space**.

Applying Force Using AddForce

Now, whenever the Space key is pressed, we're going to add a force (i.e. **jumpForce**) to our rigidbody (**rig**), using the **AddForce** function.

```
if(Input.GetKeyDown(KeyCode.Space))
{
    rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
}
```

We can use **ForceMode** to specify how to apply a force using Rigidbody.AddForce, for example:

- Force: Add a continuous force to the rigidbody.
- Acceleration: Add a continuous acceleration to the rigidbody, ignoring its mass.
- Impulse: Add an instant force impulse to the rigidbody.
- VelocityChange: Add an instant velocity change to the rigidbody, ignoring its mass.

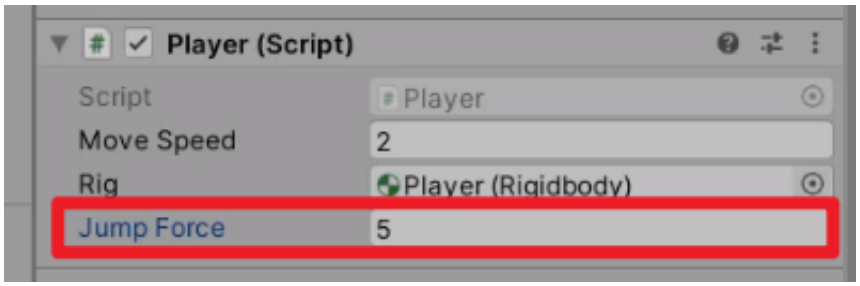
For more information, please refer to the documentation.

- Rigidbody.AddForce: https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html
- ForceMode: https://docs.unity3d.com/ScriptReference/ForceMode.html

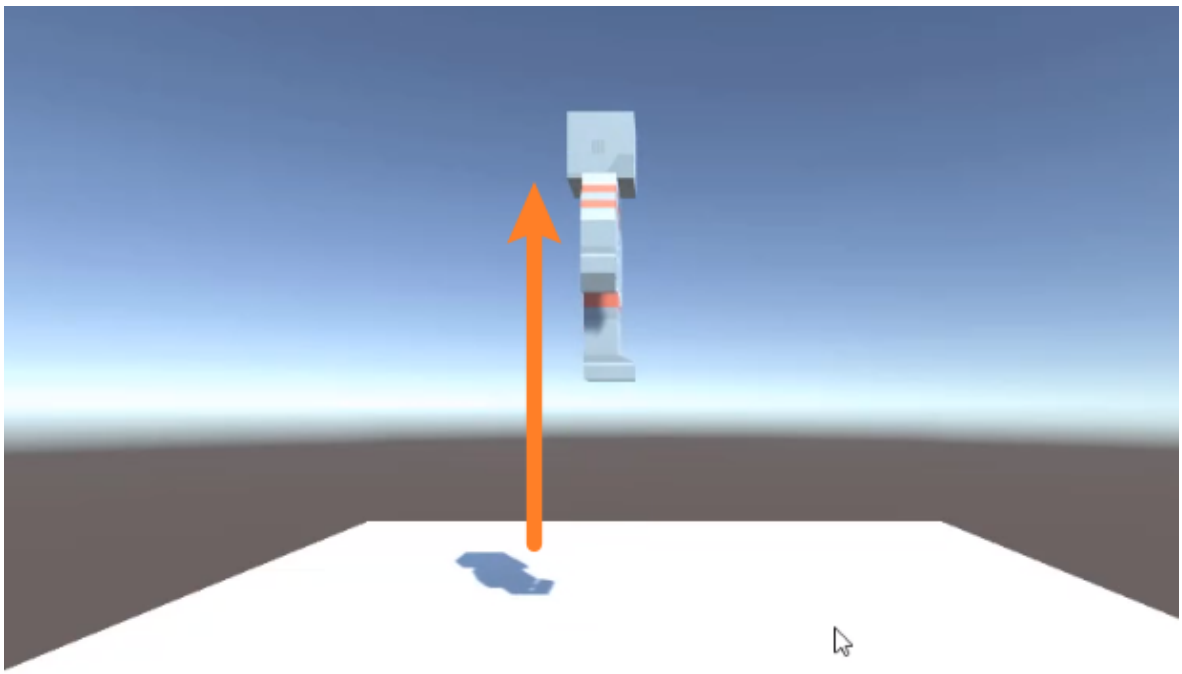
```
void Update() {

  //Get the horizontal and vertical inputs
  float x = Input.GetAxis("Horizontal") * moveSpeed;
  float z = Input.GetAxis("Vertical") * moveSpeed;

  //Set our velocity based on our inputs
  rig.velocity = new Vector3(x, rig.velocity.y, z );
```

```
  //Create a copy of our velocity variable and
  //Set the Y-axis to be 0
  Vector 3 vel = rig.velocity;
  vel.y = 0;

  //If we're moving, rotate to face our moving direction
  if(vel.x != 0 || yel.z != 0)
  {
     transform.forward = vel;
  }


  if(Input.GetKeyDown(KeyCode.Space))
  {
     rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
  }
}
```

Lastly, we're going to set our jump force value to 5 in the inspector and hit Play.



Now we can jump when we press Space. That said, our player can go all the way up to the sky if we keep pressing Space again and again.



However, we want our player to jump only if they're standing on the ground.

Ground Detection

We're going to create a new **boolean** variable to keep track of whether or not they're grounded in the next lesson.

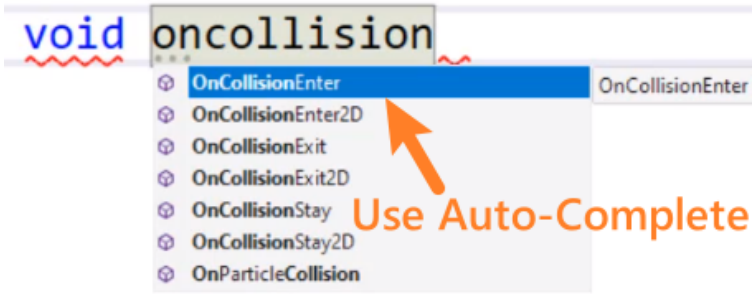
And we'll set it to **private** instead of public, as we don't need to modify the value in the Inspector.

```
private bool isGrounded;
```


In the previous lesson, we've created a **boolean** variable called 'isGrounded'. We're going to set this variable to either true/false to determine if we can jump.

Ground Detection: OnCollisionEnter

In order to detect the ground, we're going to create a function called **OnCollisionEnter()**, which is already built into UnityEngine:




```
private void OnCollisionEnter(Collision collision)
{

}
```

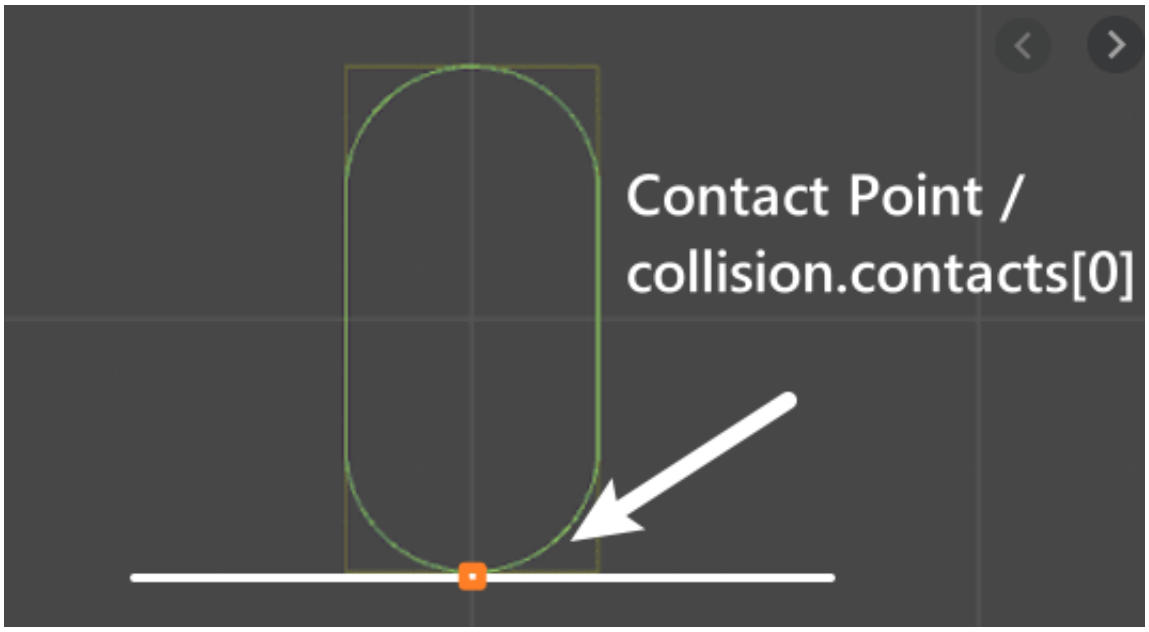
This function gets called whenever this collider/rigidbody enters another collider/rigidbody.

As you see in the parameter, it receives '**Collision**' class, which contains information about **contact points**, **impact velocity**, etc.

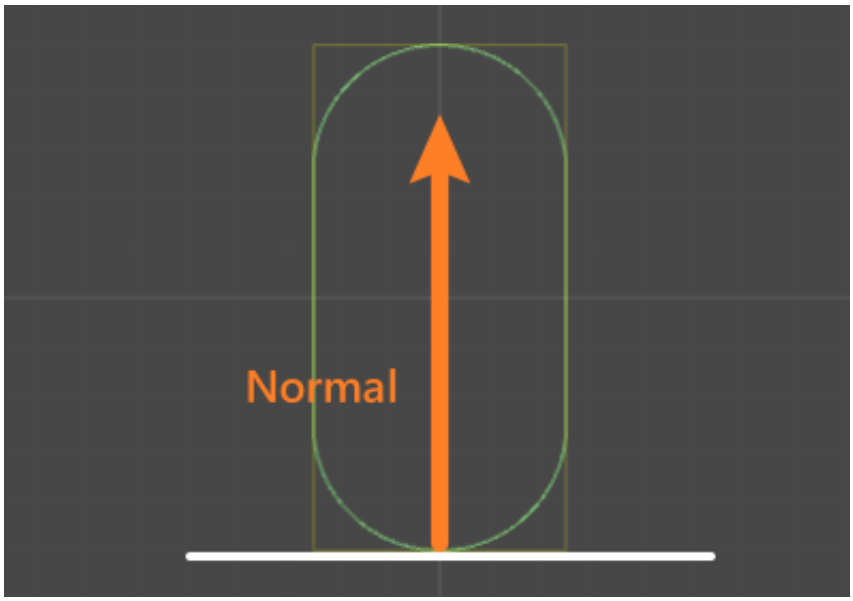
```
private void OnCollisionEnter(Collision collision)
{
```

For example, **collision.contacts** is an array of all the contact points that we hit. Since we're using a capsule collider, there is only one **contact point** at the bottom of the collider.

This contact point will be stored in collision.contacts as the first element (i.e. **collision.contacts[0]**).



And if the **normal** of this contact point is facing upwards, it means we're standing on the ground. (A **normal** is a vector that is perpendicular to its underlying surface.)



So we're going to check if the normal is facing upwards. If so, we can safely say that our player is grounded.

```
private void OnCollisionEnter(Collision collision)
{
   if(collision.contacts[0].normal == Vector3.up)
   {
      isGrounded = true;
   }
```

Ground Detection: Application

Right now, the only condition to jump is to press down the Space key:

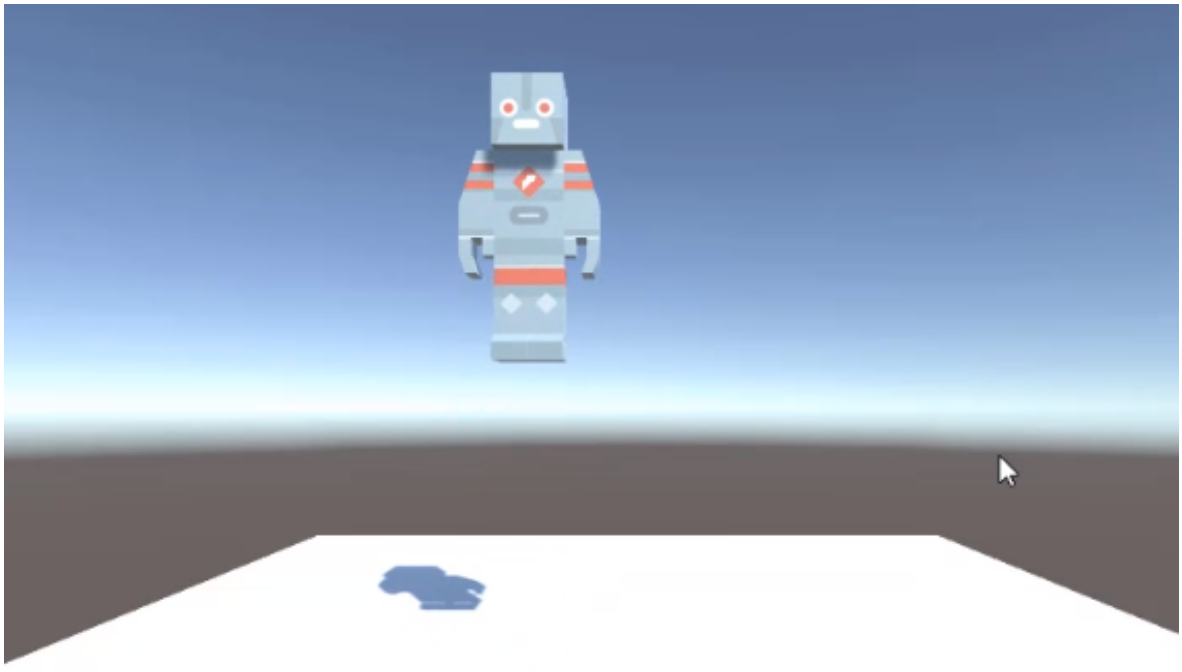
```
//If we press down the space button, then we can jump.
if(Input.GetKeyDown(KeyCode.Space))
{
   rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
}
```

We're going to add the bool variable (**isGrounded**) here as another condition to jump.

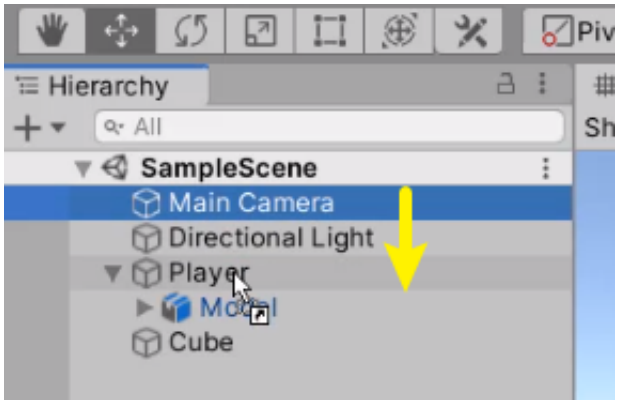
```
//If we press down the space button, AND if we're grounded, then we can jump.
if(Input.GetKeyDown(KeyCode.Space) && isGrounded)
{
   //When we jump, set isGrounded to be false.
   isGrounded = false;
   rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
}
```

Note that we used an **&&** to have two conditions that both need to be true in order for this if statement to run the code.

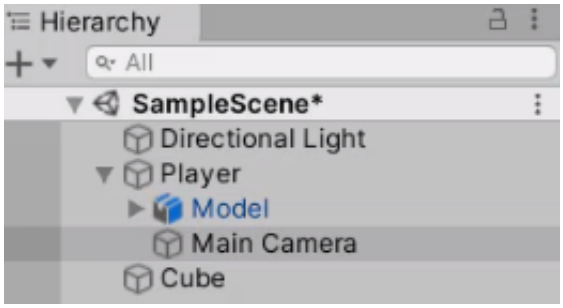
Now we won't be able to jump again until we hit the ground.



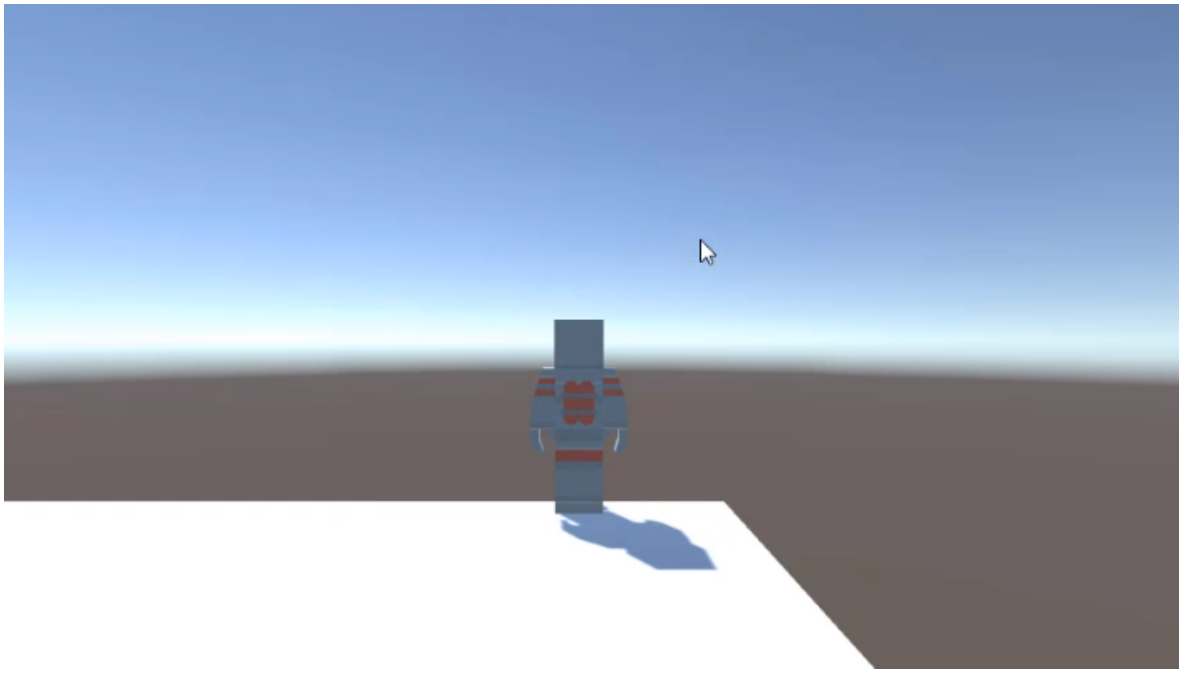
The simplest way to make a camera follow an object is to set it as a **child**. You can click on the camera and drag it into the Player object:



Now the camera is set as a **child** of the Player.



If we press play, we can see the camera follows our Player.

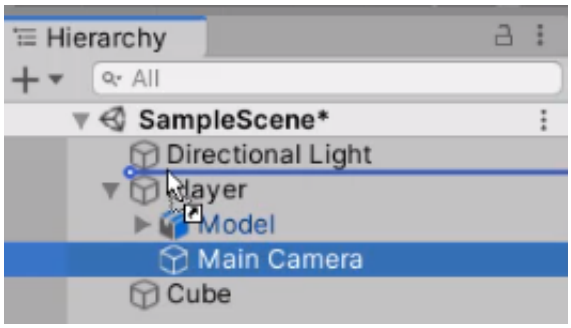


However, the camera not only follows our Player's position, but it also follows its rotation.

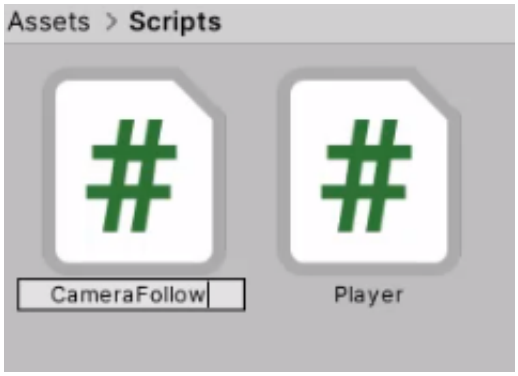
For example, if you are facing right and hit the left key, the camera is also turned 180 degrees, and this is not ideal.

Creating A CameraFollow Script

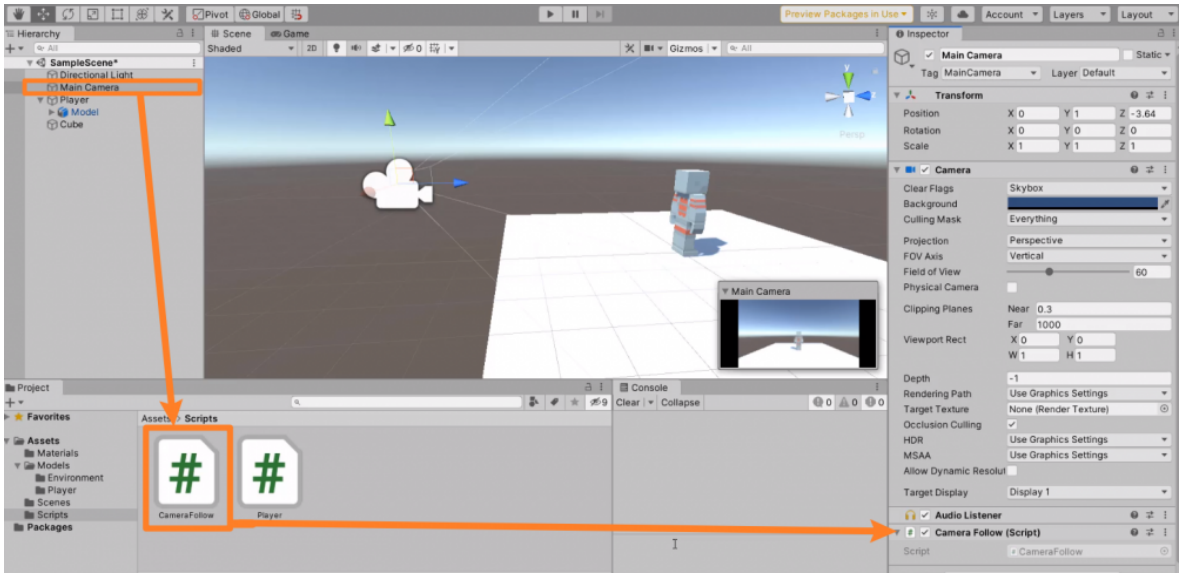
First, we're going to **unparent** the camera from the Player. (Drop it into where the blue line appears above the Player):



And we're going to create a new script called "CameraFollow" inside **Assets/Scripts**.



We're going to **attach** this script to the camera by dragging it into the Inspector.



We can then double-click the script and open it up in Visual Studio. Here, we're going to create two new variables:

```
public Transform target;
public Vecor3 offset;
```

The **target** will be used to reference a Transform component to follow, and the **offset** will be used to maintain a certain distance away from the player.

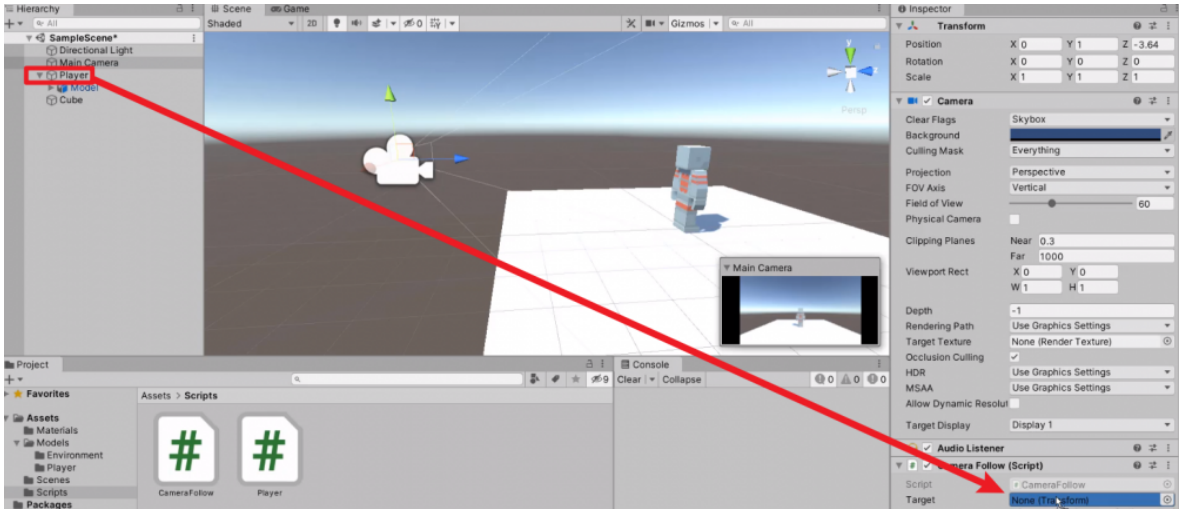
Inside the **Update** function, we're going to update our camera's position so that it follows the target's position, but with some **offset** so that our camera doesn't sit inside the player.

```
//Update is called once per frame.
void Update()
{
    transform.position = target.position + offset;
}
```

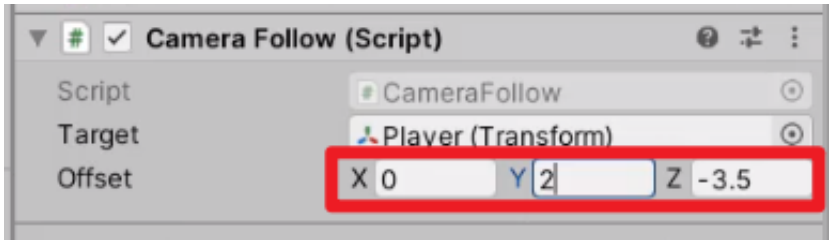
Let's hit save and return to our editor.

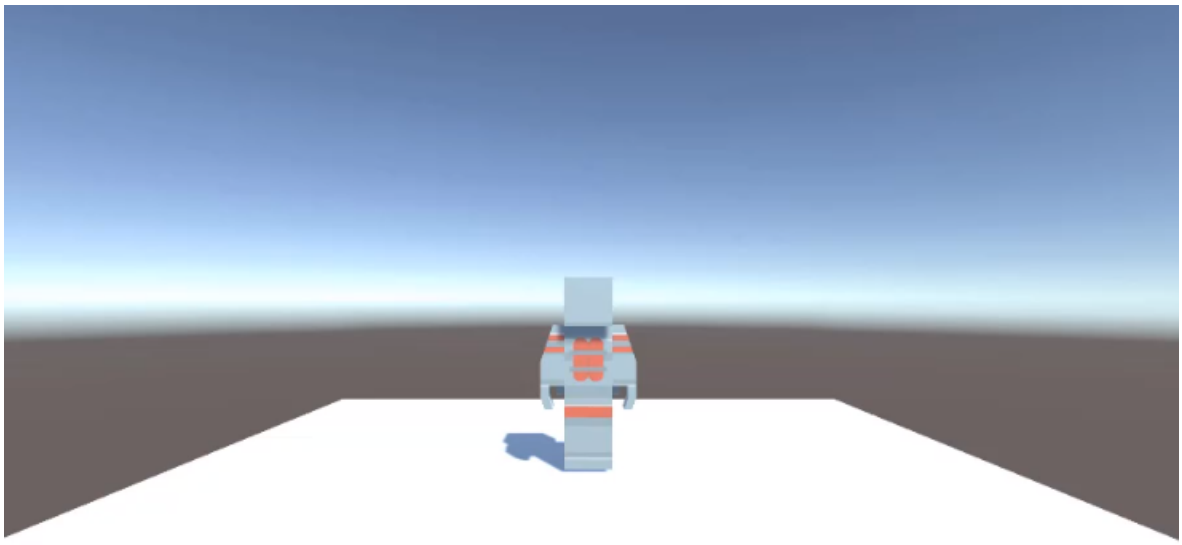
Setting Up Camera

Now we're going to drag in the **Player** into the **Target** slot.

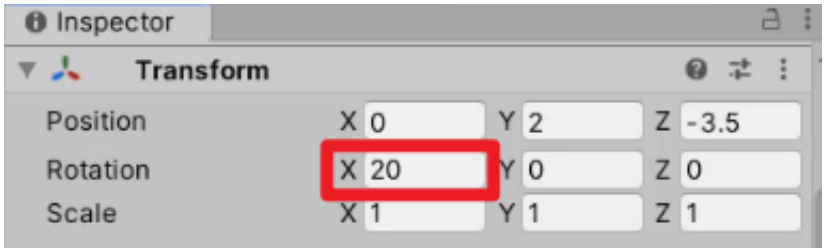


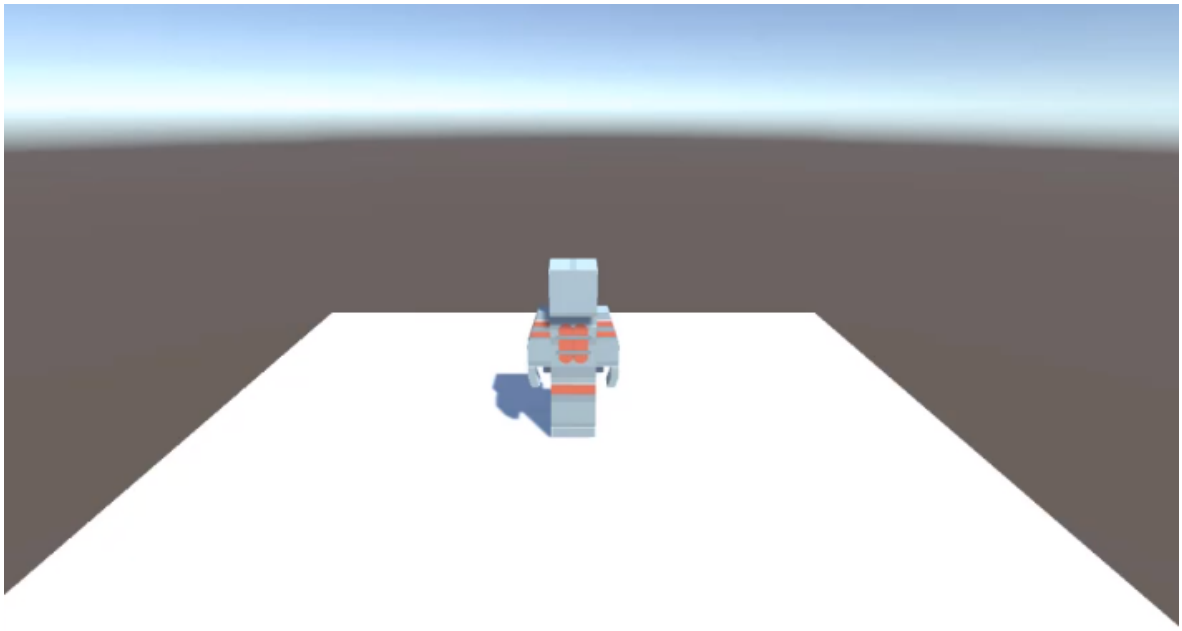
We're going to set the **offset** as (0, 2, -3.5):



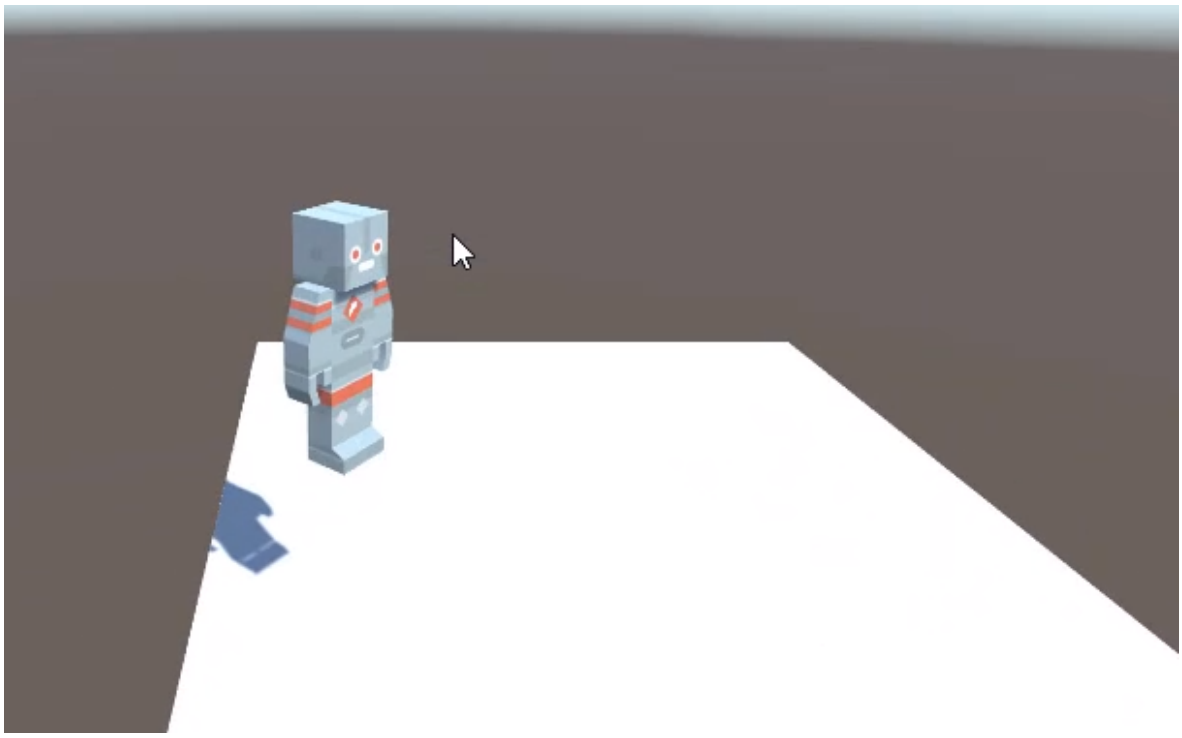


We're also going to set the **x-rotation** of our camera to be 20, so we're looking down slightly on the player.



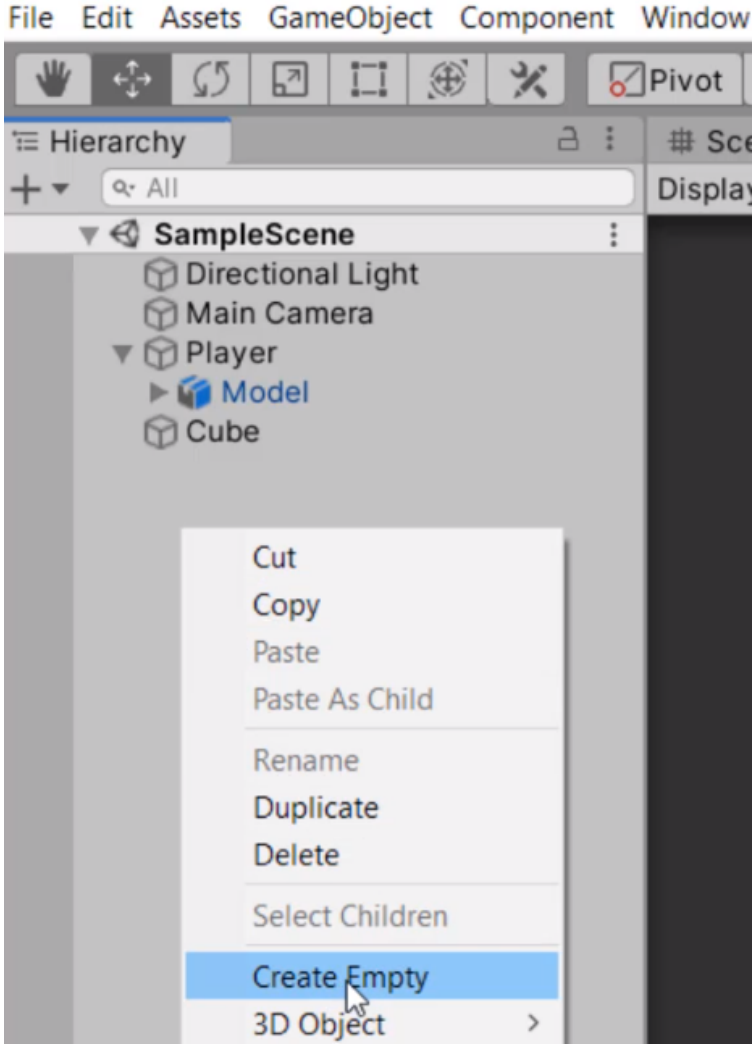


If we hit Play now, you should see that the camera is following our player, while always maintaining the **offset** (0, 2, -3.5) away from the player's position.

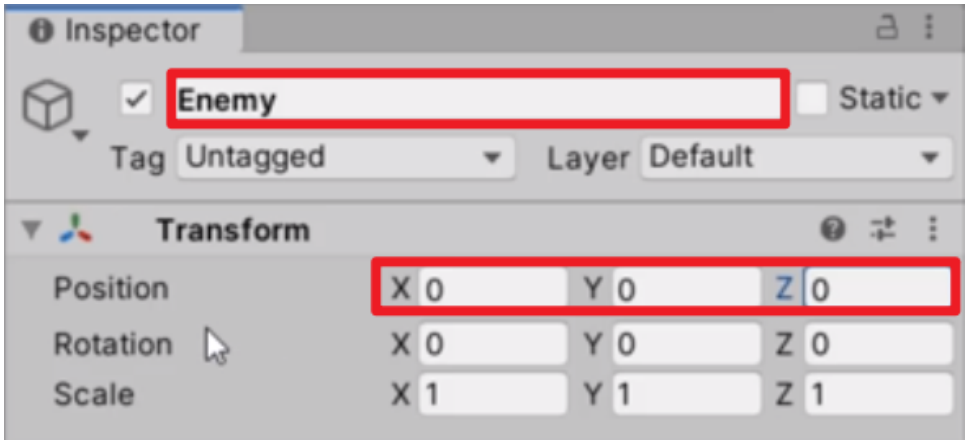


In this lesson, we're going to be creating our Enemy GameObject.

To create an empty gameObject, go to **Heirarchy > Right-click > Create Empty**.
(or **GameObject > Create > Empty**)

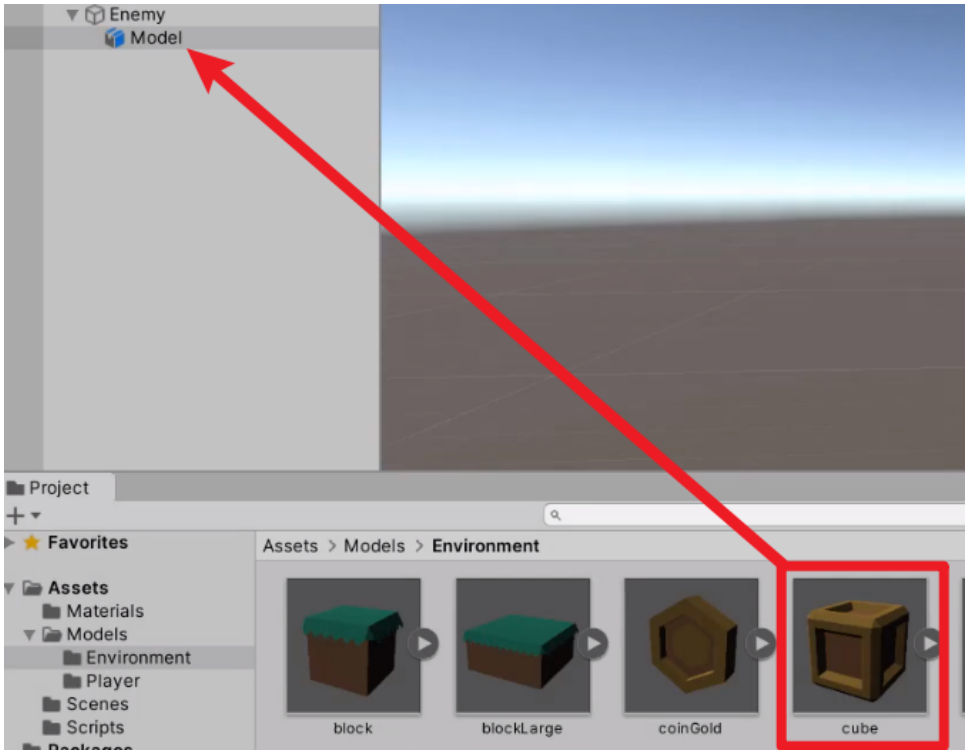


Let's make that the **Position** is set to **(0, 0, 0)**, and the name is set to "Enemy".

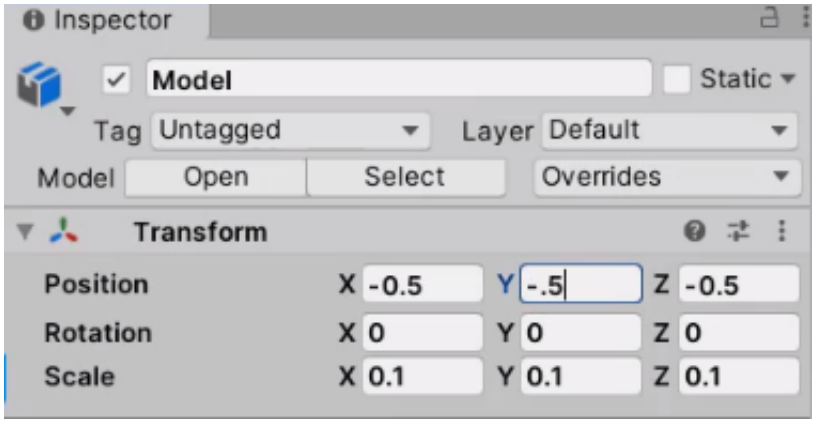


Assigning a 3D model

Go to **Assets > Models > Environment,** and drag the cube model into the **Hierarchy** as a child of Enemy.

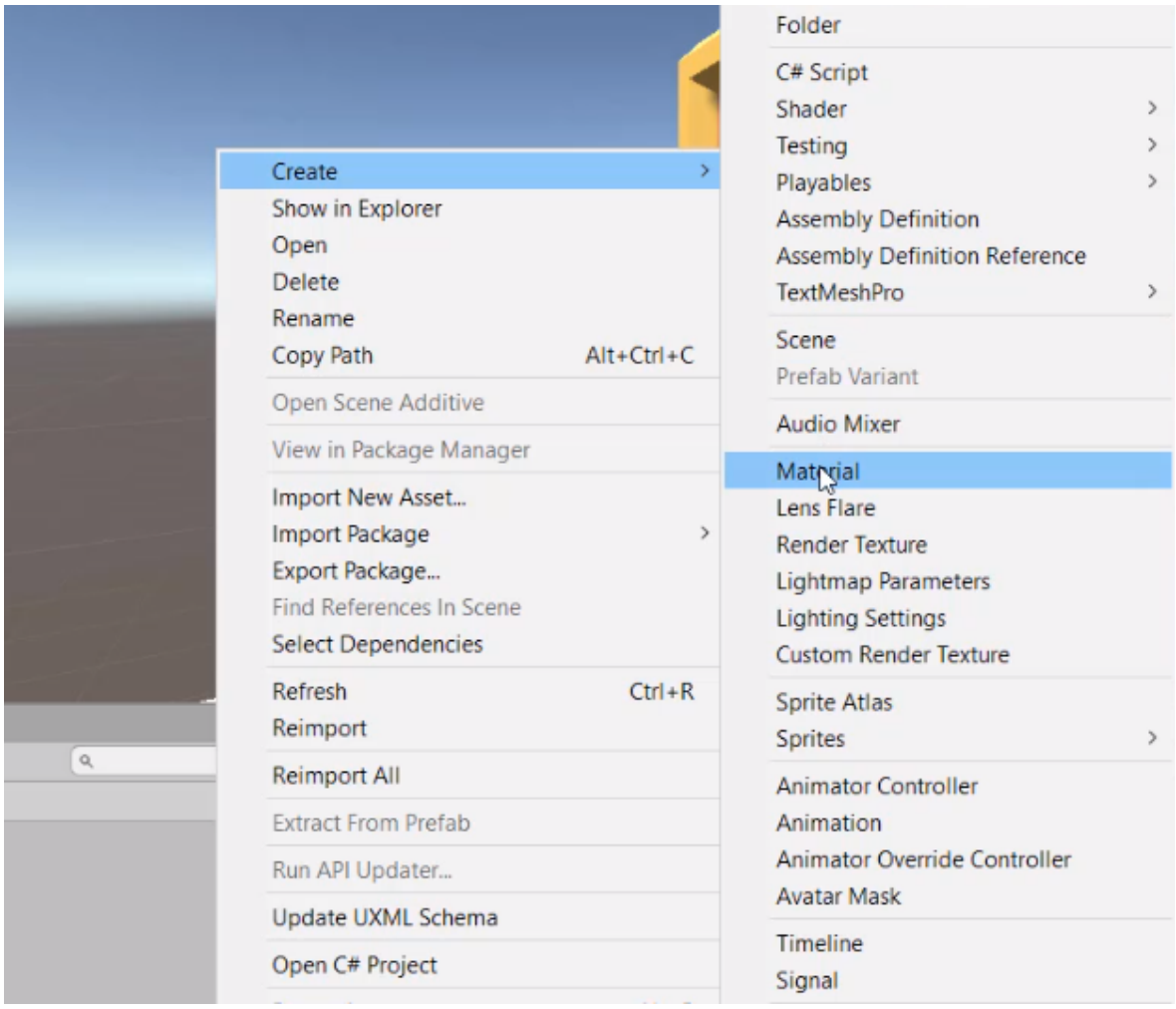


We're going to set the **Scale** as (0.1, 0.1, 0.1), and **rename** it to "Model" instead of "cube".

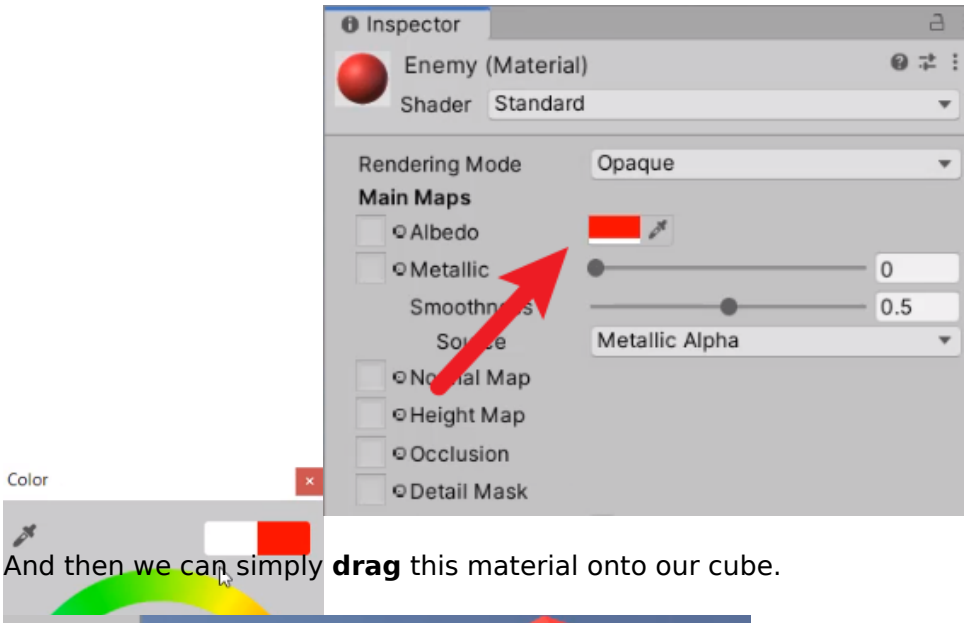


Creating A Material

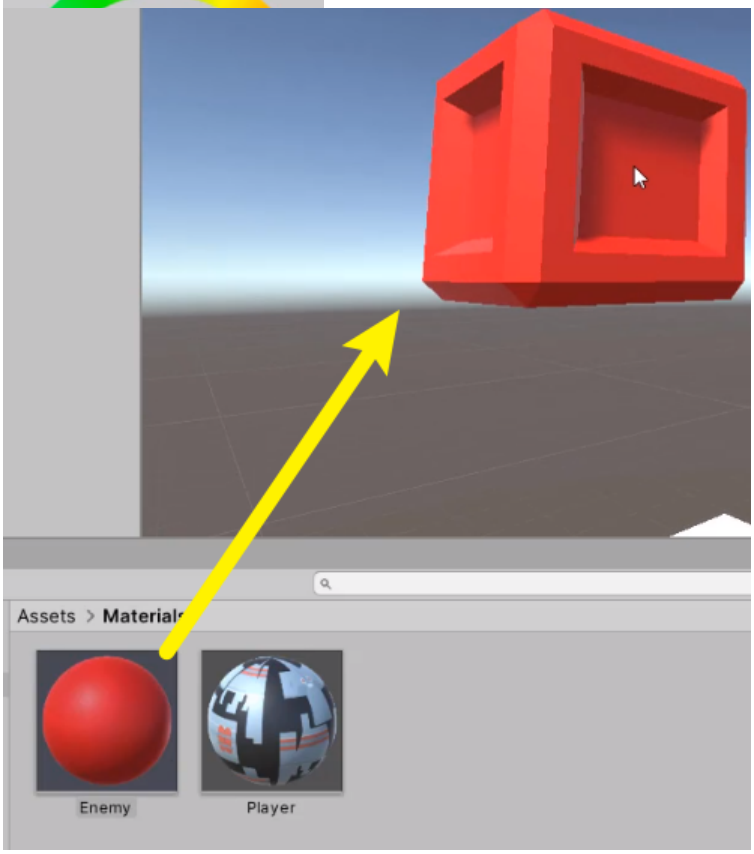
Go to **Assets > Materials**, and create a new Material for our enemy by **Right-click (Project) > Create > Material**.



We can change the colour of the material by clicking on '**Albedo**'. We've chosen the color red as it is a color associated with "danger".

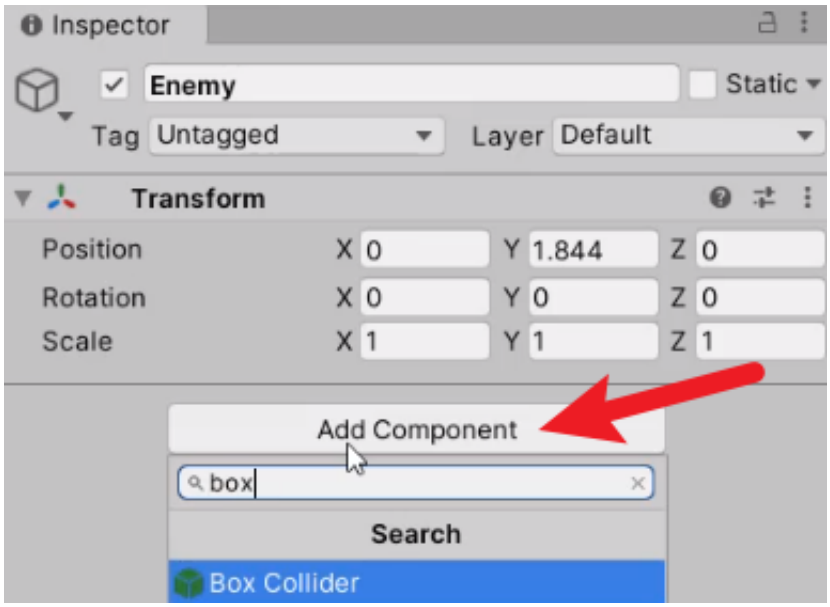


And then we can simply **drag** this material onto our cube.

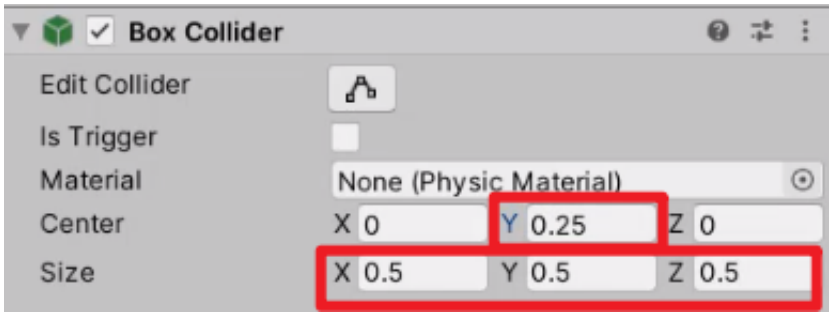


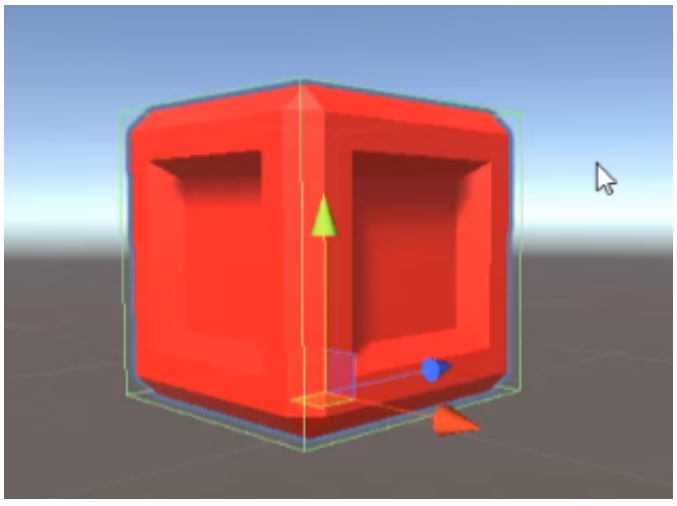
Adding A Collider

Select our Enemy, and go to **Add Component > Box Collider (Search)**.

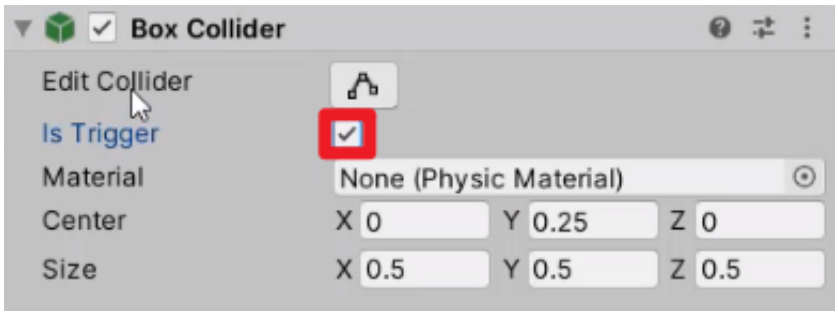


We're going to adjust the **Size** to be (0.5, 0.5, 0.5) and the **Y-center** to be (0.25), so our collider (green outline) fits our model.





We're also going to make this collider a **trigger.**

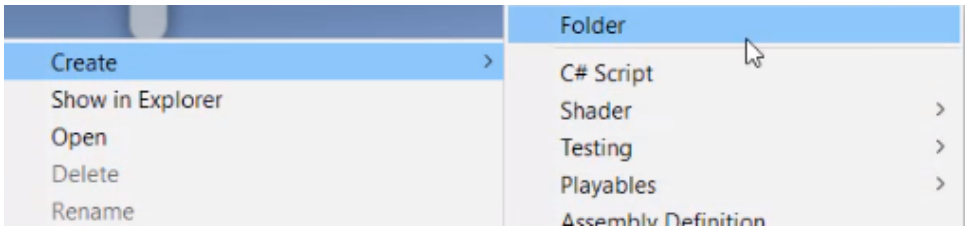


Triggers are effectively ignored by the physics engine, allowing us to **move through** the object. These are useful for **triggering events** in your game, such as collecting coins and walking through a doorway.

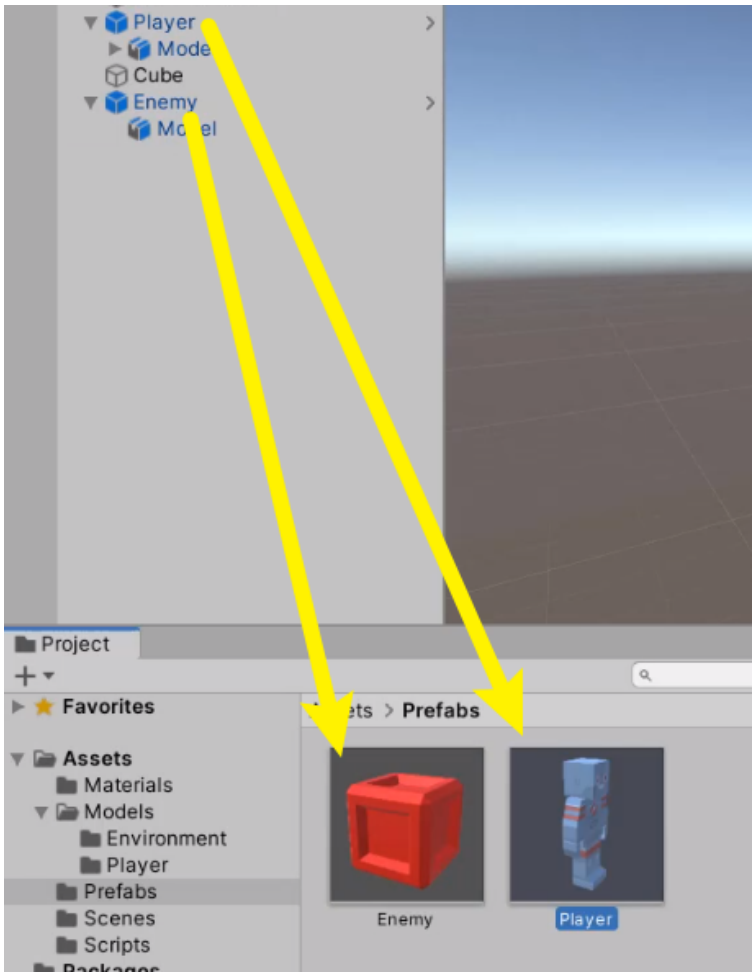
Turning GameObjects into Prefabs

Prefabs act as a template from which you can create new instances in the Scene. If there's anything that you're going to have multiples of, it is good practice to turn it into a prefab.

First, we're going to create a new folder called "Prefabs" (**Right-click > Create > Folder**)



And we can simply drag in our game objects inside of the Prefabs folder.

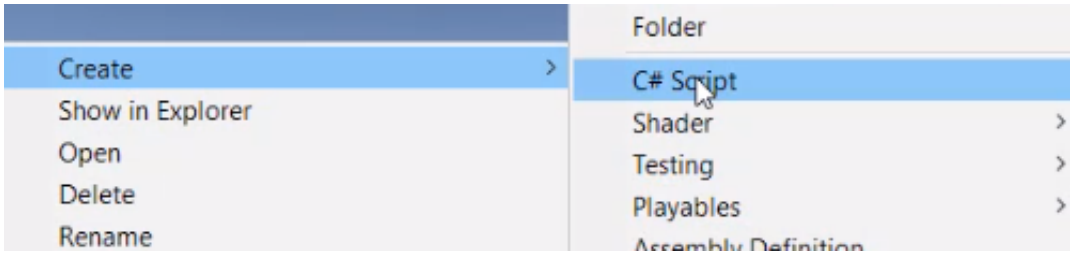


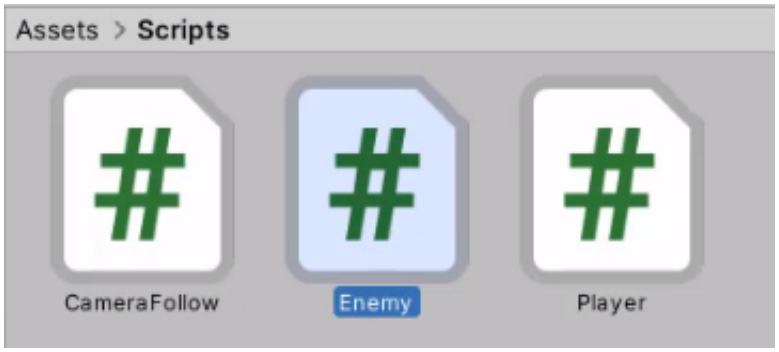
If you select the files, you can see that it appears as a "**Prefab Asset**" in the Inspector.

In this lesson, we're going to be working on moving our enemy.

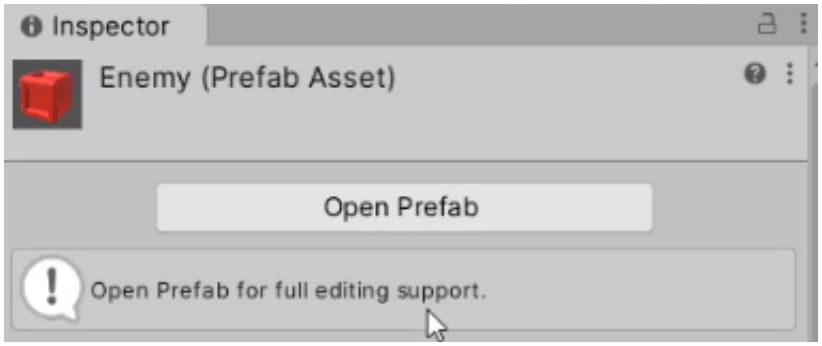
Creating A Script

Let's create a new C# Script (**Right-click > Create > C# Script**), and call it "Enemy".

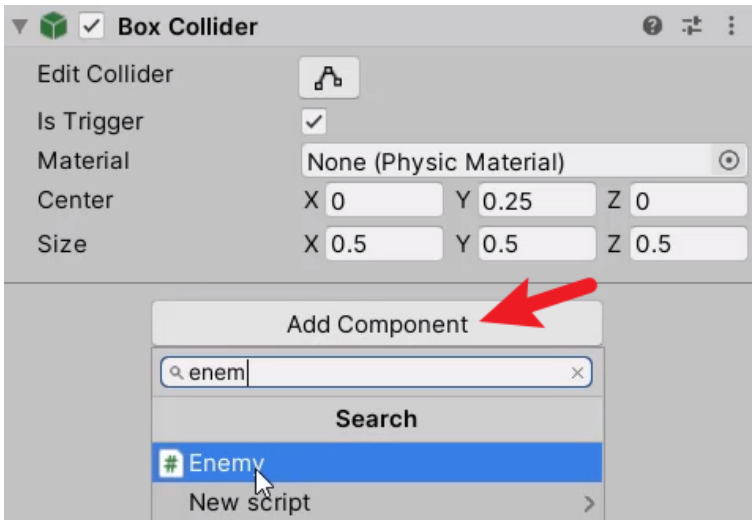




We're going to attach this script to the **Enemy Prefab** that we created in the previous lesson. To do that, select the prefab and click on **'Open Prefab'**.



Go to **Add Component > Enemy** and select the Enemy script.



Creating Variables

We're going to create a few variables based on the following questions:

- How fast is our enemy moving? (**speed**)
- Which **direction** is our enemy moving towards?
- How far in a certain **distance** is our enemy going to move?

```
public float speed;
public Vector3 moveDirection;
public float moveDistance;
```

In addition, we need to keep track of the starting position (**Vector3**) of our enemy, as we want it to move back and forth between two points.

```
private Vector3 startPos;
```

We also need to know whether we are moving to the start position, or the target position, which can be defined as a true/false **boolean** state.

```
private bool movingToStart;
```

Note that we set the last two variables as **private** since we don't need to access them in the Inspector.

Implementing Movement

First of all, we need to set our **initial position** at the start of the game.

```
public float speed;
public Vector3 moveDirection;
public float moveDistance;
```



```
private Vector3 startPos;
private bool movingToStart;

void Start()
{
    // Set start position at the start of the game
    startPos = transform.position;
}

void Update()
{
}
```

Now, we need to basically move either to a **start** position or to our **target** position (= moveDirection * moveDistance).

To do this, we're going to be using **Vector3.MoveTowards**, which is a function built into Unity that calculates a position between two points specified as 'current' and 'target'.

```
public Vector3 MoveTowards(Vector3 current, Vector3 target, float maxDistanceDelta);
```

The third parameter 'maxDistanceDelta', defines how far it moves each frame. It is basically speed multiplied by **Time.deltaTime,** which converts speed per frame to distance **per second**.

```
public Vector3 MoveTowards(Vector3 transform.position, Vector3 startPos, float speed
* Time.deltaTime);
```

For more information on MoveTowards(), refer to the documentation:
https://docs.unity3d.com/ScriptReference/Vector3.MoveTowards.html

We're going to use this function to update our position (**transform.position**) inside the Update() function:

```
transform.position = Vector3.MoveTowards(Vector3 transform.position, Vector3 startPos
, float speed * Time.deltaTime);
```

And if we have reached our start position, we need to switch our direction by setting **movingToStart** to false.

```
if(transform.position == startPos)
{
    movingToStart = false;
}
```

Here is the Enemy script that we've written so far:

```
public float speed;
public Vector3 moveDirection;
public float moveDistance;

private Vector3 startPos;
private bool movingToStart;

void Start()
{
    startPos = transform.position;
}

void Update()
{
   if(movingToStart)
   {
       transform.position = Vector3.MoveTowards(transform.position, startPos, speed *
 Time.deltaTime);
       if(transform.position == startPos)
       {
           movingToStart = false;
       }
   }
   else
   {
       //To-do: If reached startPos, move towards the opposite direction.
   }
}
```

```
public float speed;
public Vector3 moveDirection;
public float moveDistance;

private Vector3 startPos;
private bool movingToStart;

void Start()
{
    startPos = transform.position;
}

void Update()
{
   // are we moving to the start?
   if(movingToStart)
   {
       // over time move towards the start position
       transform.position = Vector3.MoveTowards(transform.position, startPos, speed *
 Time.deltaTime);

       // have we reached our target?
       if(transform.position == startPos)
       {
           movingToStart = false;
       }
   }
   // are we moving away from the start?
   else
   {
      //Move towards the target position
   }
}
```

In the previous lesson, we made our enemy move towards the **startPos**, and set **movingToStart** equal to false if it reaches the target.

Now we need to move towards the opposite direction; which means we're going to be doing pretty much the same thing, but with a different target position.

```
// are we moving away from the start?
else
{
   //Move towards the target position
   transform.position = Vector3.MoveTowards(transform.position, startPos + (moveDirec
tion * moveDistance), speed * Time.deltaTime);
}
```

And we're going to be checking if it has reached the target:

```
// are we moving away from the start?
else
{
   //Move towards the target position
   transform.position = Vector3.MoveTowards(transform.position, startPos + (moveDirec
tion * moveDistance), speed * Time.deltaTime);

   if(transform.position == startPos + (moveDirection * moveDistance))
}
```

Then we're going to set **moveToStart** to true. Now our enemy will be ping-ponging between the two if-else statements.

```
// are we moving TO the start?
if (movingToStart)
{
   // over time move towards the start position
   transform.position = Vector3.MoveTowards(transform.position, startPos, speed * Tim
e.deltaTime);

   // have we reached our target?
   if(transform.position == startPos)
   {
       movingToStart = false;
   }
}

// are we moving away FROM the start?
else
{
   //Move towards the target position
   transform.position = Vector3.MoveTowards(transform.position, startPos + (moveDirec
tion * moveDistance), speed * Time.deltaTime);

   if(transform.position == startPos + (moveDirection * moveDistance))
   {
       movingToStart = true;
   }
}
```

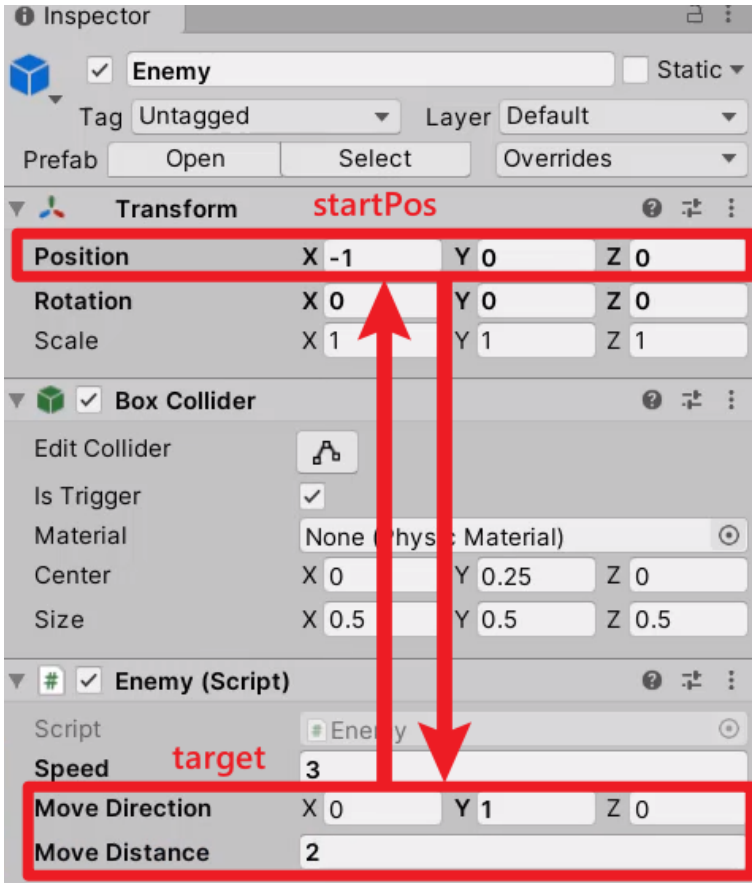
Here's what our Enemy script should look like:

```
public float speed;
public Vector3 moveDirection;
public float moveDistance;

private Vector3 startPos;
private bool movingToStart;

void Start()
{
```

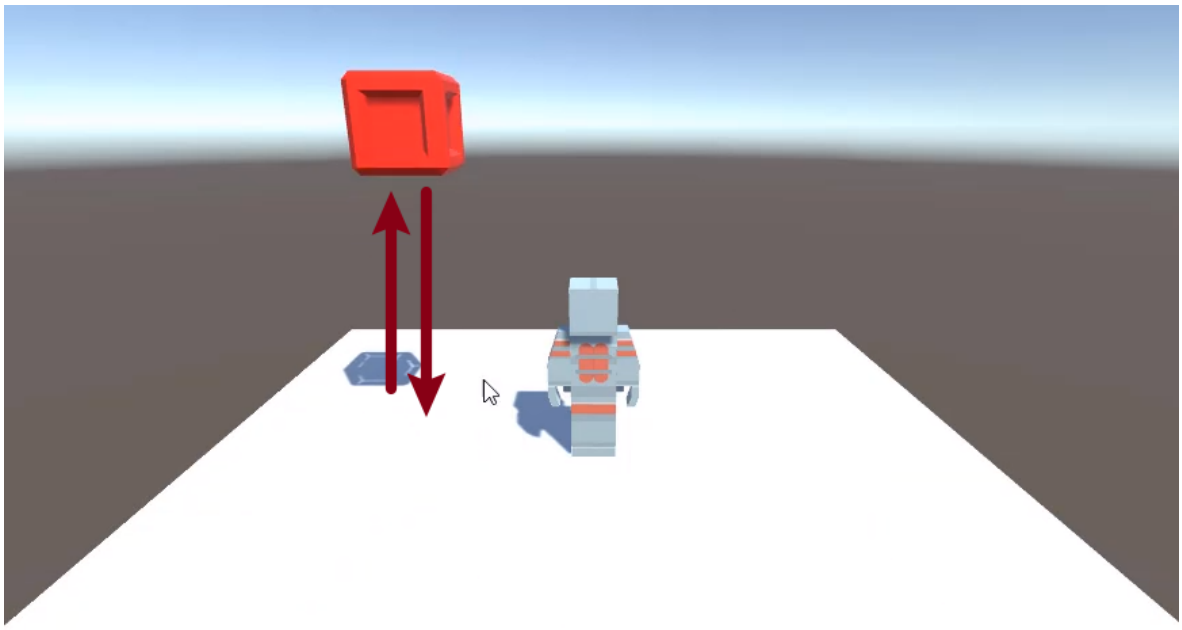
```
    startPos = transform.position;
}

void Update()
{
   if(movingToStart)
   {
       transform.position = Vector3.MoveTowards(transform.position, startPos, speed *
 Time.deltaTime);
       if(transform.position == startPos)
       {
           movingToStart = false;
       }
   }
   else
{
   transform.position = Vector3.MoveTowards(transform.position, startPos + (moveDirec
tion * moveDistance), speed * Time.deltaTime);

   if(transform.position == startPos + (moveDirection * moveDistance))
   {
       movingToStart = true;
   }
}
```

Testing Out

Let’s open up Unity Editor, and specify values for the public variables:



Now if we press Play, you should see the enemy moves between two points.



In this lesson, we're going to be setting up the player's game-over state.

There are two possible conditions for game-over:

- The player falls off the edge of the map down into the abyss, or
- The player gets hit by the enemy

If either one of the two conditions is met, we're going to restart the current scene.

Importing A Library

A library is a collection of pre-compiled modules that a program can use.

At the top of the Player script, we're going to be importing a **library** called **UnityEngine.SceneManagement**:

```
using UnityEngine.SceneManagement;
```

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
6   public class Player : MonoBehaviour
7   {
8       public float moveSpeed;
9       public Rigidbody rig;
```

Re-loading Scene using SceneManager

We're going to create a new function called **GameOver()**:

```
public void GameOver() {

}
```

Now inside the GameOver function, we're going to call SceneManager.LoadScene() method, which loads a scene by its name or index in Build Settings.

(Documentation: https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.LoadScene.html)

```
public void GameOver()
{
    SceneManager.LoadScene()
}
```

We're going to pass our current scene's build index by getting SceneManager.**GetActiveScene().buildIndex**:


```
//this function is going to reload the scene whenever it gets called.
public void GameOver()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex)
}
```

We can then call the GameOver() function if the player falls off the platform:

```
//Game over if we fall 10m off the platform
if(transform.position.y < -10)
{
    GameOver();
}
```

In this lesson, we're going to make our enemy script detect if it hit our player, and if so, we're going to make it reset the scene.

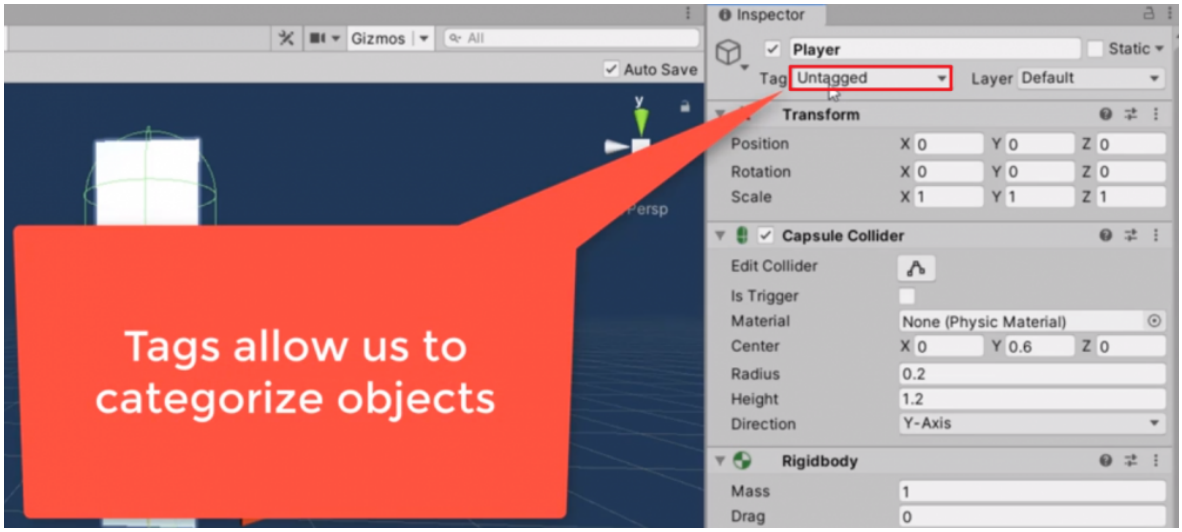
First of all, we're going to create a new function called **OnTriggerEnter** inside **Enemy.cs**:

```
private void OnTriggerEnter(Collider other)
{

}
```

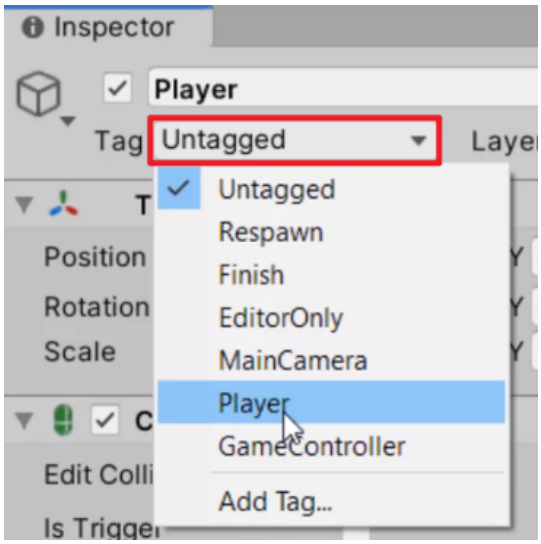
Unity calls **OnTriggerEnter** when a GameObject collides with another GameObject. The parameter named 'other' returns the other Collider involved in this collision.

Setting Up Tags

A **Tag** is a reference word which you can assign to GameObjects in Unity Editor:



We're going to select our Player GameObject, click on the "Tag" drop-down list, and choose "Player".



Using CompareTag To Filter GameObjects

Now, back into our Enemy script, we're going to check if the other collider's **tag** is "Player".

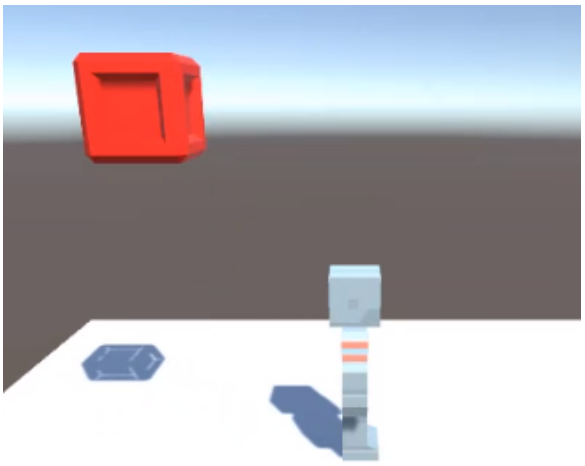
```
if(other.CompareTag("Player"))
```

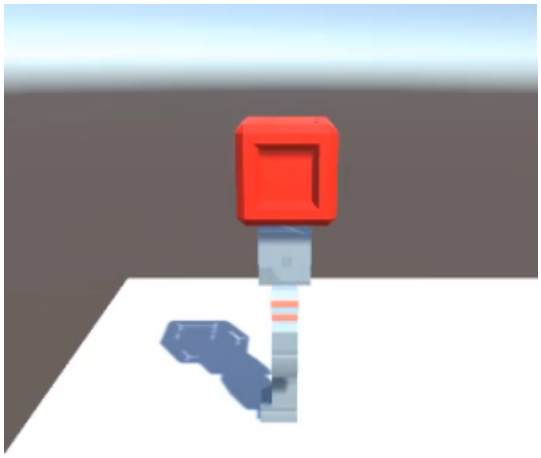
The **CompareTag** function will return true if the specified string (i.e. "Player") matches the tag of the other Collider. (If it hits a wall or a ground, it will return 'false'.)

So if the collider tag is 'Player', we're going to call the **GameOver** function.

```
private void OnTriggerEnter(Collider other)
{
   //If the collider tag is 'Player'...
   if(other.CompareTag("Player"))
   {
       //Call GameOver() that is inside "Player" class.
       other.GetComponent<Player>().GameOver();
   }

}
```

Now let's hit save and press **Play**. The moment we hit the cube, the scene will reset.

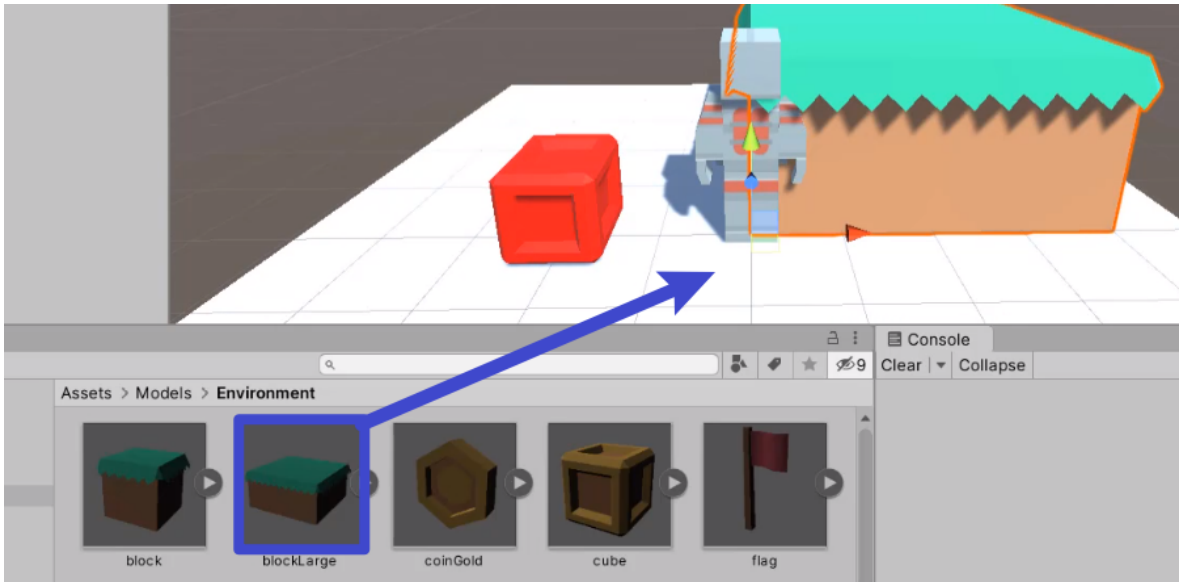




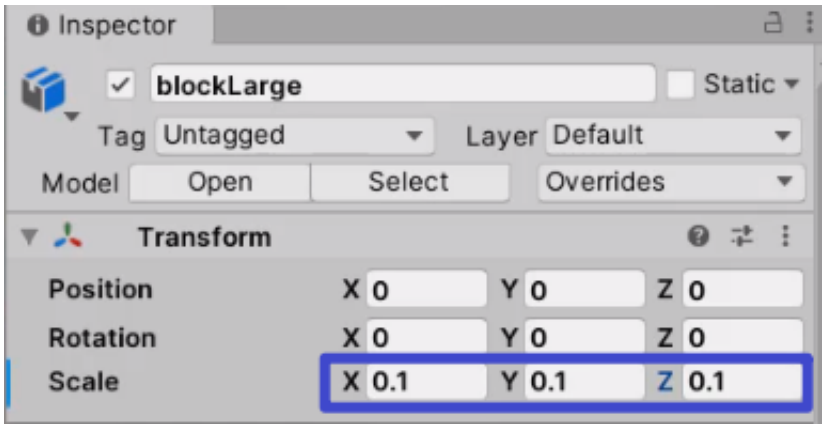
In this lesson, we're going to set up a 3D environment for our player to jump around.

Creating A Platform

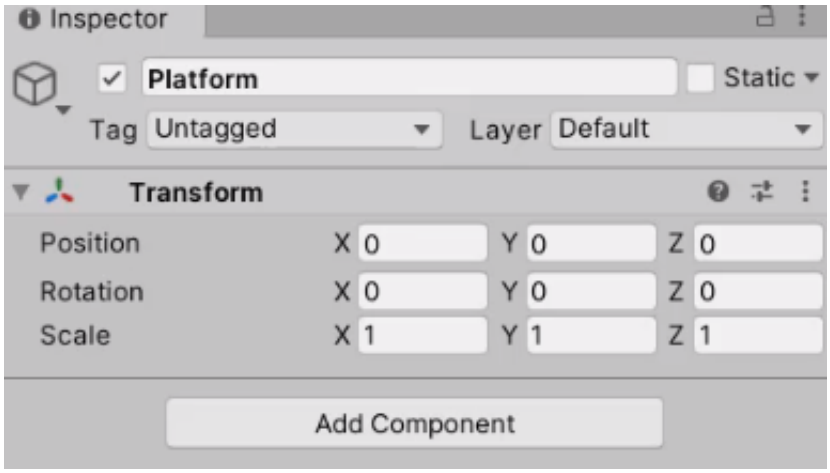
Let's go to **Assets > Models > Environment**, and drag in "blockLarge" into the **Scene** view.



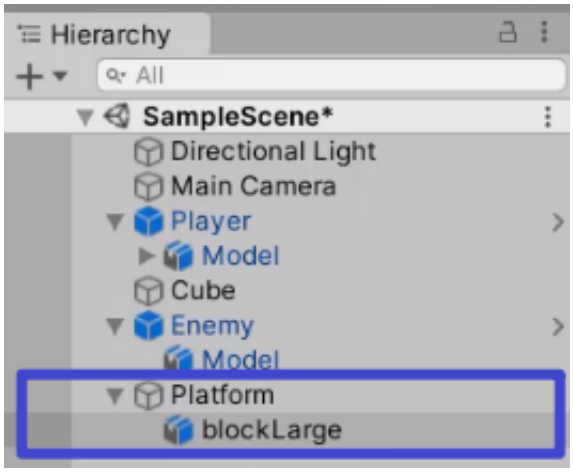
We're going to set the **Scale** to be 0.1 on all thee axis.



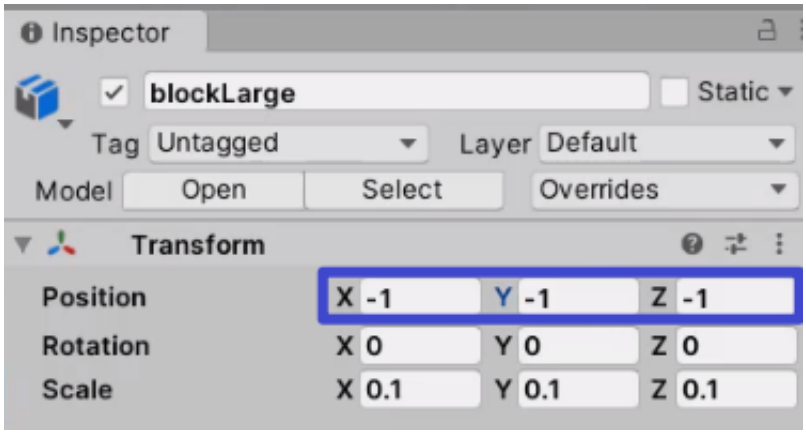
Let's **create an empty** game object to put this inside of, and call it "Platform".



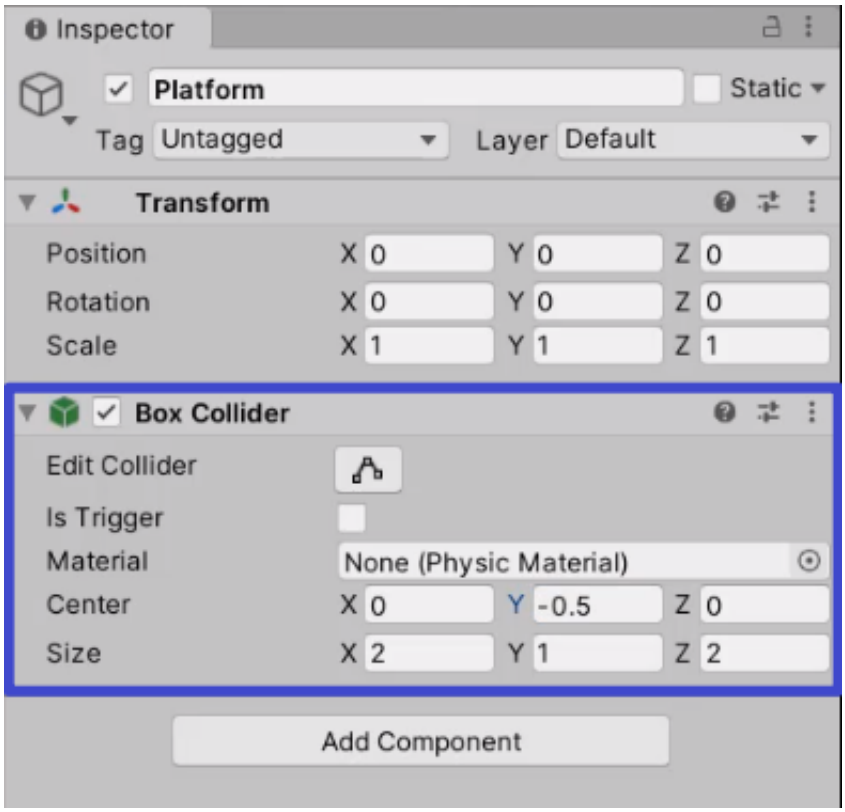
Make sure to **drag** the model into Platform as a child object.



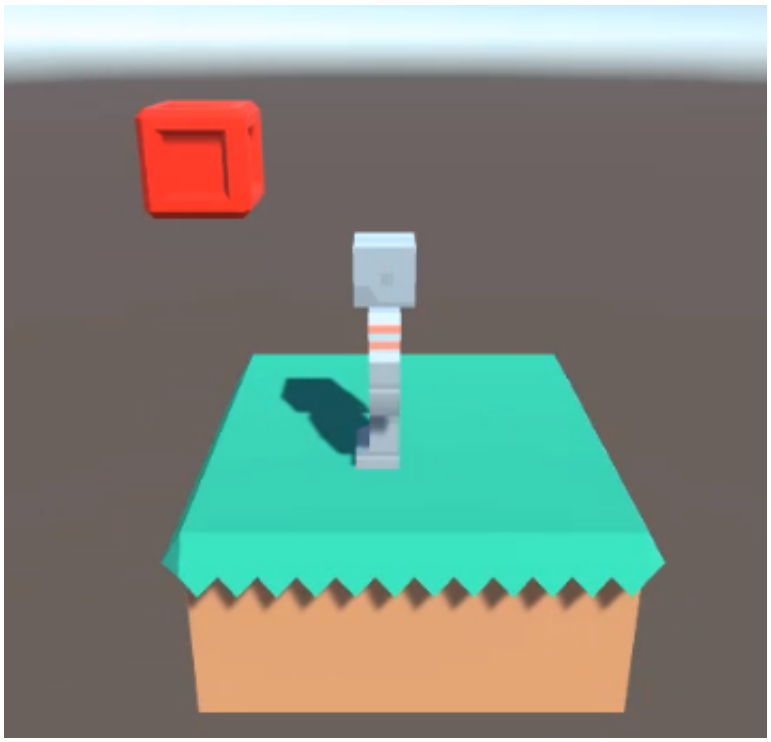
We can delete the cube that we created previously, and **position** our model underneath our Player.



And finally, let's add a **Box Collider** so that our player doesn't fall through the platform.

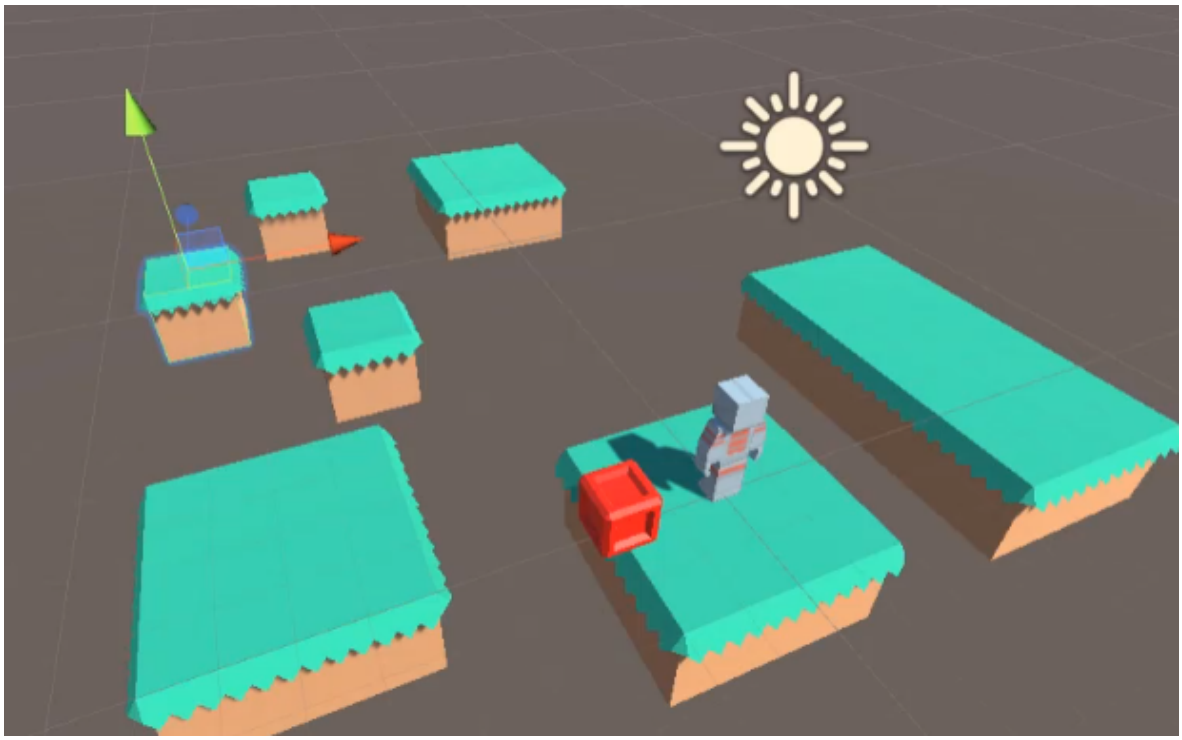


We can now press **Play** and test it out.

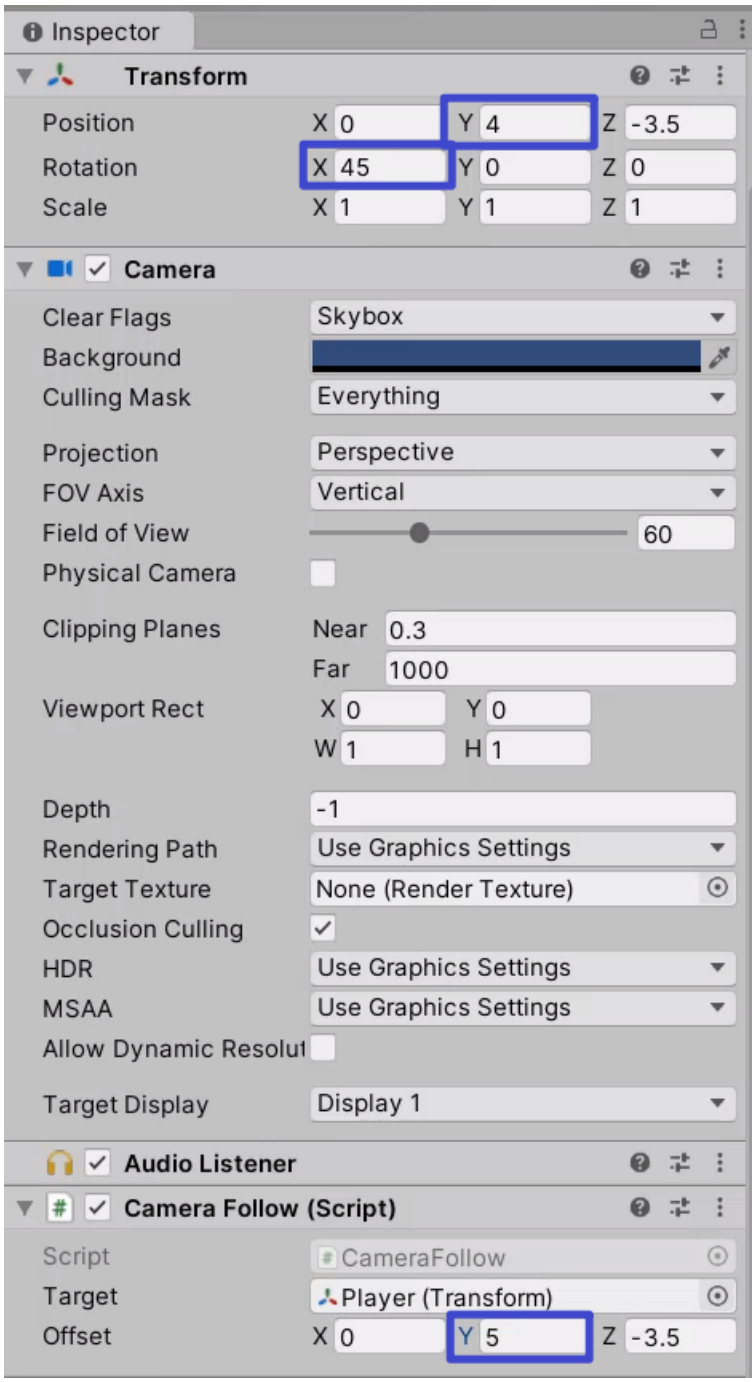


Creating A Level Layout

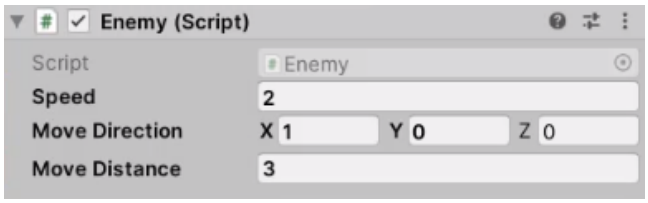
It's time to create your own level by **duplicating** the platform (**Ctrl+D**). Feel free to set up as large as you want, and remember any changes made during Play mode won't be saved.

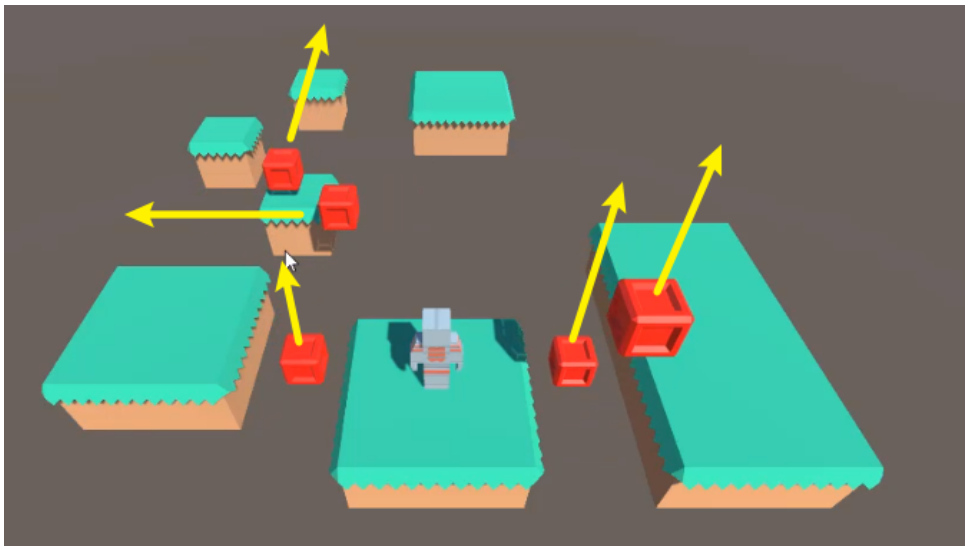


We changed how the **camera** is oriented so that it is more of a top-down view.



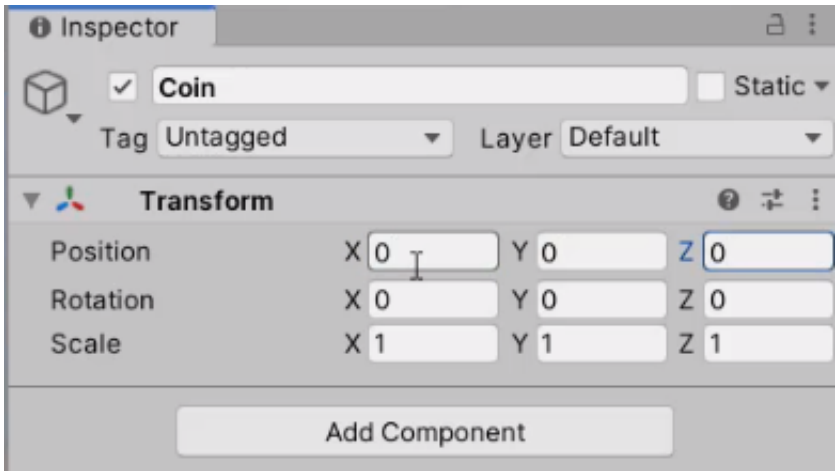
We also added in a few more **Enemy** prefab instances, and set each property with different values:



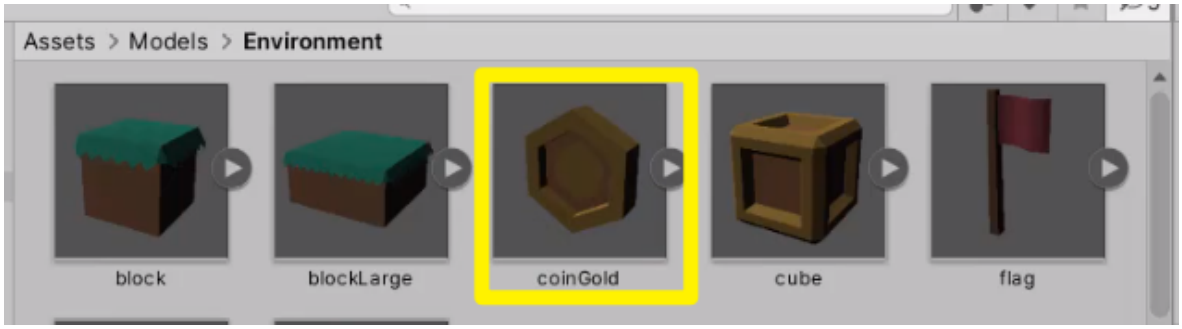


In this lesson, we're going to be creating a coin that the player can pick up to get a score.

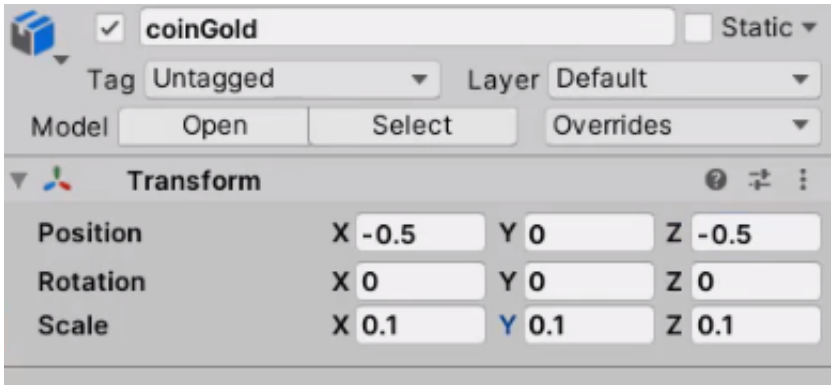
Let's create an **empty** GameObject named "Coin", and reset its **position** to 0, 0, 0.

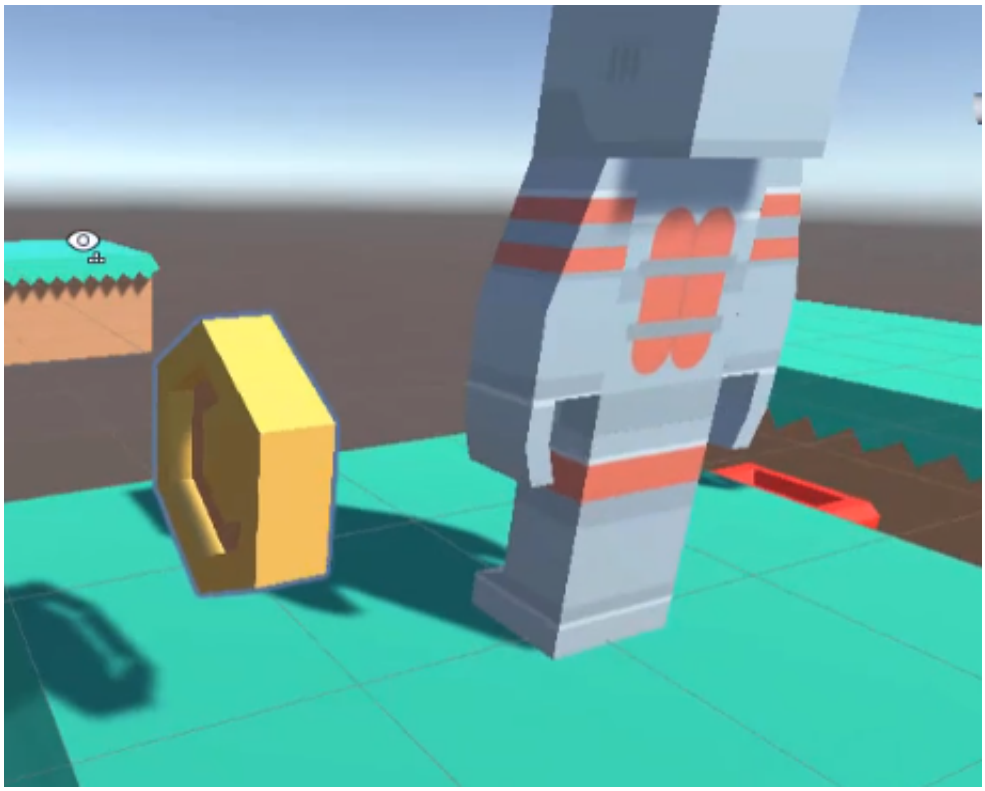


Go to **Assets > Models > Environment** and drag our coin model in as a child of the Coin empty gameObject.



We're going to adjust the **position** and the **scale**:

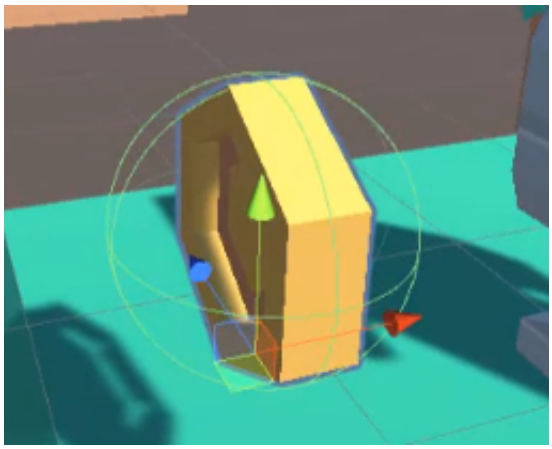




Adding A Collider

Next, we're going to add in a **collider** to detect trigger collisions when we go into this object.

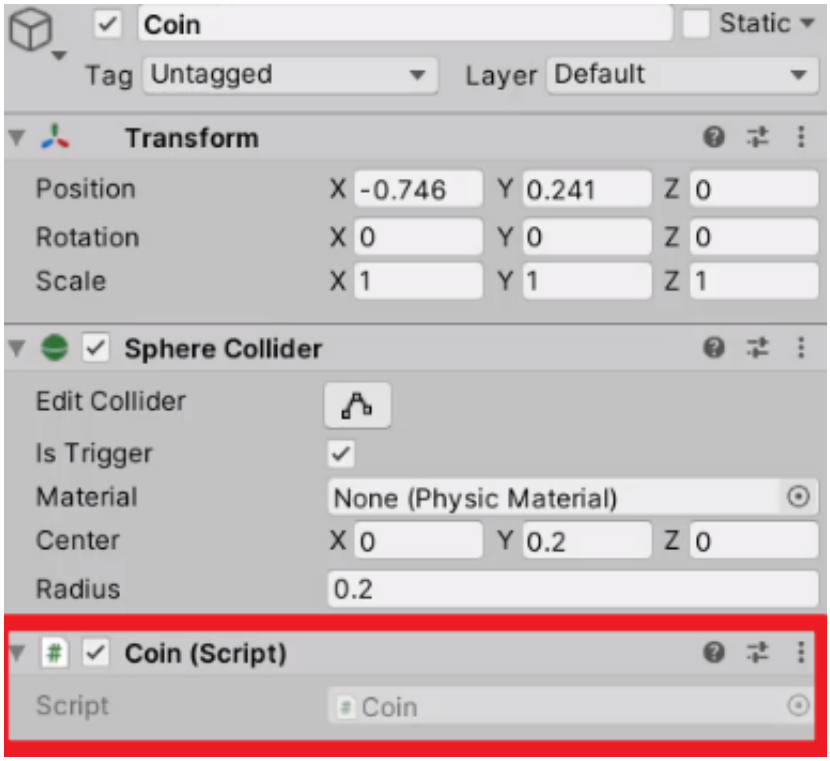
Make sure that the "**Is Trigger**" property is enabled, as it doesn't need to have a solid surface.



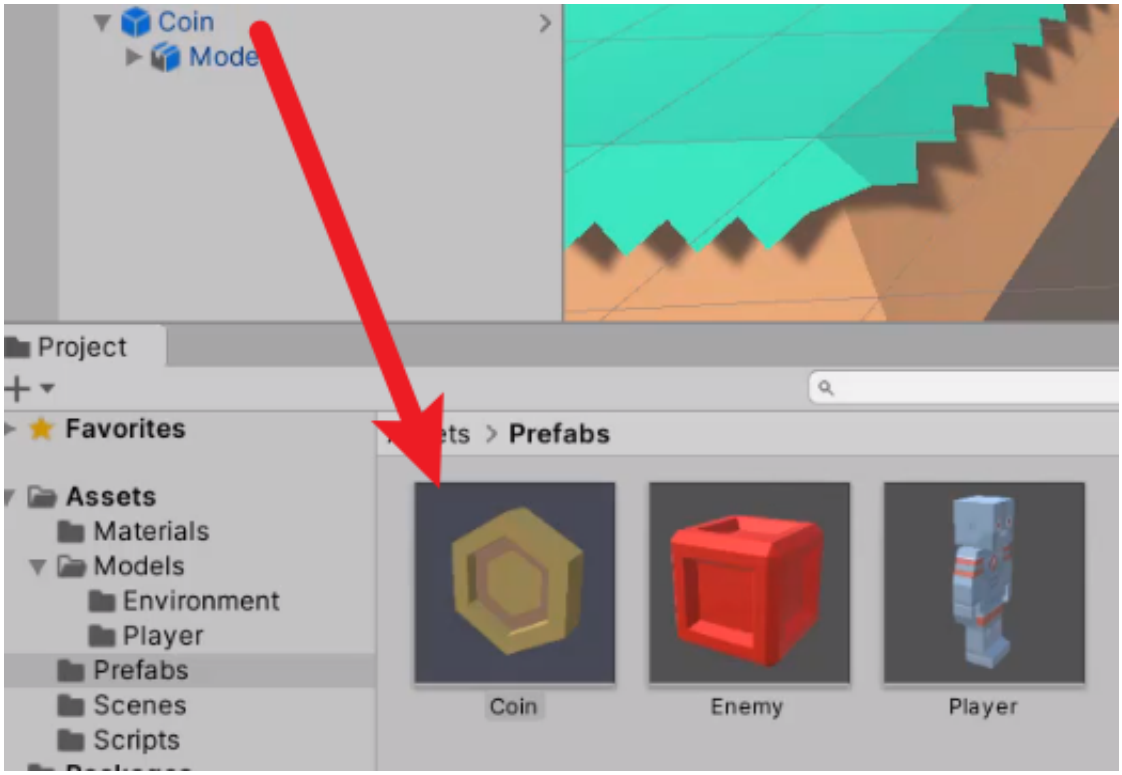
Creating A Coin Script

We're going to **create a new script** called "Coin" (Right-click > Create New > C# Script):

We can then go ahead and **attach** this to the Coin GameObject.



With all the steps above done, we can save our Coin object as a **prefab** and we can start placing as many coins as we want around.



Spinning The Coin

Here inside our Coin Script, we're going to declare a new float variable called **rotateSpeed**:

```
public class Coin : MonoBehaviour
{
    //How fast is the coin going to be rotating around?
    public float rotateSpeed;

    // Update is called once per frame
    void Update()
    {

    }

}
```

In the Update function, we're going to access our **transform** component, and call the **Rotate()** function.

```
public class Coin : MonoBehaviour
{
    //How fast is the coin going to be rotating around?
    public float rotateSpeed;

    // Update is called once per frame
```



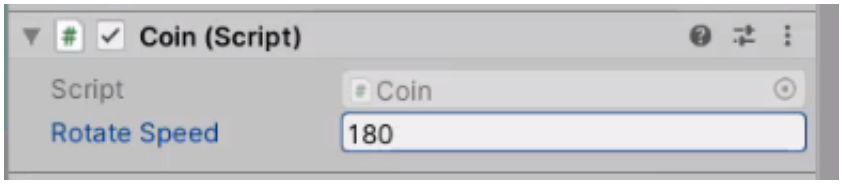
```
    void Update()
    {
        transform.Rotate();
    }

}
```

Inside the parenthesis, we can pass in the speed and the direction of the rotation. If you want to rotate vertically, we need to rotate around Y-axis (i.e. **Vector3.up**).

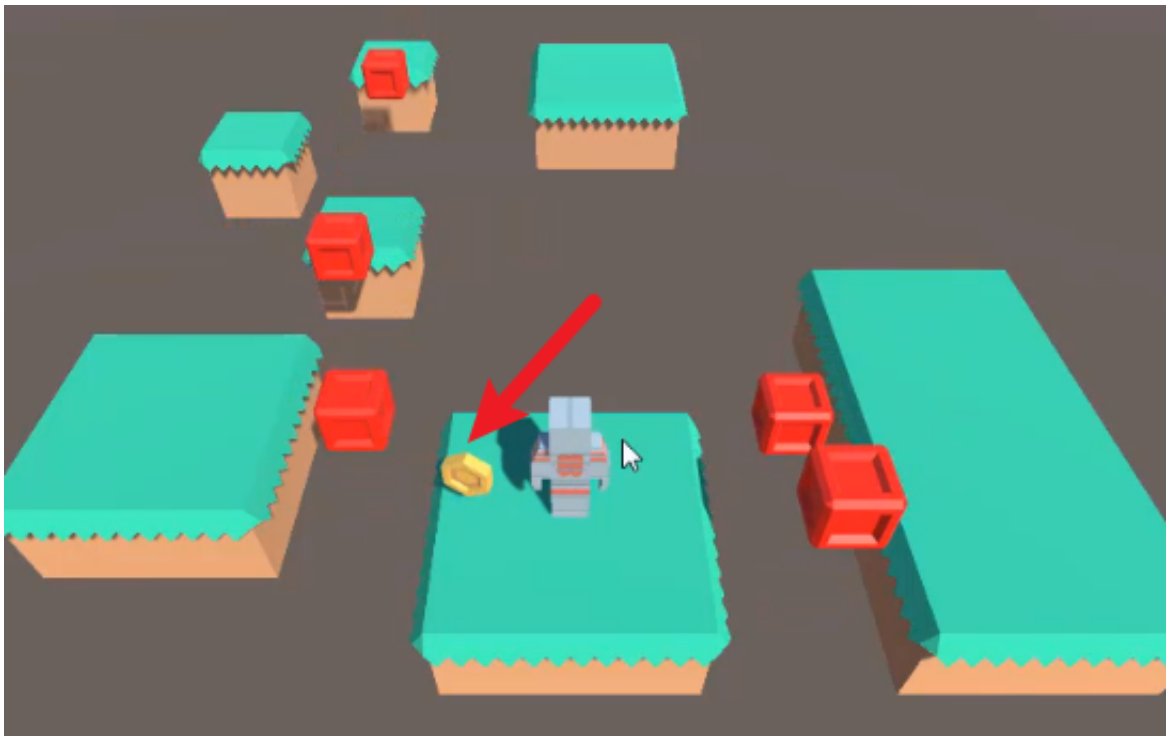
```
public class Coin : MonoBehaviour
{
    //How fast is the coin going to be rotating around?
    public float rotateSpeed;

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(Vector3.up, rotateSpeed * Time.deltaTime);
    }

}
```

Note that we multiplied our rotateSpeed with **Time.deltaTime** to convert it from degrees **per frame** to degrees **per second**.

Let's save the script, select our Coin prefab, and set the rotate speed to 180. So every two seconds, it should do a full revolution.



When you press Play, you should see that our coin is now spinning.



In this lesson, we're going to be setting up a collision detection with our coin and add the score to our player.

Let's open up our Player script (**Player.cs**), and create a new public int called score.

```
public int score
```

We're going to add a new function called **AddScore**, which takes in an **int** variable (i.e. the amount to add to a score):

```
public void AddScore(int amount)
{

}
```

When the function is called, we're going to add the amount to the score.

```
public void AddScore(int amount)
{
    score += amount;
}
```

Now, we need to detect if we have hit the coins inside **OnTriggerEnter()**.

```
private void OnTriggerEnter (Collider other)
{
    //If the object we hit is tagged as "Player"
    if(other.CompareTag("Player"))
    {
    }
}
```

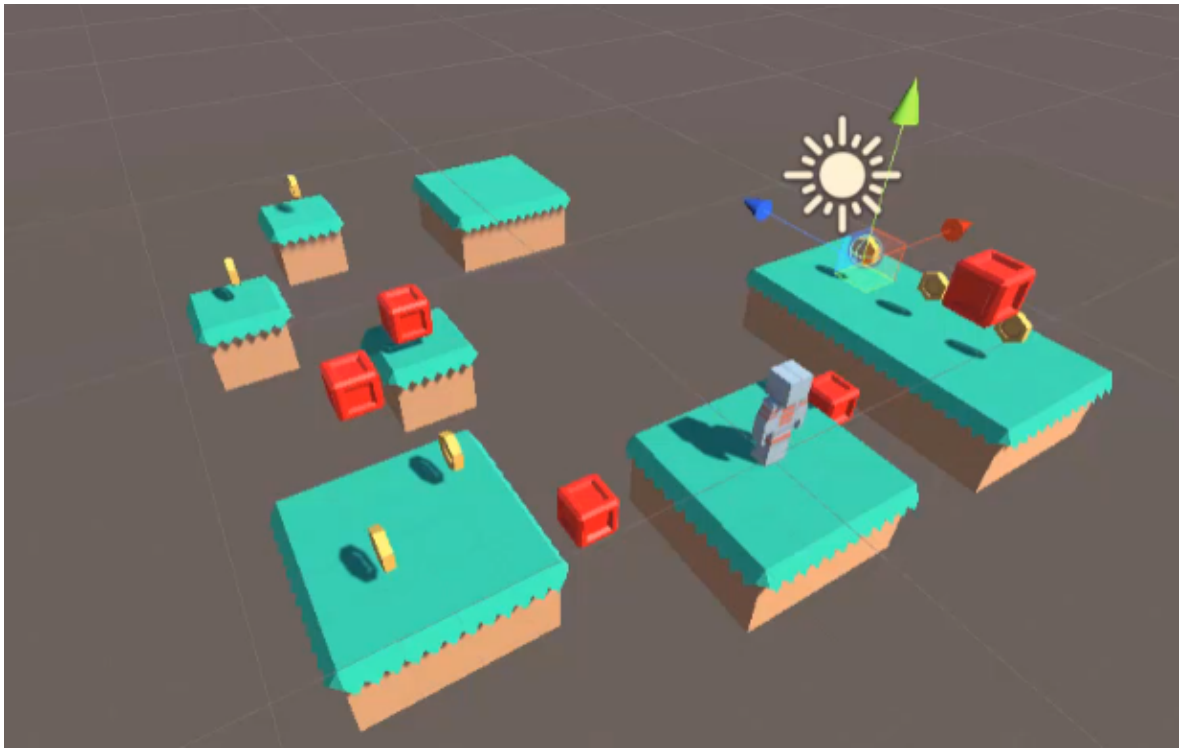
Then we're going to access the Player script and call the **AddScore** function that we created earlier. Remember to pass in an integer value for the amount of score to add:

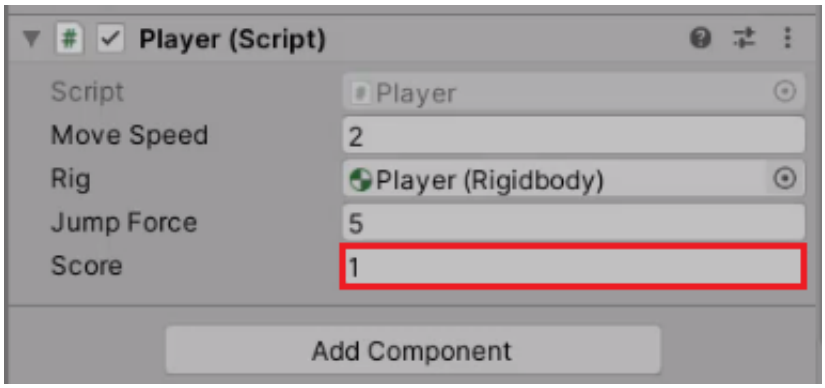
```
private void OnTriggerEnter (Collider other)
{
    //If the object we hit is tagged as "Player"
    if(other.CompareTag("Player"))
    {
        other.GetComponent<Player>().AddScore(1);
    }
}
```

And the coin object should be destroyed using **Destroy()** function.

```
private void OnTriggerEnter (Collider other)
{
    //If the object we hit is tagged as "Player"
    if(other.CompareTag("Player"))
    {
        other.GetComponent<Player>().AddScore(1);
        Destroy(gameObject);
    }
}
```

Let’s save this and return to our Editor. Now if we collide with a coin, our score gets increased by one.

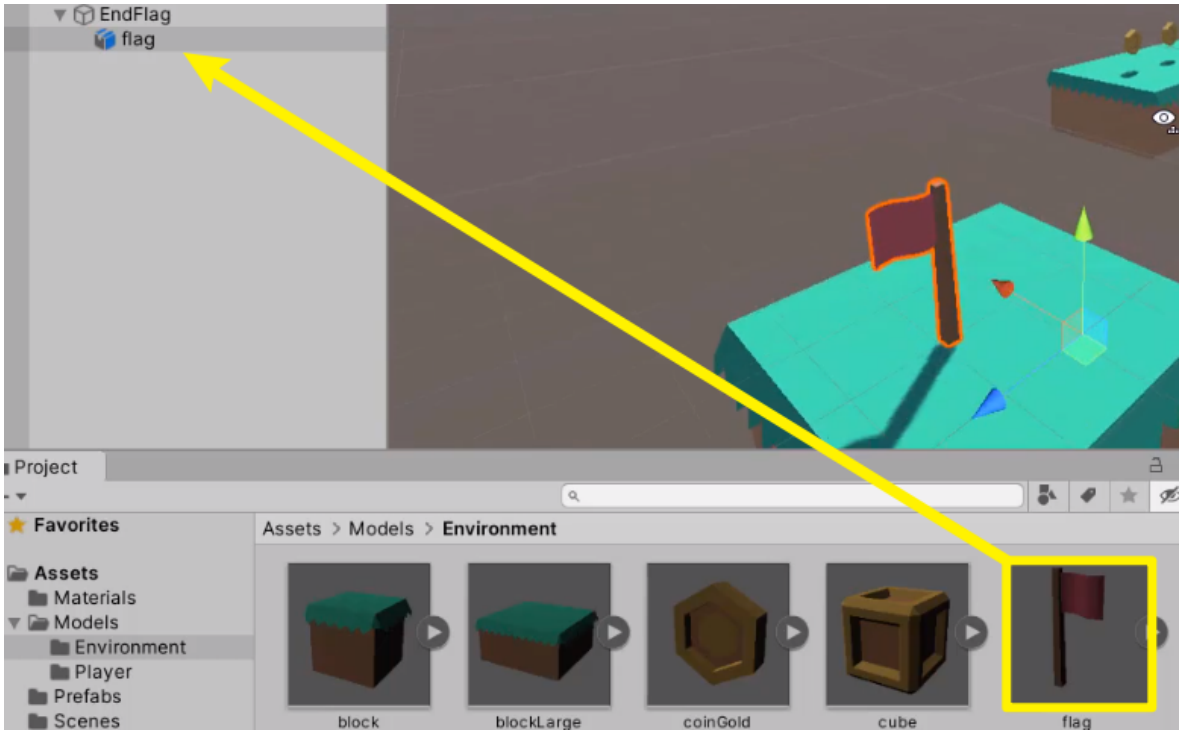




Let’s create a new empty game object, and name it “EndFlag”.

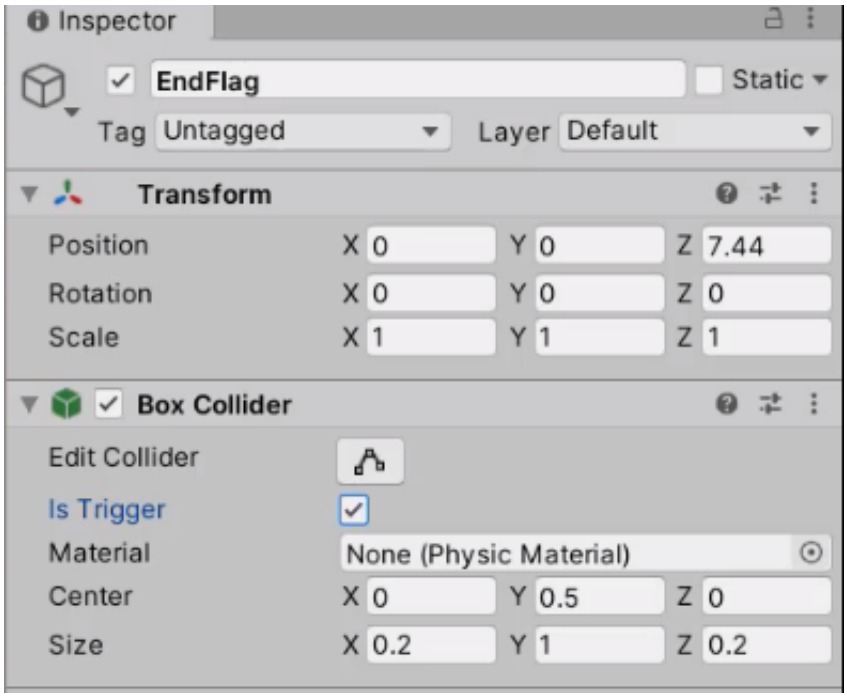
Importing A Flag Model

Go to the Environment folder (**Assets > Models > Environment**). Select EndFlag and drag our flag model in as a child object.

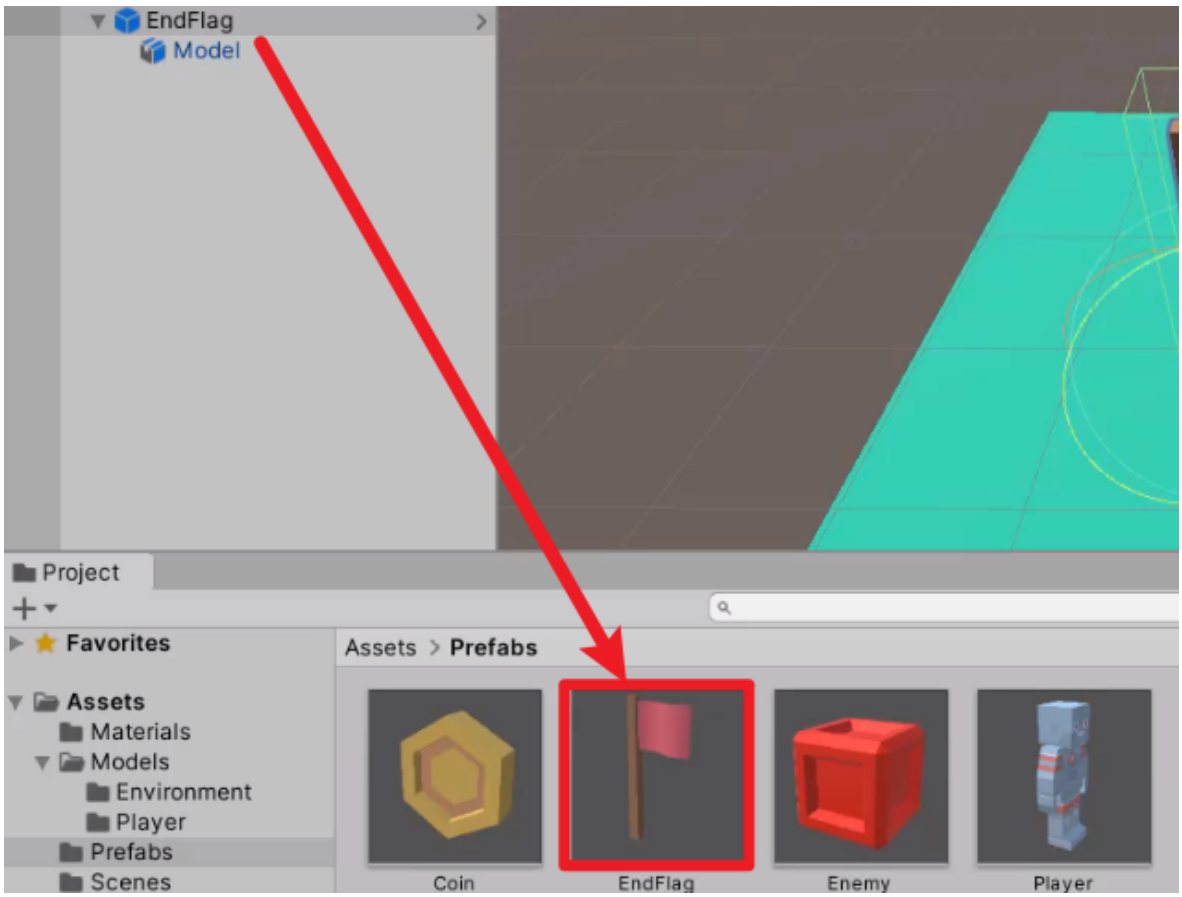


The default size may be too large, so we’re going to set the **Scale** as (0.1, 0.1, 0.1).

We’re going to add a **Box Collider** component to the parent object (EndFlag) and set it to be a trigger so our player can walk through it.



Now we can drag the End Flag into the Project folder (**Assets > Prefabs**) and save it as a Prefab.



Scripting End Game Functionality

Let’s open up the EndFlag script (**Assets > Script**) and create these variables:


```
public string nextSceneName;
public bool lastLevel;
```

The variables will be used in the **OnTriggerEnter** function, which detects if the flag has collided with an object tagged as “Player”:

```
private void OnTriggerEnter(Collider other)
{
    if(other.CompareTag("Player"))
    {
    }
}
```

We’re then going to check if this is the last level. If so, we want to go back to the menu scene. If not, we want to move on to the next level.

```
private void OnTriggerEnter(Collider other)
{
    if(other.CompareTag("Player"))
    {
        if(lastLevel == true)
        {
            Debug.Log("You win!");
            //To-do: Go back to the menu scene.
        } else
        {
            //To-do: Move on to the next level.
        }
    }
}
```

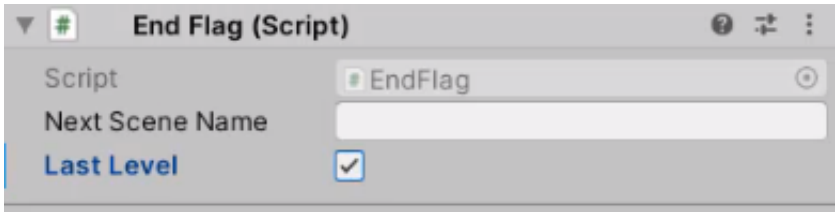
To implement **SceneManager**, we need to import the **SceneManagement** library at the top of our script:

```
using UnityEngine.SceneManagement;
```

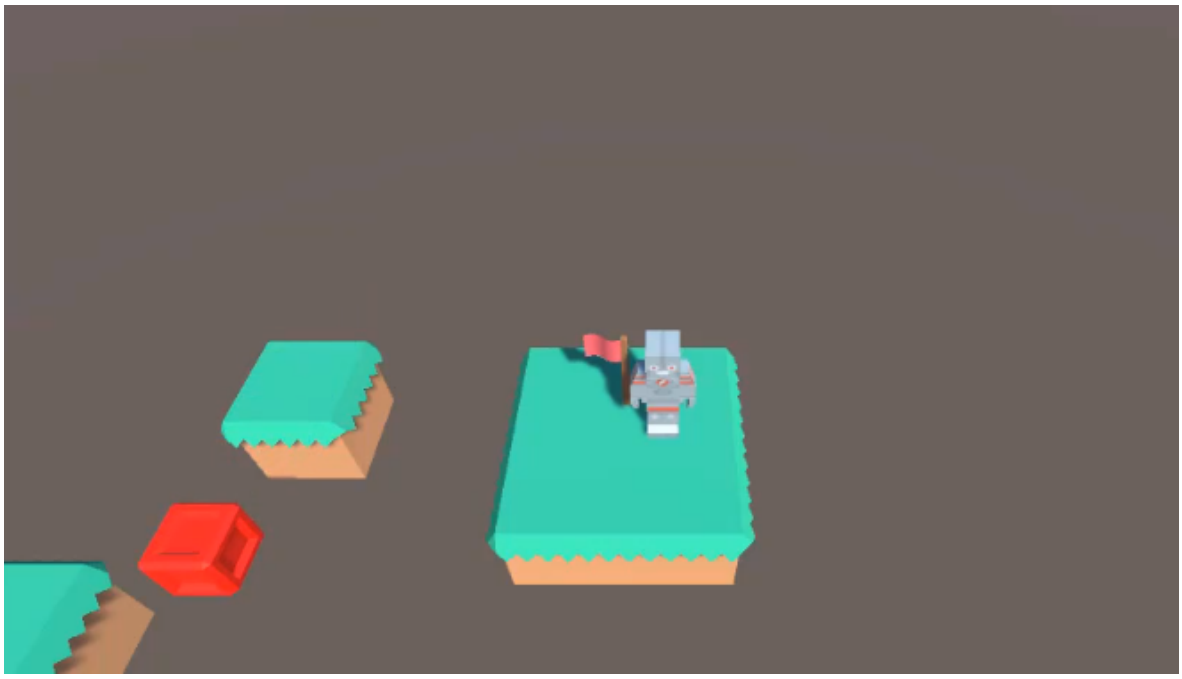
```
private void OnTriggerEnter(Collider other)
{
    if(other.CompareTag("Player"))
    {
        if(lastLevel == true)
        {
            Debug.Log("You win!");
            //To-do: Go back to the menu scene.
        } else
        {
             SceneManager.LoadScene(nextSceneName);
```

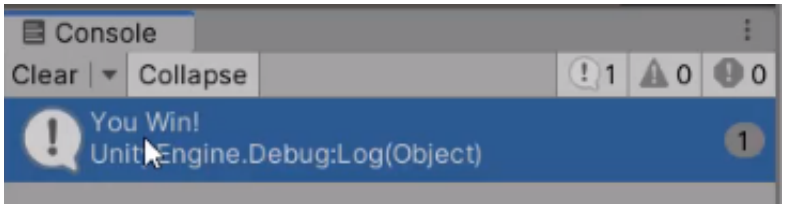
```
        }
    }
}
```

Currently, we only have one level available so we're going set the '**Last Level**' property to true in the Unity Editor.



Now if you touch the flag, you'll see the message ("You Win!") pops up on the Console.

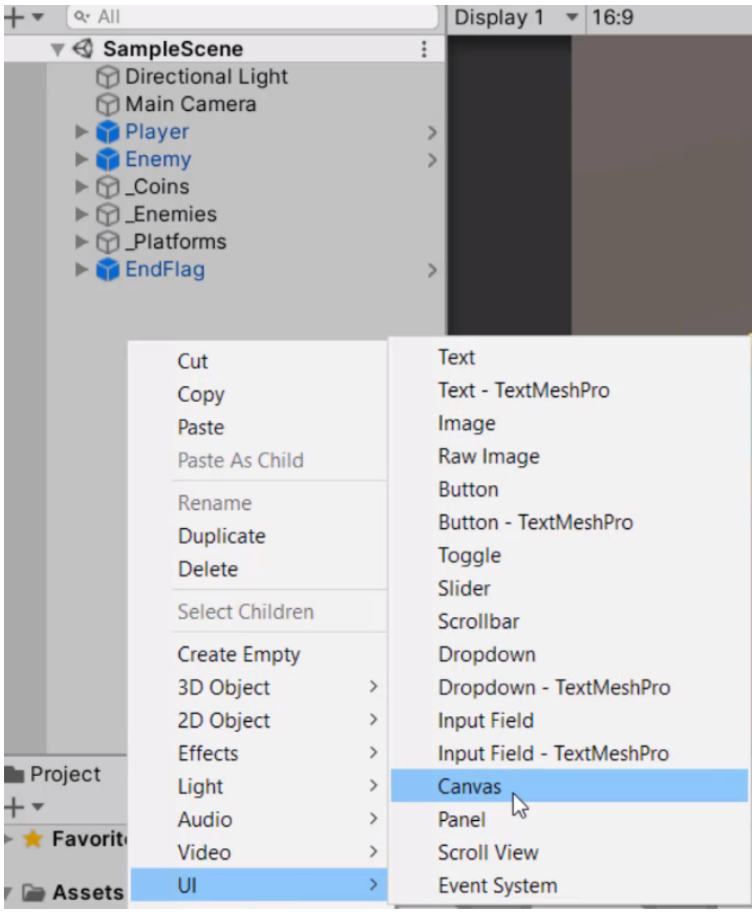




In this lesson, we're going to be implementing UI (User Interface) elements.

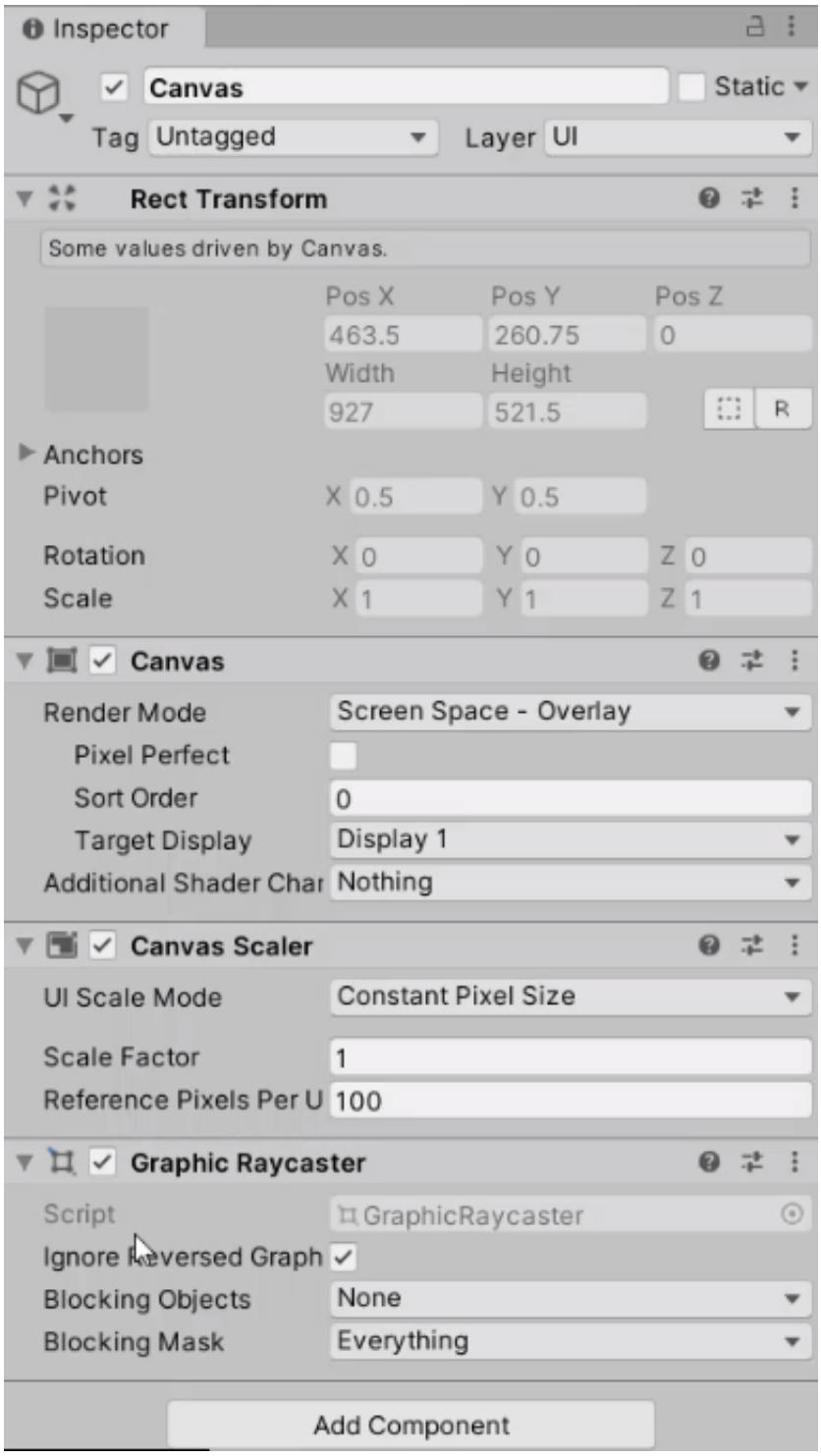
First, we're going to create a new gameObject called **Canvas,** which contains all of the UI elements.

To create a canvas, right-click on the **Hierarchy > UI > Canvas**.



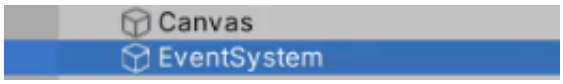
When you create any UI element object, a Canvas object will be created automatically if there isn't one in the scene already.

Canvas Components



As we create a new Canvas, the following components get automatically created, along with a new gameObject called **EventSystem**:

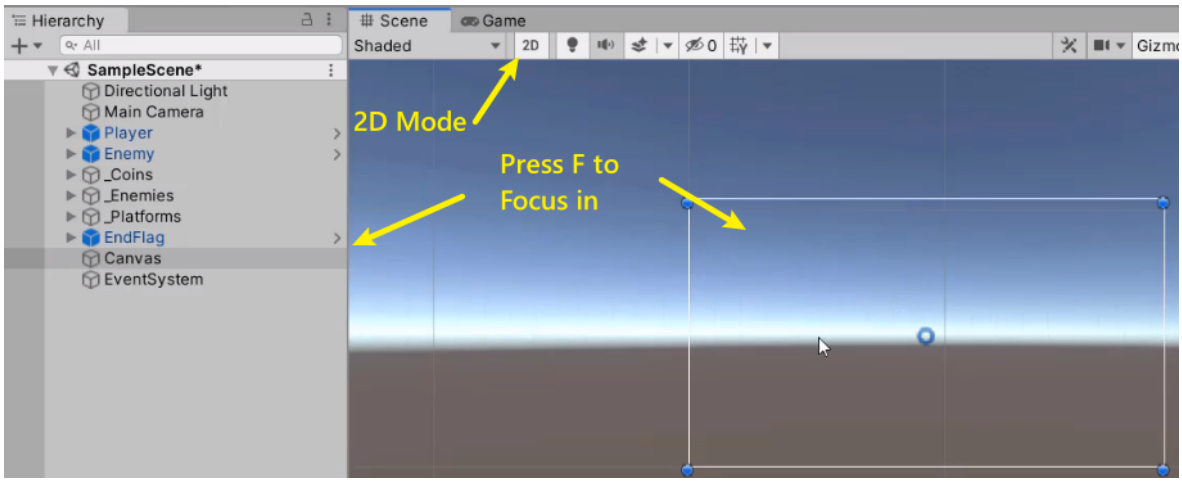
- The **Rect Transform** component is the 2D layout counterpart of the Transform component.
- The **Canvas** component allows the overall management of the Canvas object.
- The **Canvas Scaler** component is used for controlling the overall scale and pixel density of UI elements in the Canvas.
- The **Graphic Raycaster** manages click events by looking at all objects on the canvas and determines if any of them have been hit by a Raycast.



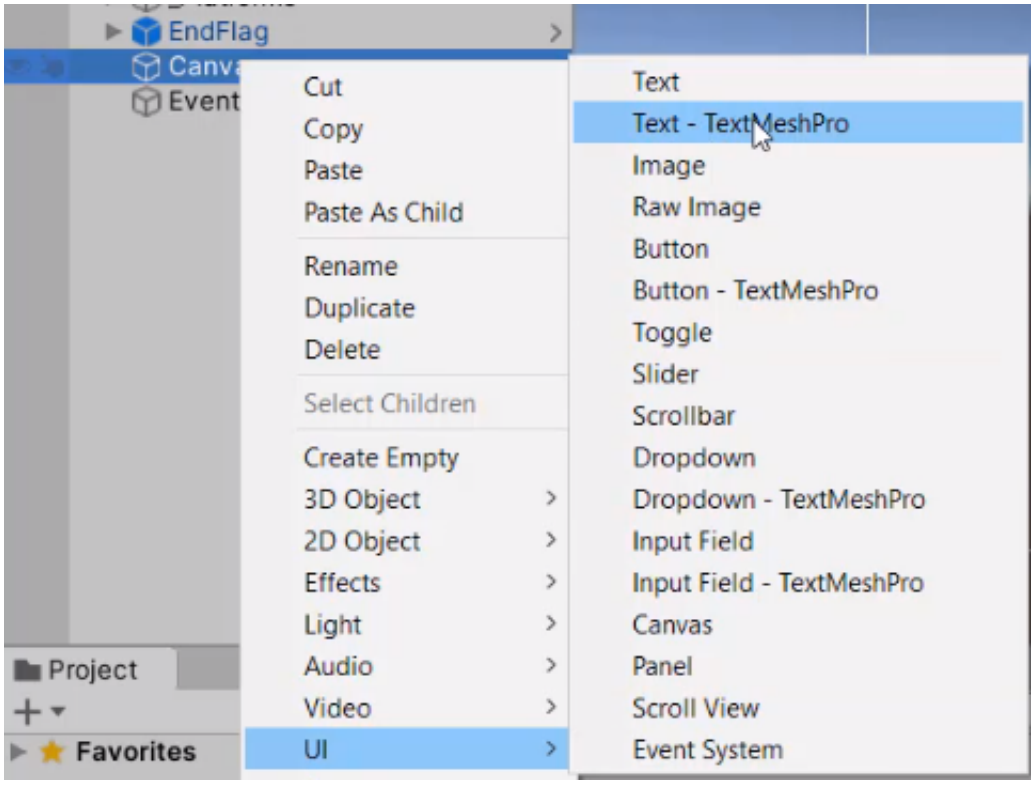
EventSystem is responsible for processing and handling events in a Unity scene.

Adding A Text Element

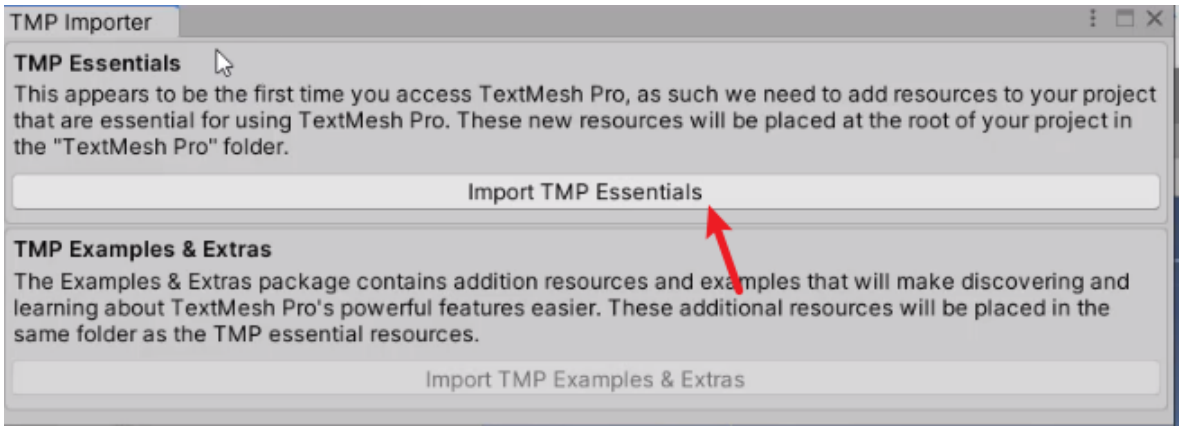
Let's select our Canvas, and enable **2D mode** in the Scene view (Press **F** to focus in on it).



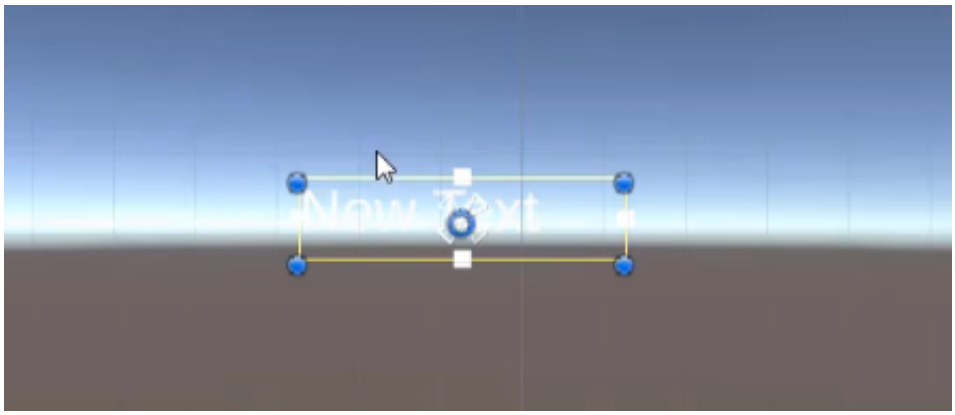
You can add a text element to the Canvas if you **right-click** on it, and go to **UI > Text – TextMeshPro**.



You may see this pop-up window. Click '**Import TMP Essentials**'.

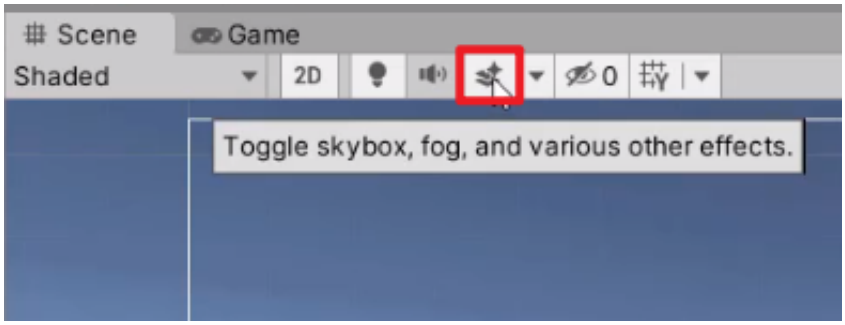


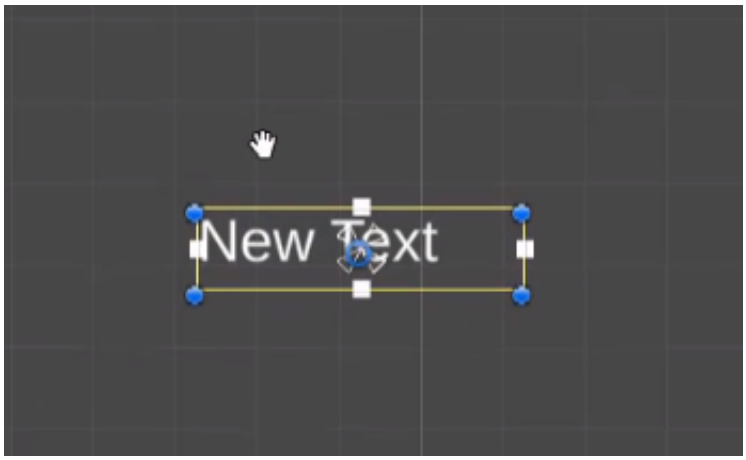
Once that's done, you'll see a text appearing in the Scene view.



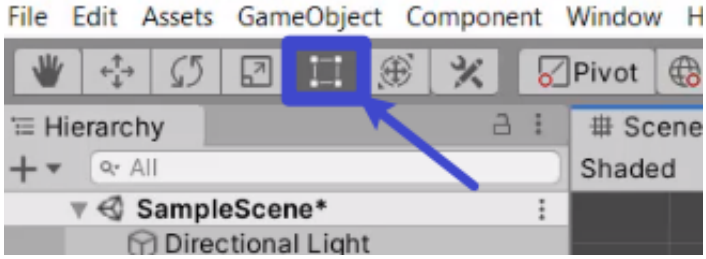
Modifying UI Element

You can **toggle skybox** on/off by clicking on this icon:

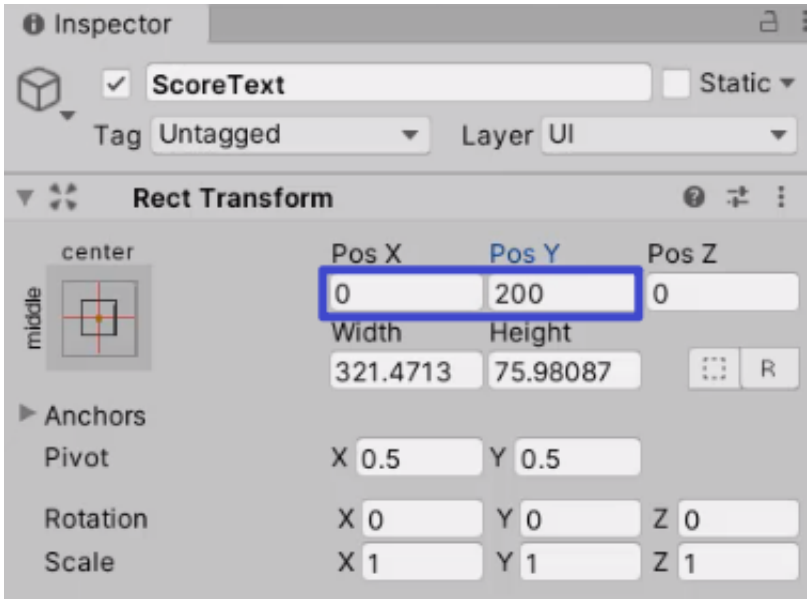




To change the position of the element, you can drag it around with the **Rect tool** selected:

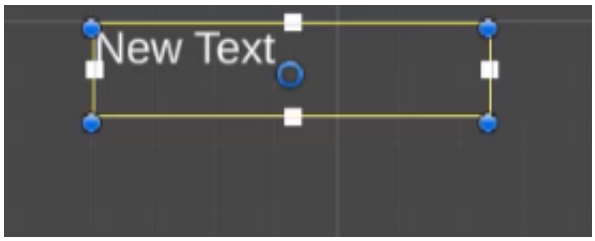


Alternatively, you can set the position using **Rect Transform** in the Inspector.

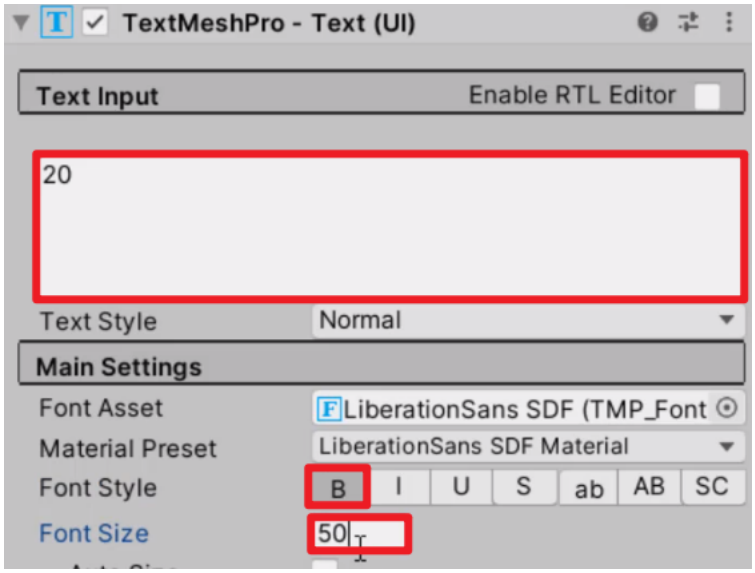


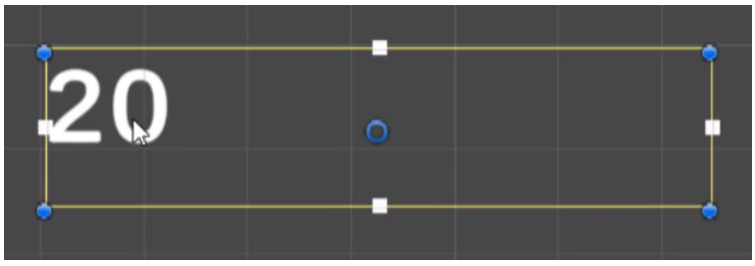
And you can click and drag on the blue circles to change the **size** of the element.

In the previous lesson, we have set up our UI canvas and a text object for player score.



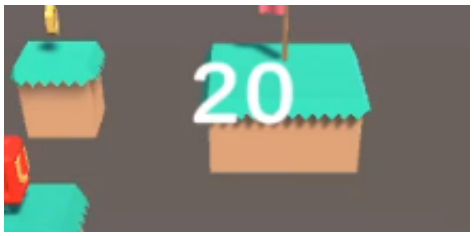
Let's change the **text input** to "20", select **Bold**, and increase the **font size** to 50.



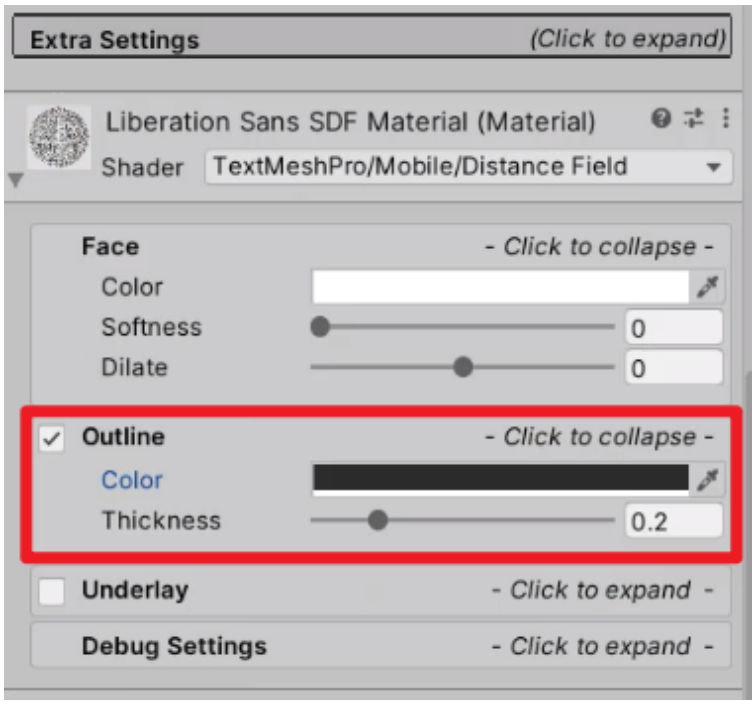


The text is currently aligned to the top right. Let's change it to be aligned at the **Center**.

You will notice that the text is blending in with the surrounding environment.



Let's try increasing the font size to 70, and enabling the **Outline** in 'Extra Settings'.



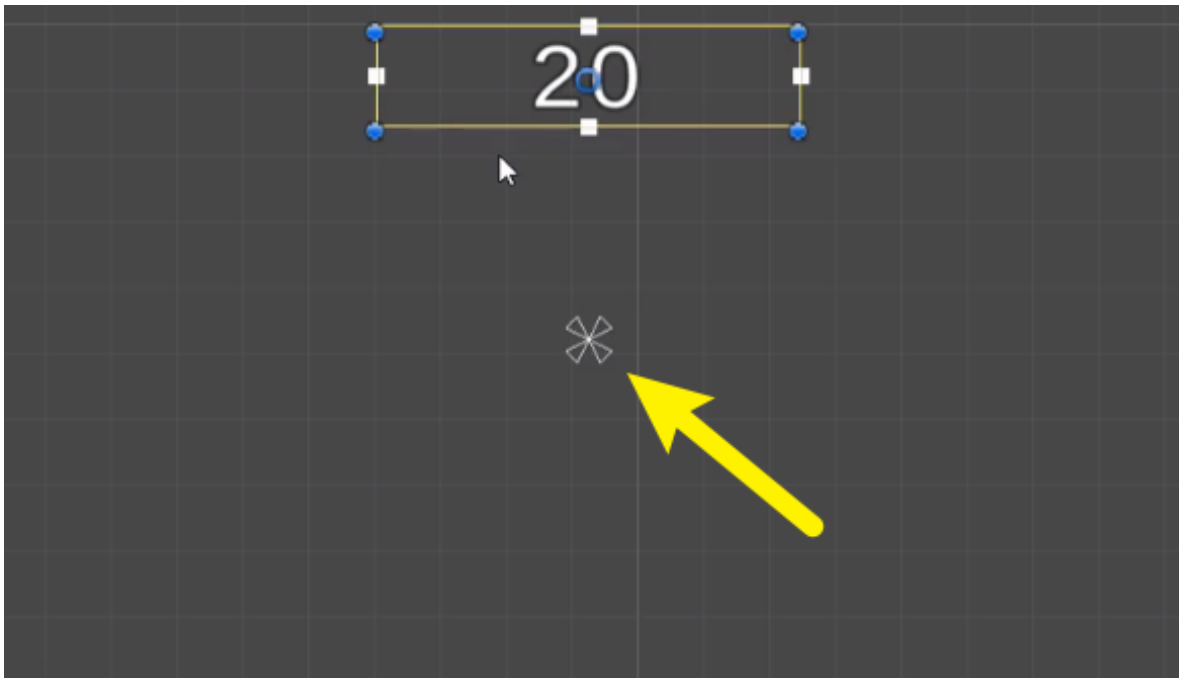
As you can see, the text is now much more visible.

Modifying Anchor Point

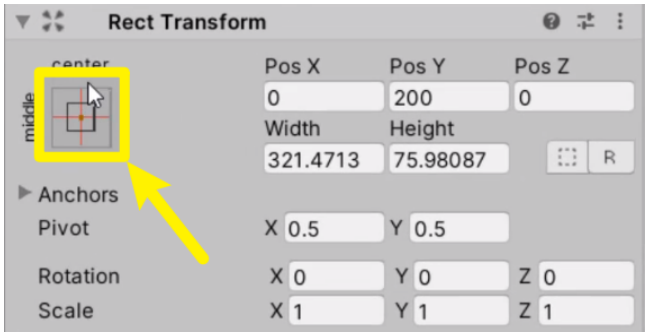
The **Anchor Handles** are represented by four triangles as shown below:

These handles ensure that our UI element is positioned and scaled appropriately to fit varying resolutions (or aspect ratios).

The anchor handles of our text are currently positioned at the center of the screen.

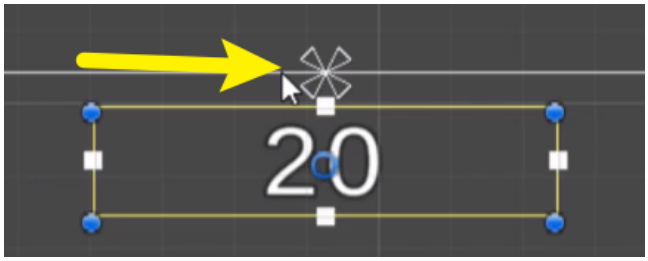


To change their position, you can click on this icon in the **Rect Transform** component.



We can simply click on one of the presets to change our anchoring:

Now our anchoring is at the top center.



Creating UI Script

Let's go ahead and create a new C# script called "UI".

First, we need to import the TextMesh Pro library (**TMPro**) at the top of our script:

```
using TMPro;
```

And declare a new **TextMeshProUGUI** variable to reference our score text.

```
public TextMeshProUGUI scoreText;
```

We're then going to create a new function called "**SetScoreText**", that takes in an integer variable for the score to set.

```
public void SetScoreText (int score)
{

}
```

This function will set our score text (string) to be the same as the score (integer). Note that we need to convert the score (integer) to be string by using **ToString()**:

```
public void SetScoreText (int score)
{
    scoreText.text = score.ToString();
}
```

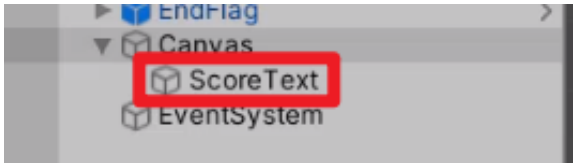
Now, let's go over to our **Player** script and create a variable to reference our UI script:

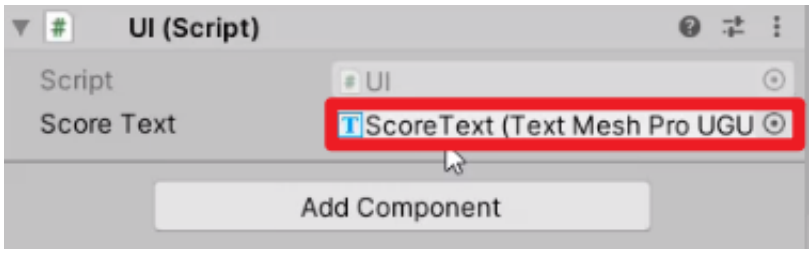
```
public UI ui;
```


And we're going to call the **SetScoreText** from the UI script whenever the AddScore function is called.

```
public void AddScore (int amount)
{
    score += amount;
    ui.SetScoreText(score);
}
```

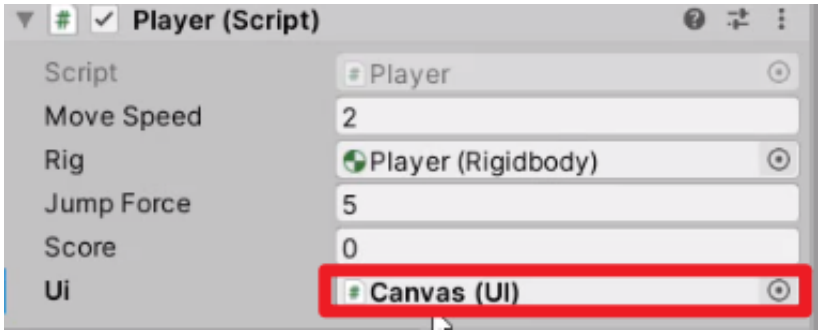
After saving the script, let's drag the ScoreText game object into the Score Text field of our UI script.





We're going to set the default value of our score text to 0,

...and drag our canvas object into the UI field of our Player script.



Now we should see that the score text goes up as we pick up more coins.

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

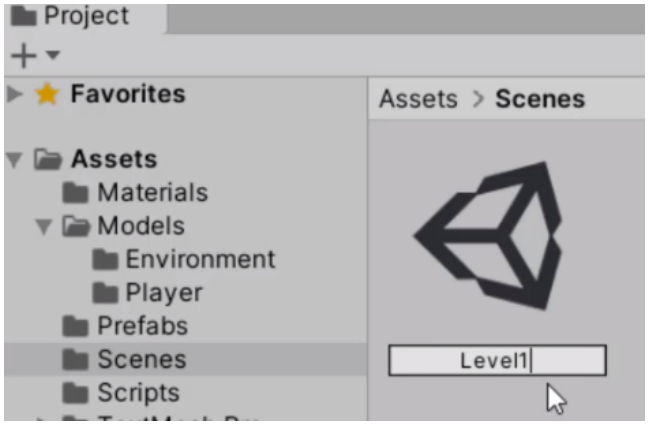
In this lesson, we're going to be setting up a menu scene.

Creating A New Scene

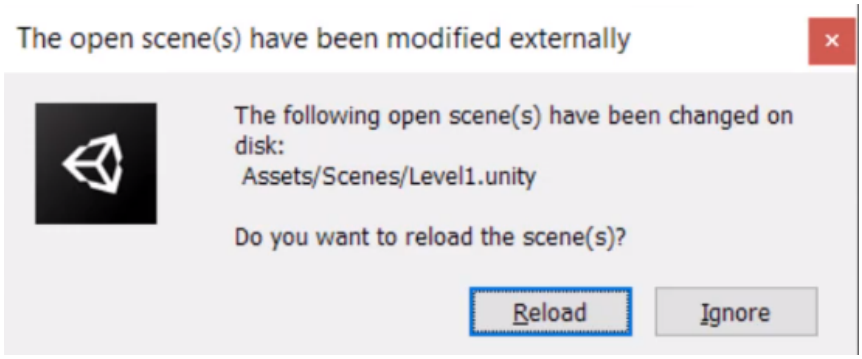
The following instruction has been updated, and differs from the video:

Before continuing with the following step, which requires renaming the scene, be sure to **Save** (select Save Scene from the file menu, or hit Ctrl/Cmd + S). Any unsaved changes in your scene will be **lost when renaming** and reloading an **unsaved Scene!**

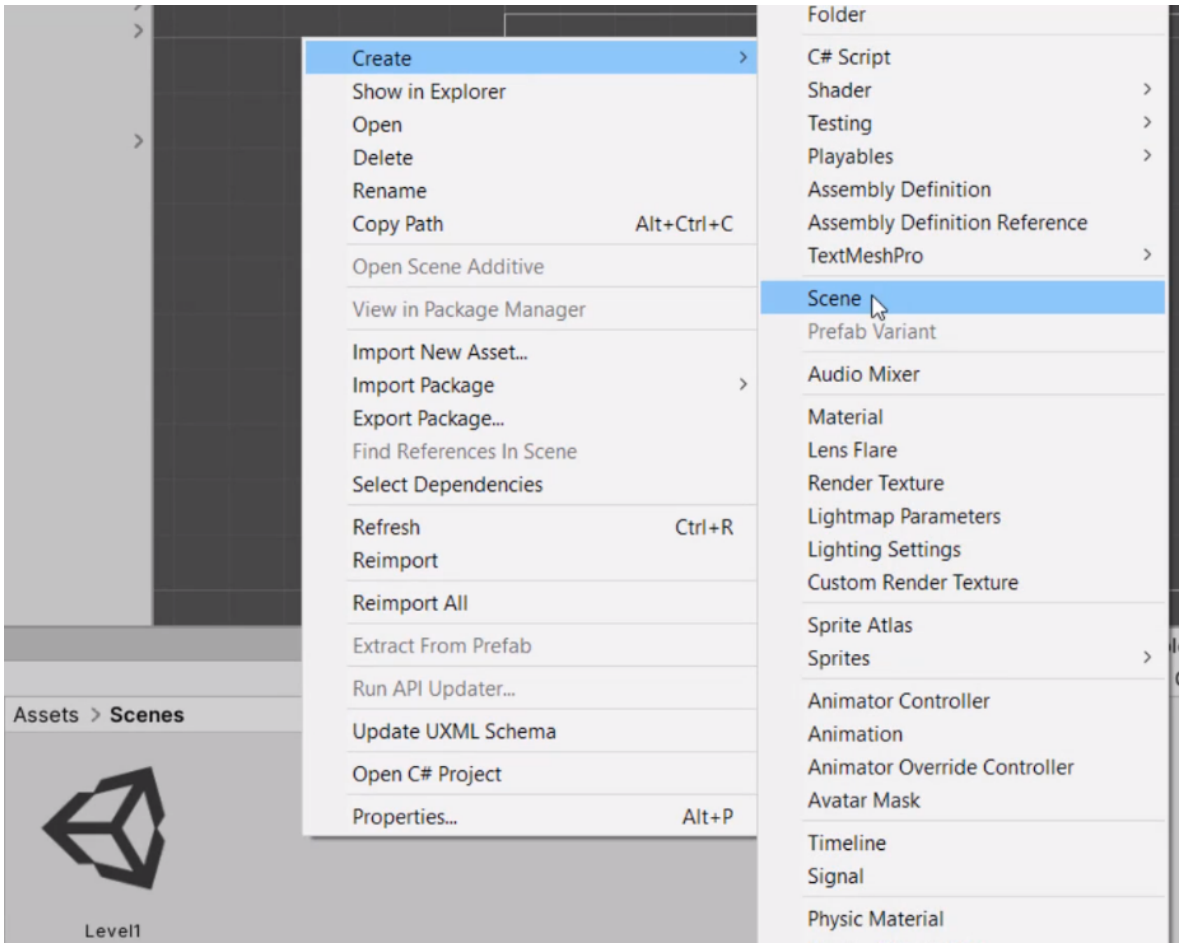
Let's go over to the Scenes folder (**Assets > Scenes**), select the current scene (SampleScene.unity), and rename it to 'Level1'.



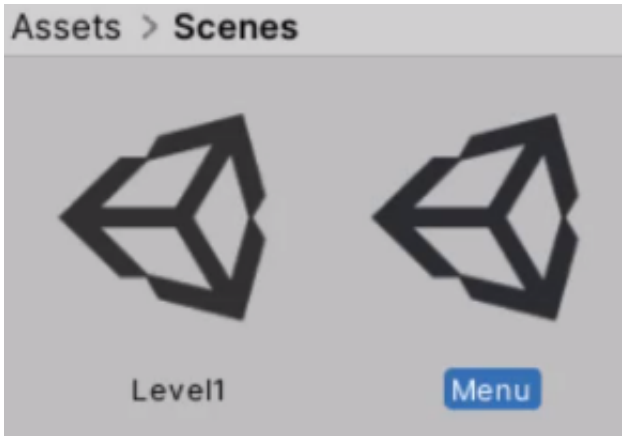
Click 'Reload' if the following prompt pops up.



Now we're going to create a new scene by Project (Right-click) **> Create > Scene**.

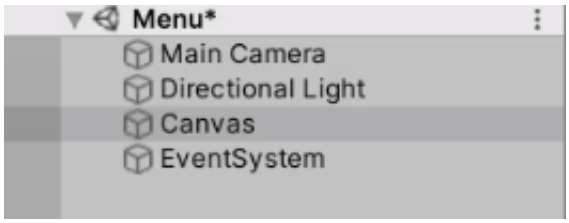


We're going to call this one "Menu", and double-click to open up.

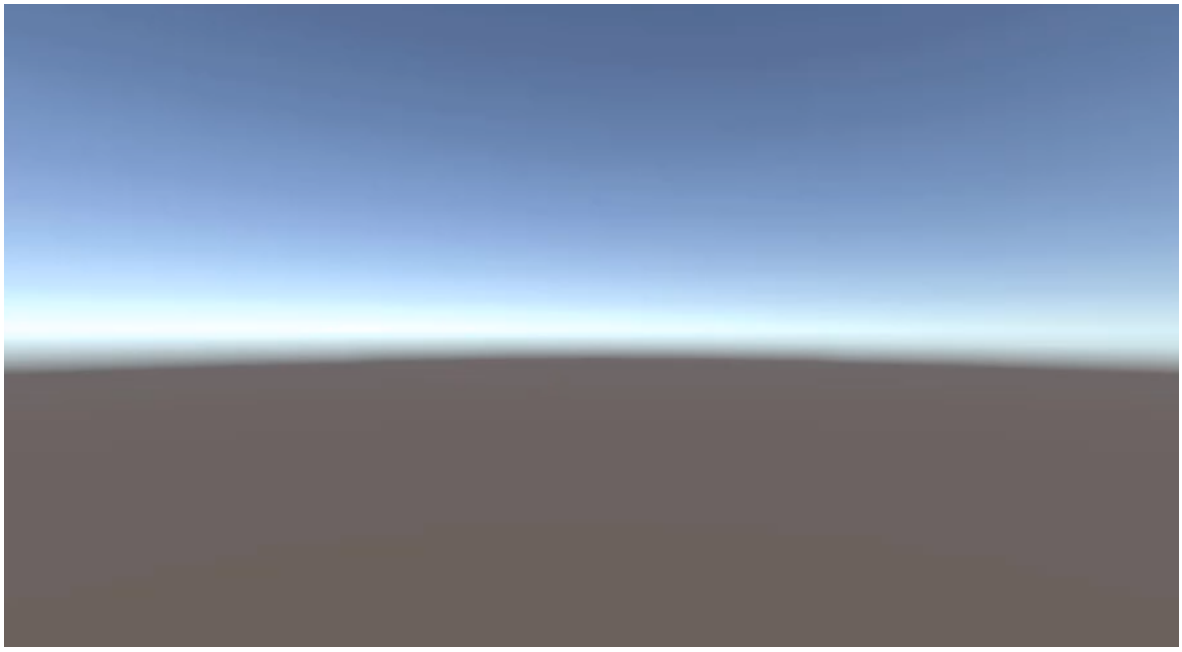


Setting Up Menu UI

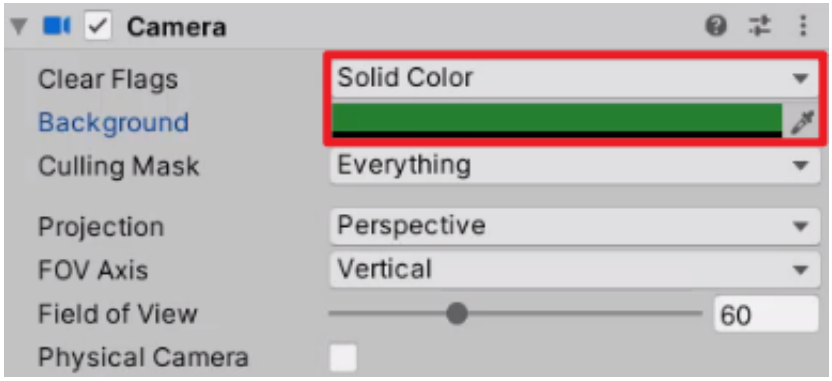
Let's create a new canvas game object by going to **Create > Canvas**.

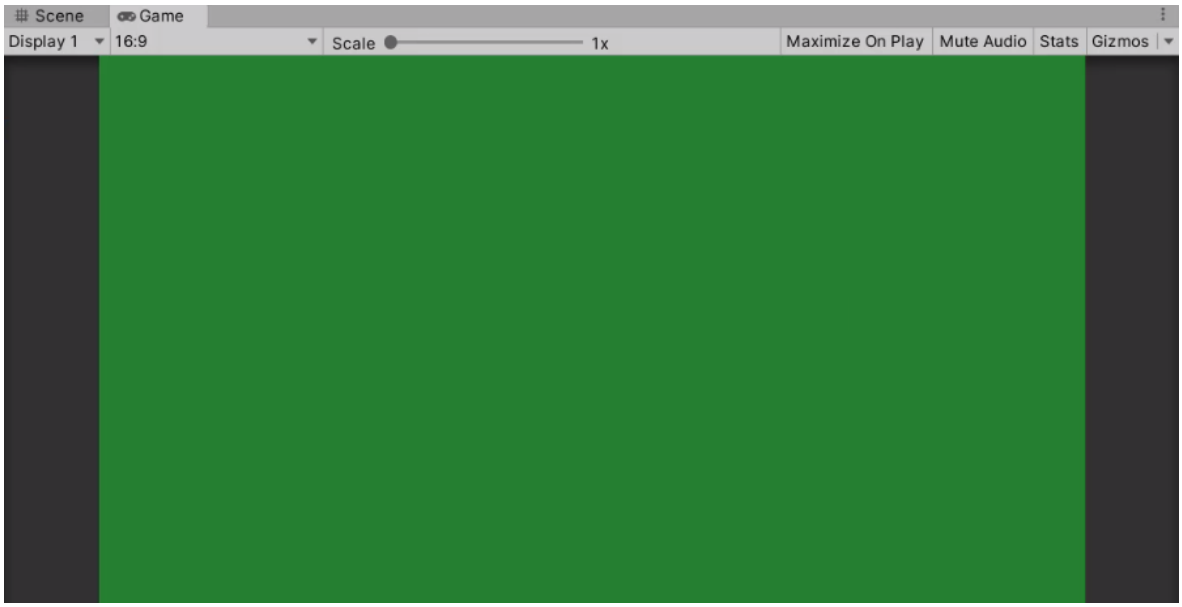


Right now our scene's background is set to the default **Skybox**.



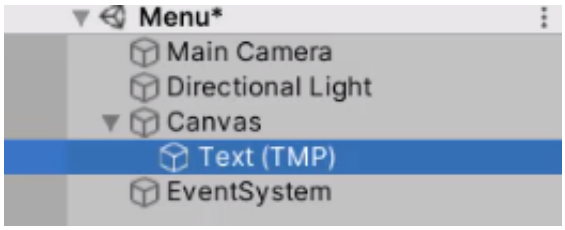
To change the background to a flat color, we can select our camera and change the **Clear Flags** property to "Solid Color".



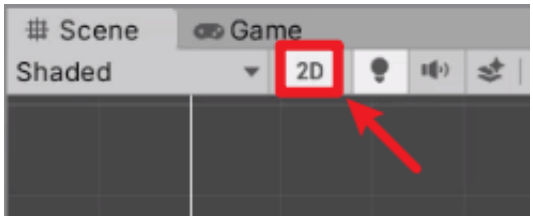


Creating A Title

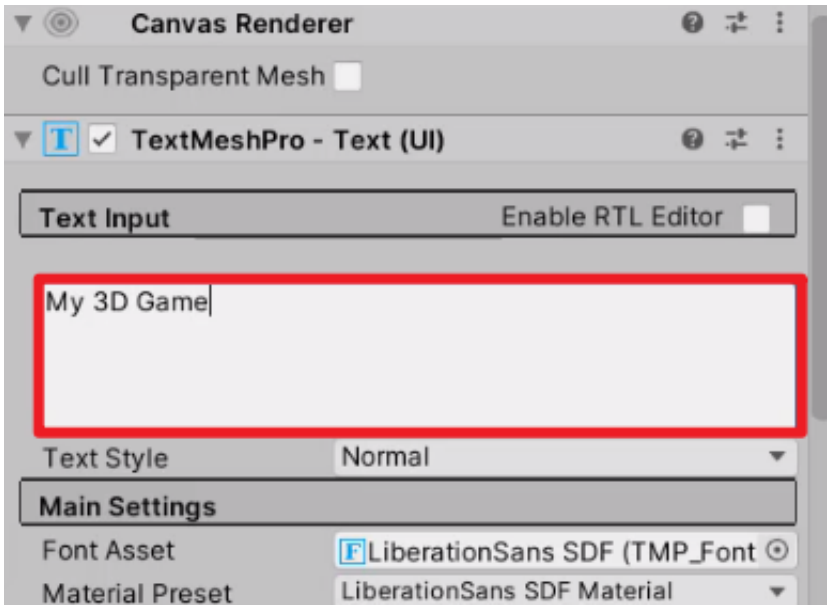
Let's create a new **Text** object as a child to our Canvas (Right-click on Canvas > **UI > Text**). We'll rename this to "Title".



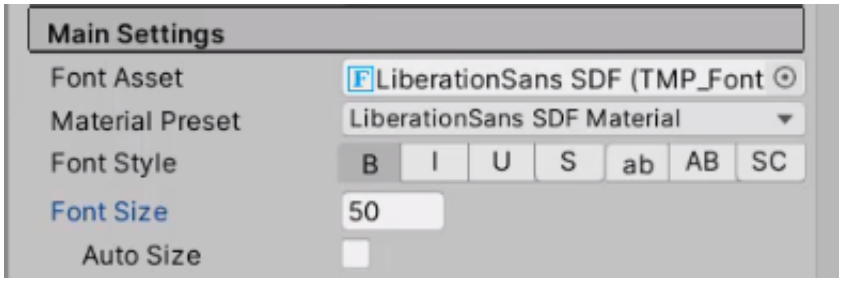
We're going to switch our Scene view to **2D mode:**



And change the **text input** to "My 3D Game".

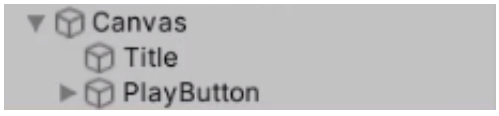


Make sure to **align** it to the center, and increase the **font size**.



Now we have a simple title text for the main menu.

We're going to create a new **button** object to our Canvas, and call this "PlayButton".

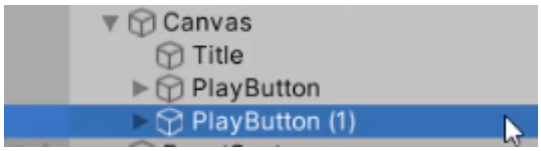


Let's place the button under the title, and adjust its size:

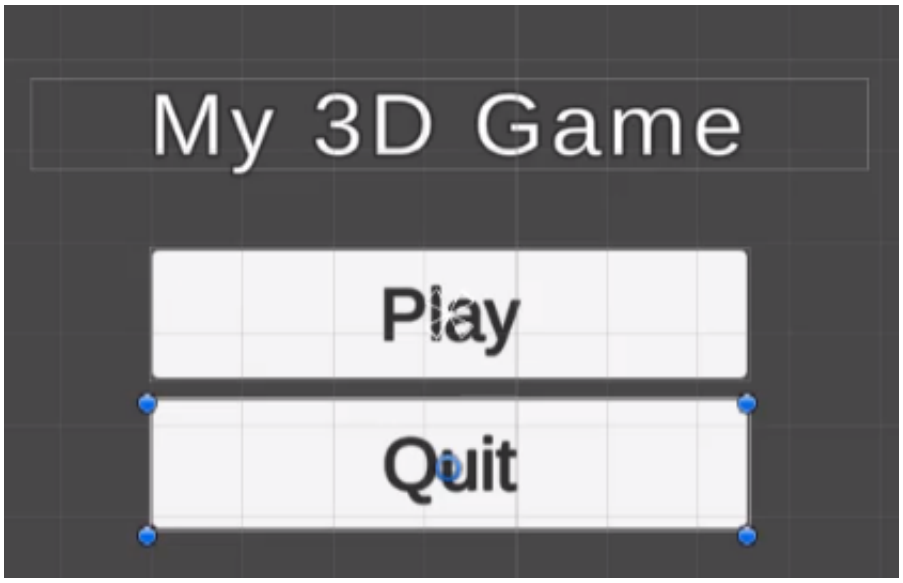
We can now press **Play** and try clicking on the button.

You will notice that the button responds to clicks, but nothing happens. (In the next lesson, we're going to connect this button to a script and call a function.)

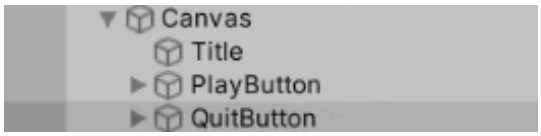
Let's **duplicate** the button (Ctrl+D),



...and set their text inputs to "Play" and "Quit" respectively.



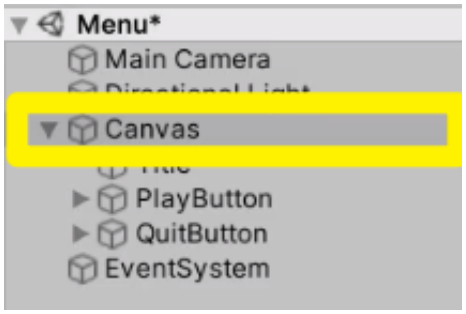
Make sure to change the name of the second button to “QuitButton”.



Setting up Menu Script

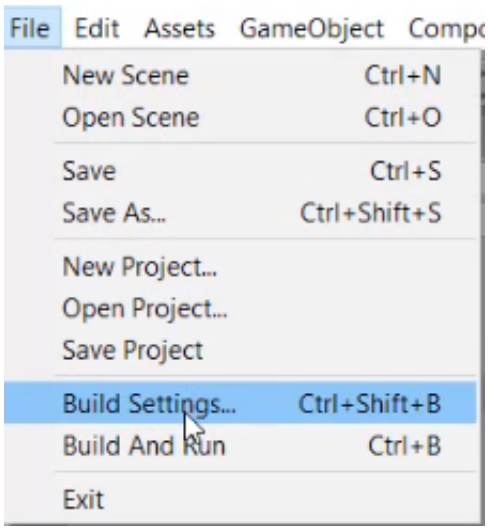
Let’s create a new **C# script** called “Menu” inside **Assets/Scripts**.

We’re going to **attach** this script to our **Canvas,** by selecting our Canvas…



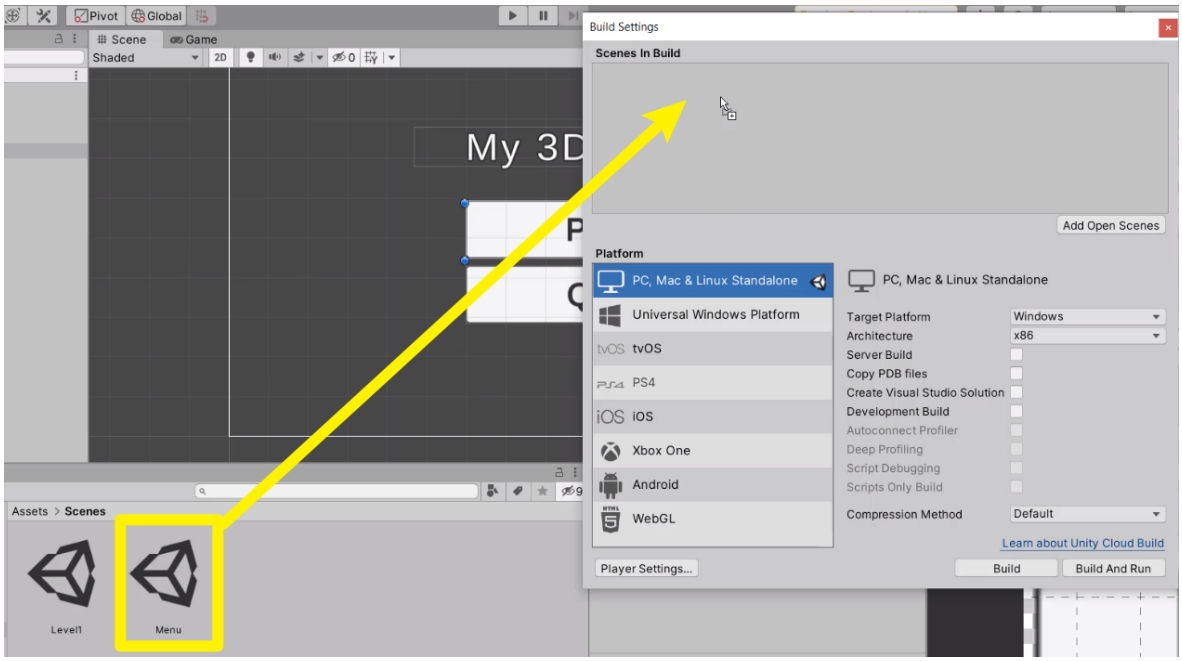
… and dragging it into the Inspector.

Now, in order for the menu script to load our scenes, we need to first add the scenes to the **Build Settings (File > Build Settings...)**.

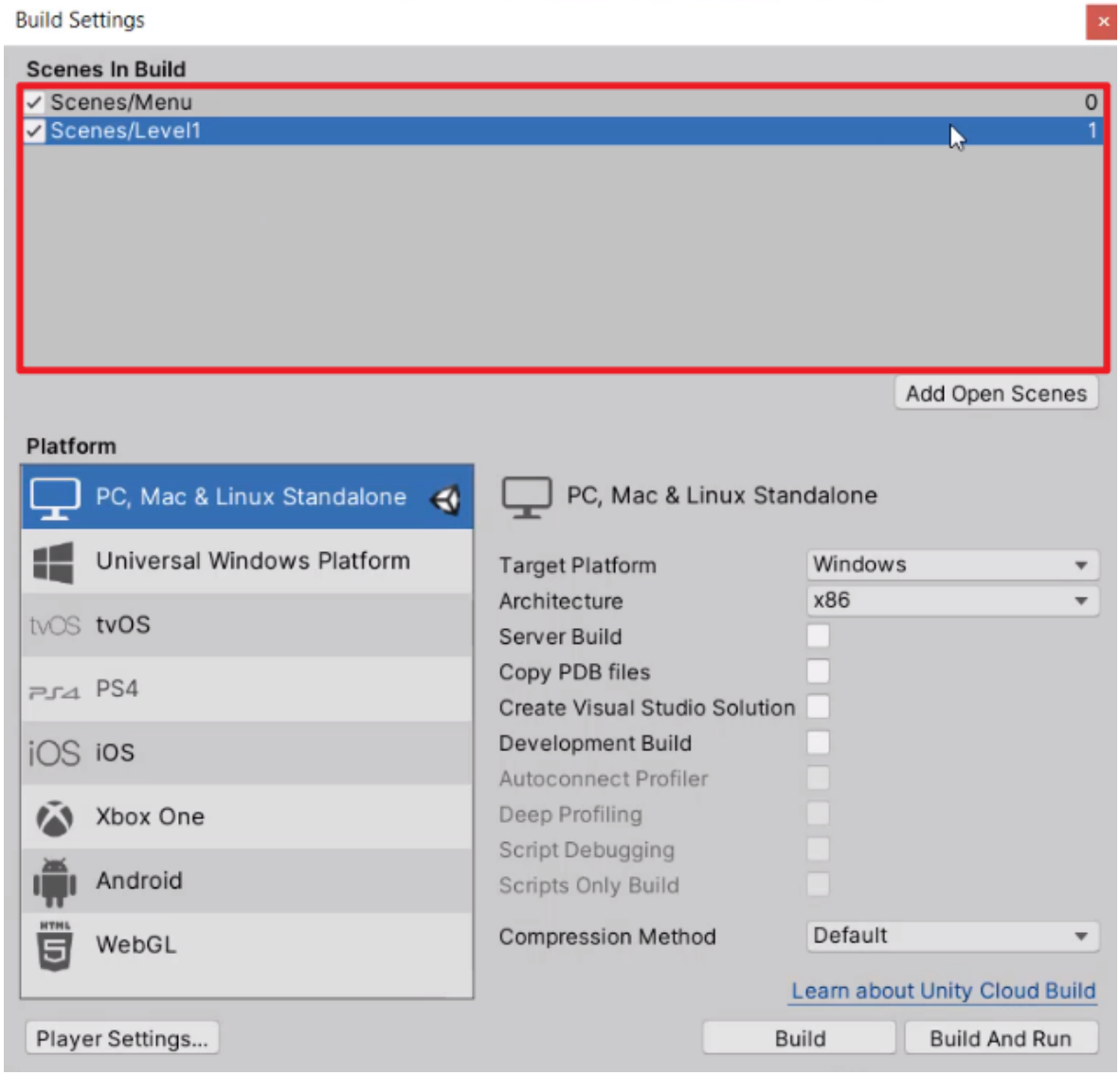


We need to include all the scenes that we want our game to have within '**Scenes In Build**':

To add scenes, simply drag and drop the scene files into the highlighted box.

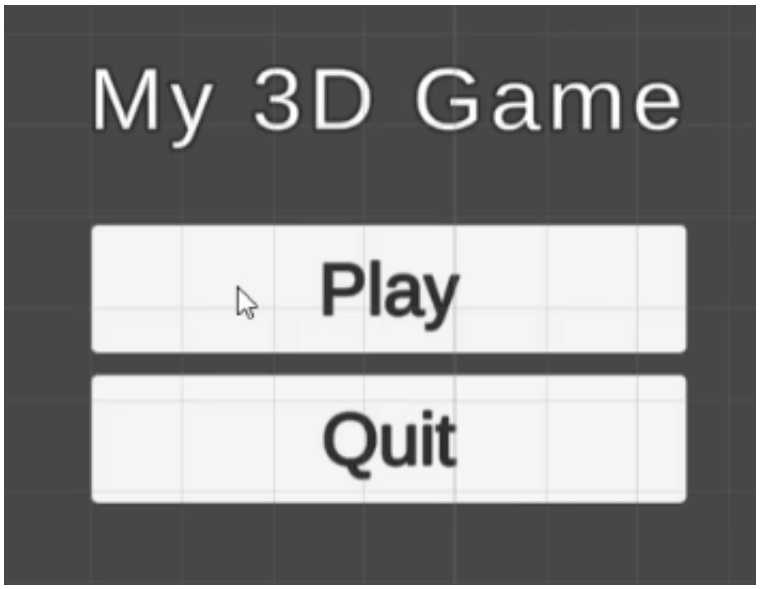


Note that the build index (e.g. Menu: 0) is displayed on the right side. We can now switch between these scenes by referring to their index number.



By default, the game will launch into the first scene that is marked as index 0.

Now we have our menu UI with buttons. It's time to link these buttons with our Menu script.



Creating Button Click Events

Let's create two functions: one for the **Play** button, and one for the **Quit** button. These functions will get called whenever we press the buttons.

```
public void OnPlayButton ()
{

}
public void OnQuitButton ()
{

}
```

Inside these functions, we're going to be loading scenes using **SceneManager**:

```
public void OnPlayButton ()
{
   SceneManager.LoadScene(1);
}
public void OnQuitButton ()
{

}
```

SceneManager requires the **SceneMangement** package to be imported:

```
using UnityEngine.SceneManagement;
```

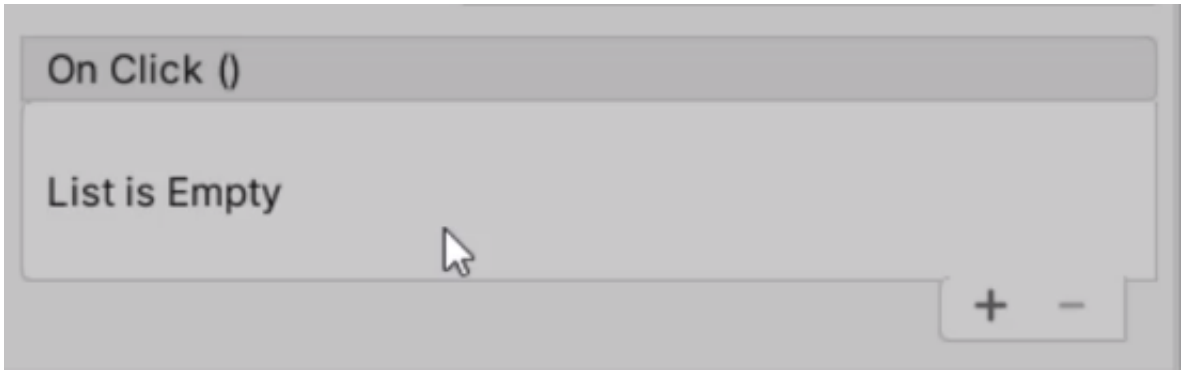
For the Quit button, we're going to call **Application.Quit()** as it shuts down the running application.

```
public void OnPlayButton ()
{
   SceneManager.LoadScene(1);
}
public void OnQuitButton ()
{
   Application.Quit();
}
```

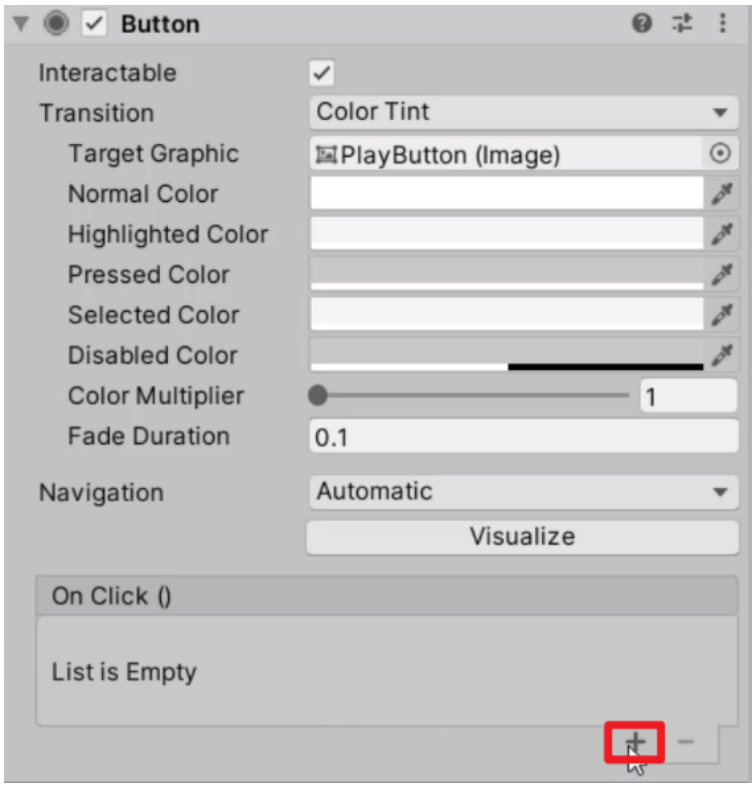
Make sure that the functions are **public** since we'll be setting it in the inspector. Let's save it and return to Unity Editor.

Linking Button To Script

We'll select the PlayButton game object, and add an event to the **OnClick()** field inside the **Button** component.

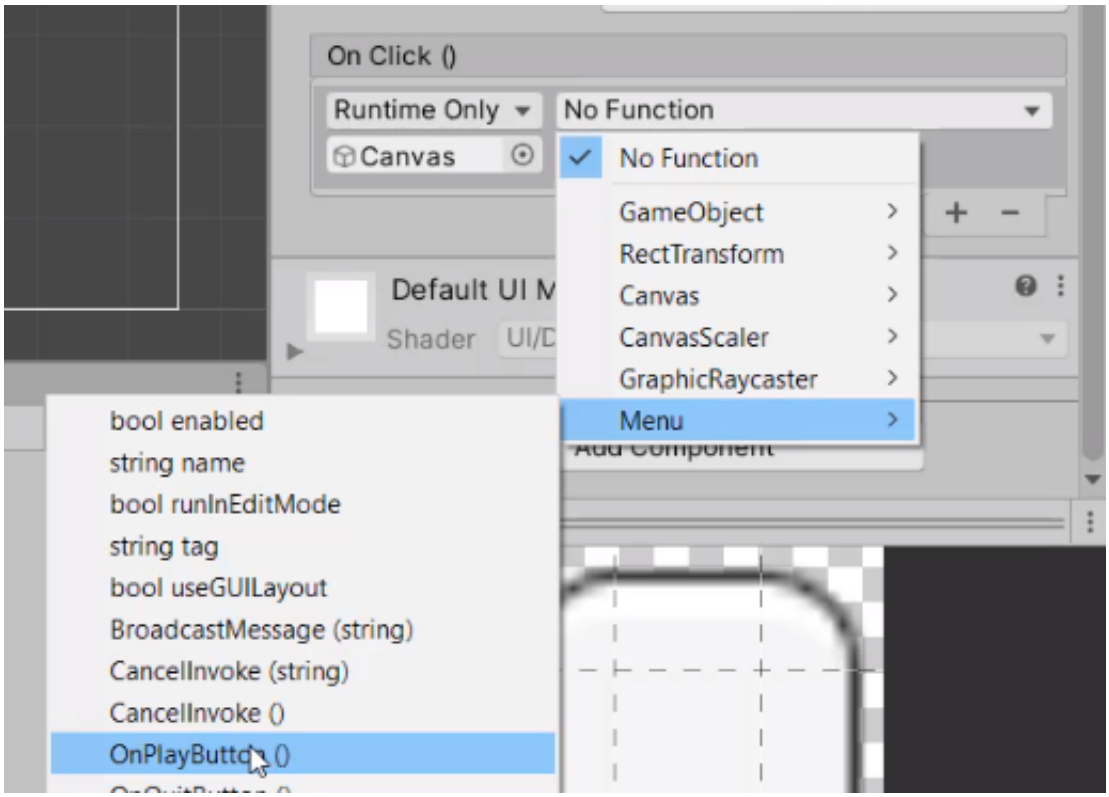


This is a list of various different functions that gets called once the button is pressed. We can add an event by pressing this plus icon:



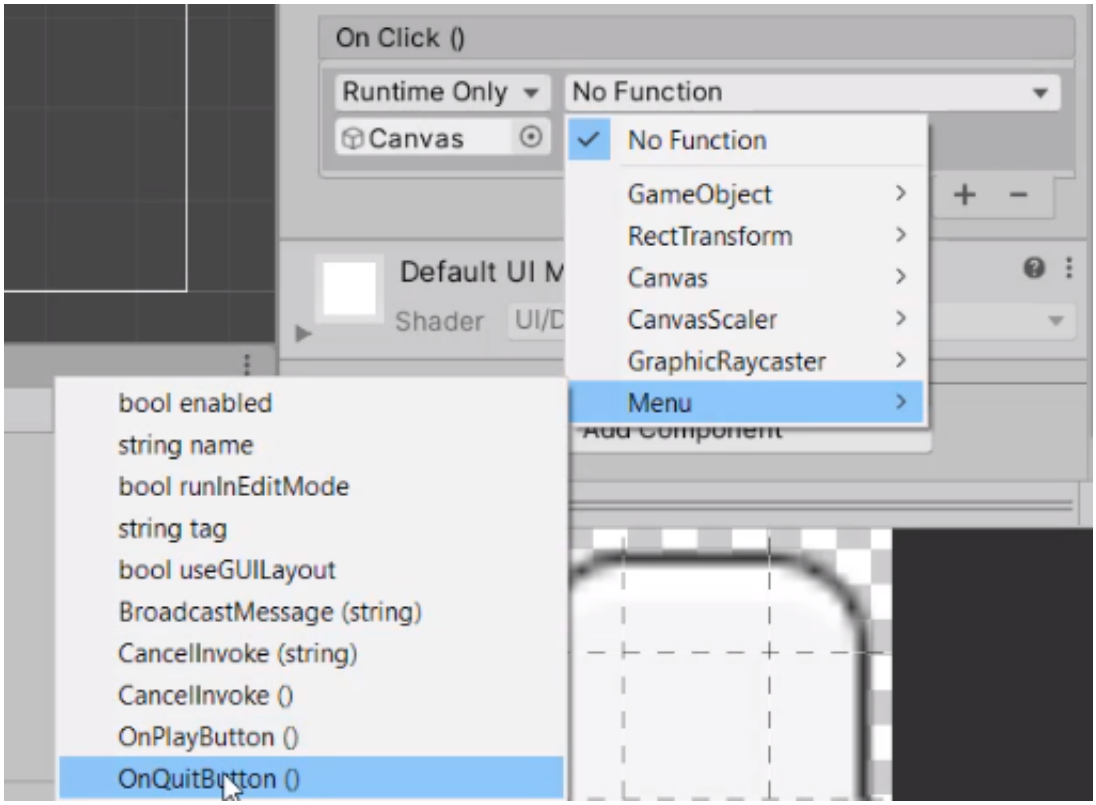
This will create a new field, where we can drag any object into.

We're going to drag our Canvas object into this field, and select **Menu > OnPlayButton()**.



When we press on the Play button (in Play mode), it leads us to the “level1” scene.

Now let’s repeat this step for the Quit button. We’re going to drag our Canvas object into the object field, and select **Menu > OnQuitButton()**.



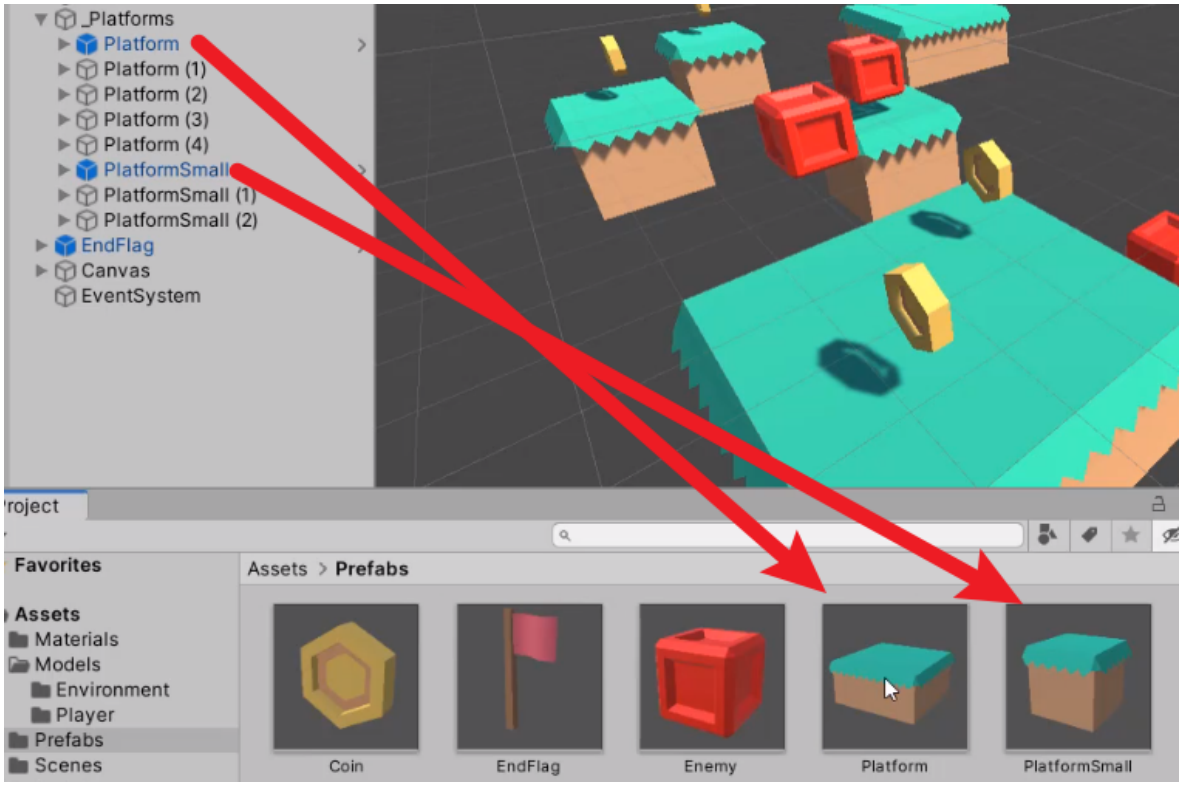
You’ll see that the Quit button doesn’t do anything when we are inside of the editor. However, once

we build this game (e.g. as an application), it will quit the game.

In this lesson, we're going to go over how we can create extra levels.

Making Prefabs

First of all, check if there is anything that you want to share across levels or have multiples of, and turn it into a **prefab**. This will save us time spent on recreating the same object.



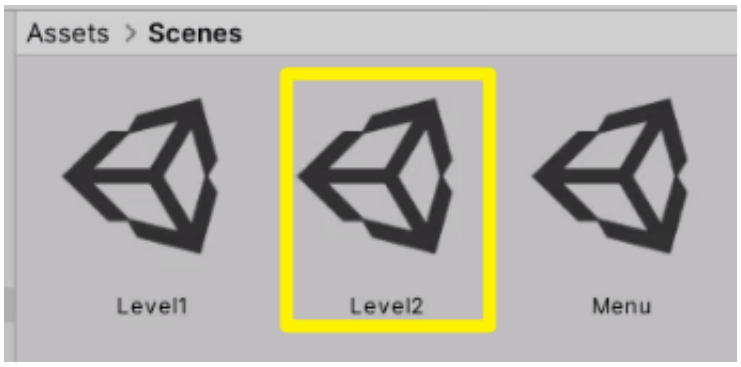
We're going to turn the following objects into prefabs:

- Main Camera
- Player
- Enemy
- Coin
- Platform
- PlatformSmall
- Score Canvas
- EndFlag

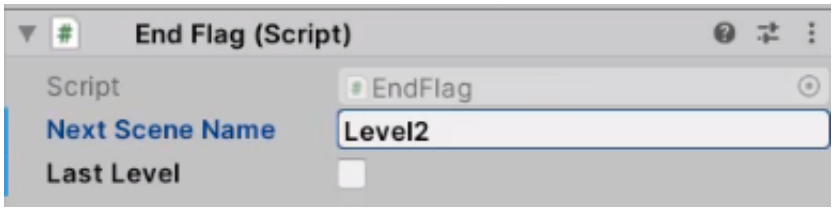
Remember to make the current Main Camera as a prefab, since it has the **CameraFollow** script on it.

Creating A New Scene

Let's create a new scene by going to **Project (Right-click) > Create > Scene**. We'll call this scene "Level2".



Now let's select our **EndFlag** game object, set its **Next Scene Name** to "Level2".



Make sure to set the **Last Level** checkbox to false as well, because now this scene (Level1) is not the last level anymore.

It is important to include Level2 in our **Build Settings** (File > Build Settings).

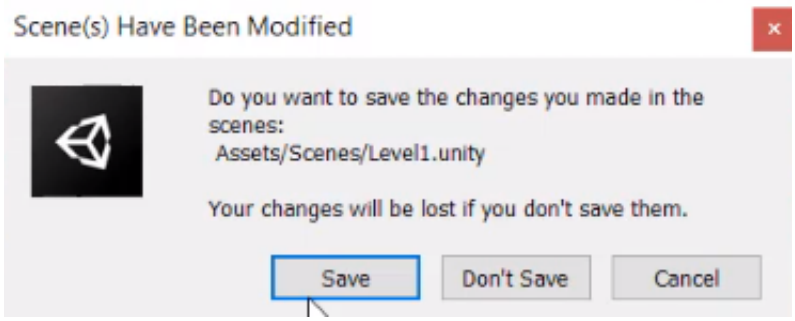
Let's open up **EndFlag** script, and replace the Console.Log("You win!") with the actual **LoadScene** function.

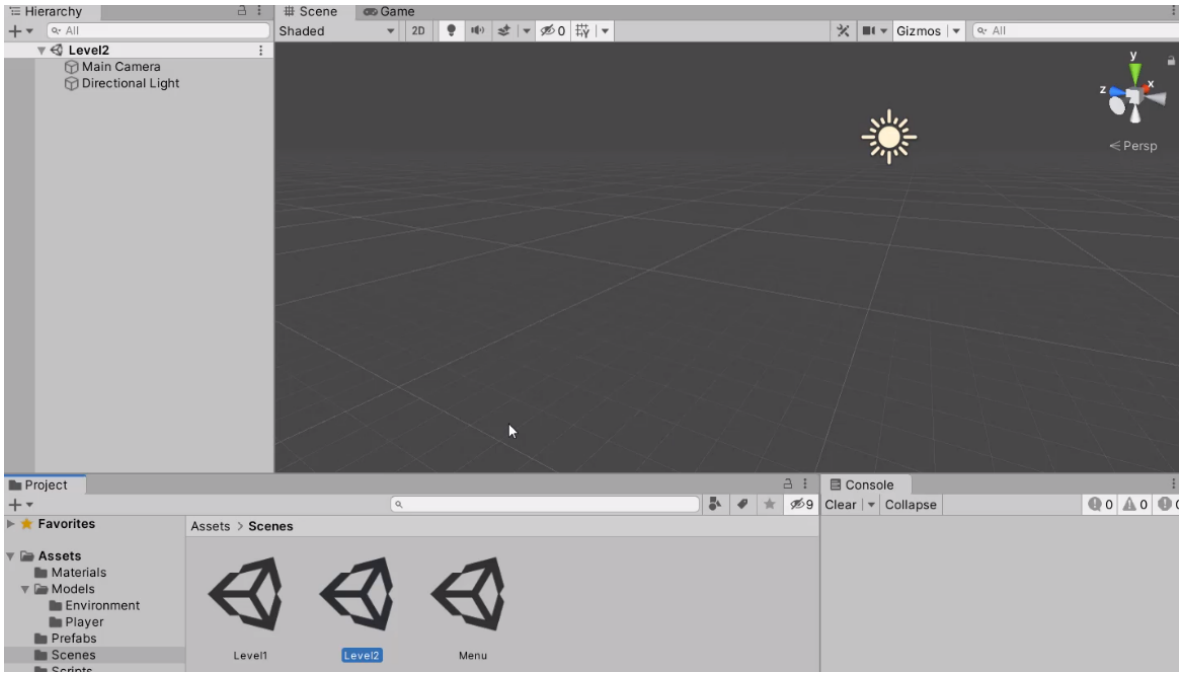
```
if (lastLevel == true)
{
    SceneManager.LoadScene(0);
}
```

Now when we reach the flag, level2 is going to be loaded in.

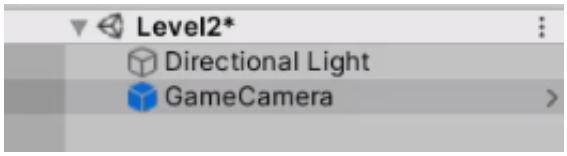
Setting Up A New Level

Make sure you save the changes of Level1 and open up Level2.

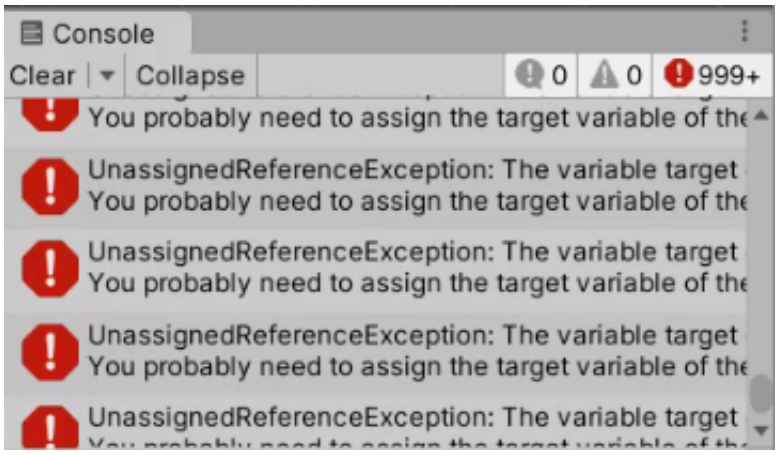




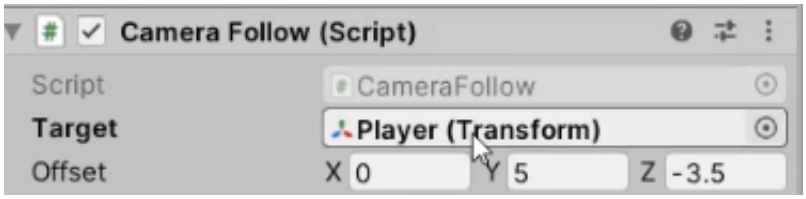
We're going to need to delete the existing Main Camera in the scene and replace it with our prefab camera.



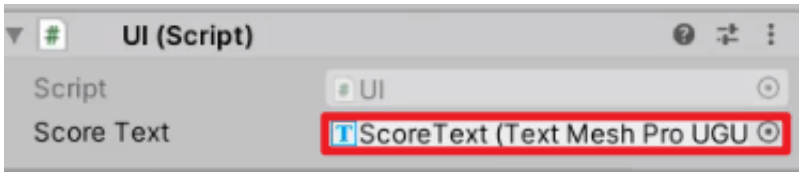
If you press Play now, you will see this error: Unassigned Reference Exception. This is because all of our object references are reset.

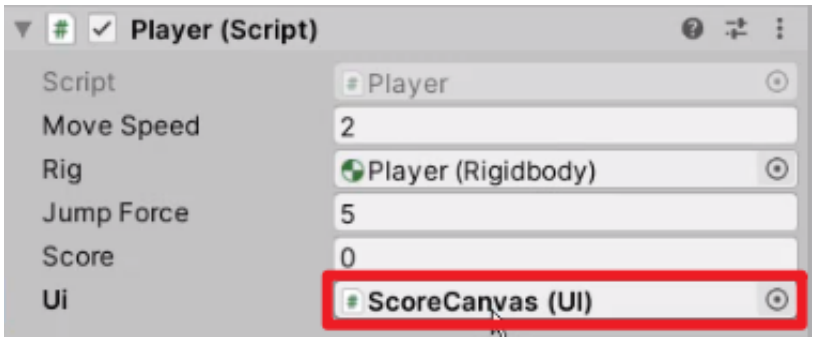


To fix this error, we're going to drag in our **Player** game object into the **Target** field of our prefab camera.



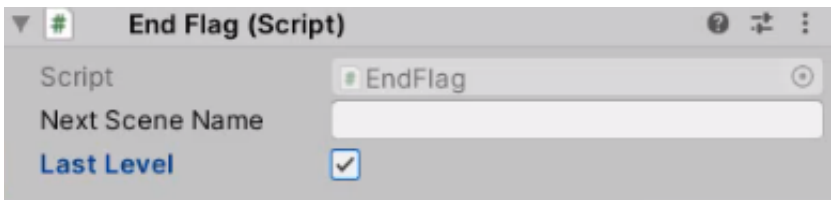
And for the same reason, we're going to set object references for the **ScoreText** variable and the **UI** variable.



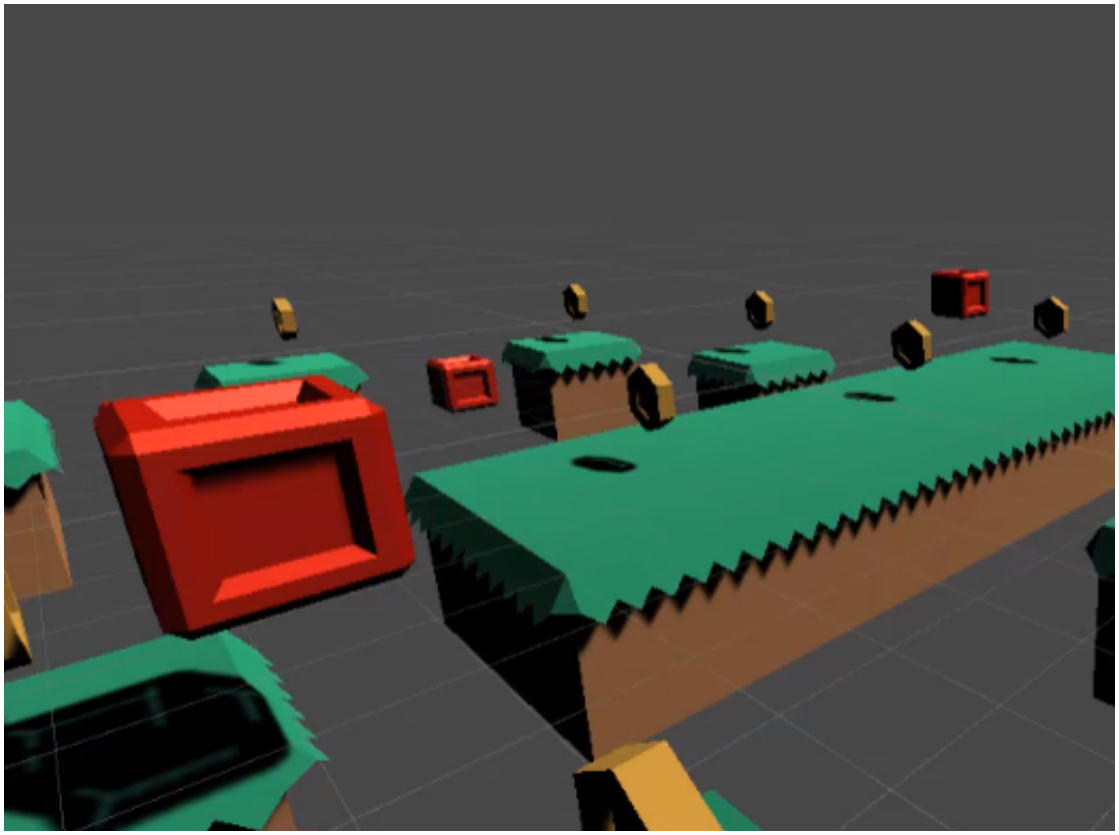


We also need to make sure we enter values for **Next Scene Name** and **Last Level**.

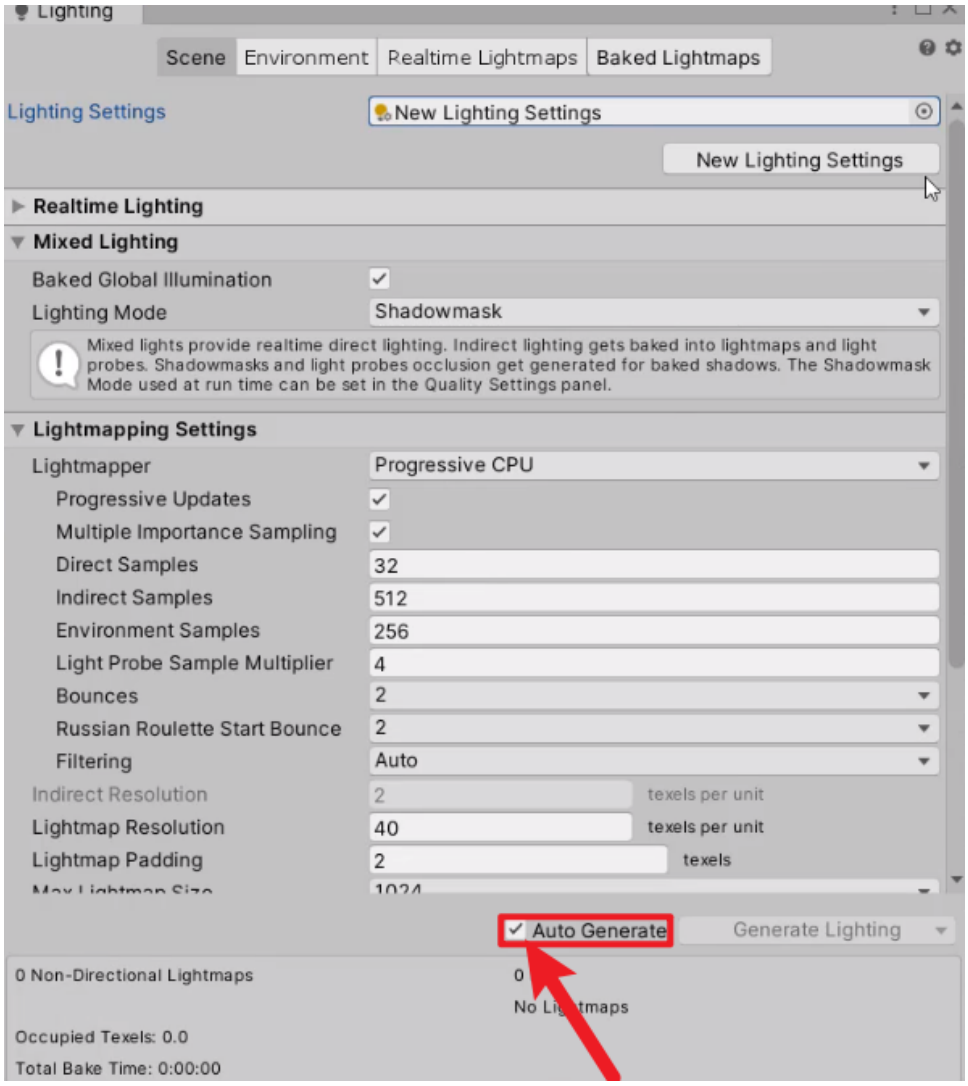
Since we only have two levels, we're going to leave the next scene name field empty and check the last level property.

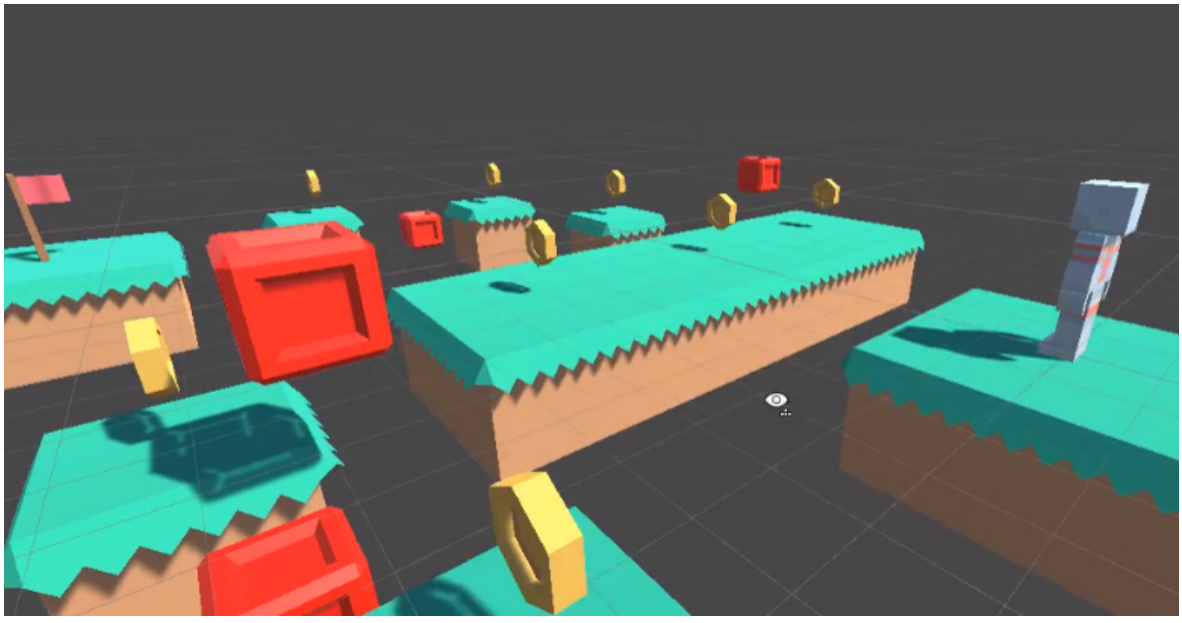


Again, if you encounter this lighting issue with your new scene, you can enable '**Auto Generate**' from Lighting Settings.



You can open up the **Lighting Settings** window by clicking this light icon on the bottom right corner of the screen.





Now when we reach the end flag of level 2, we will return to the menu.