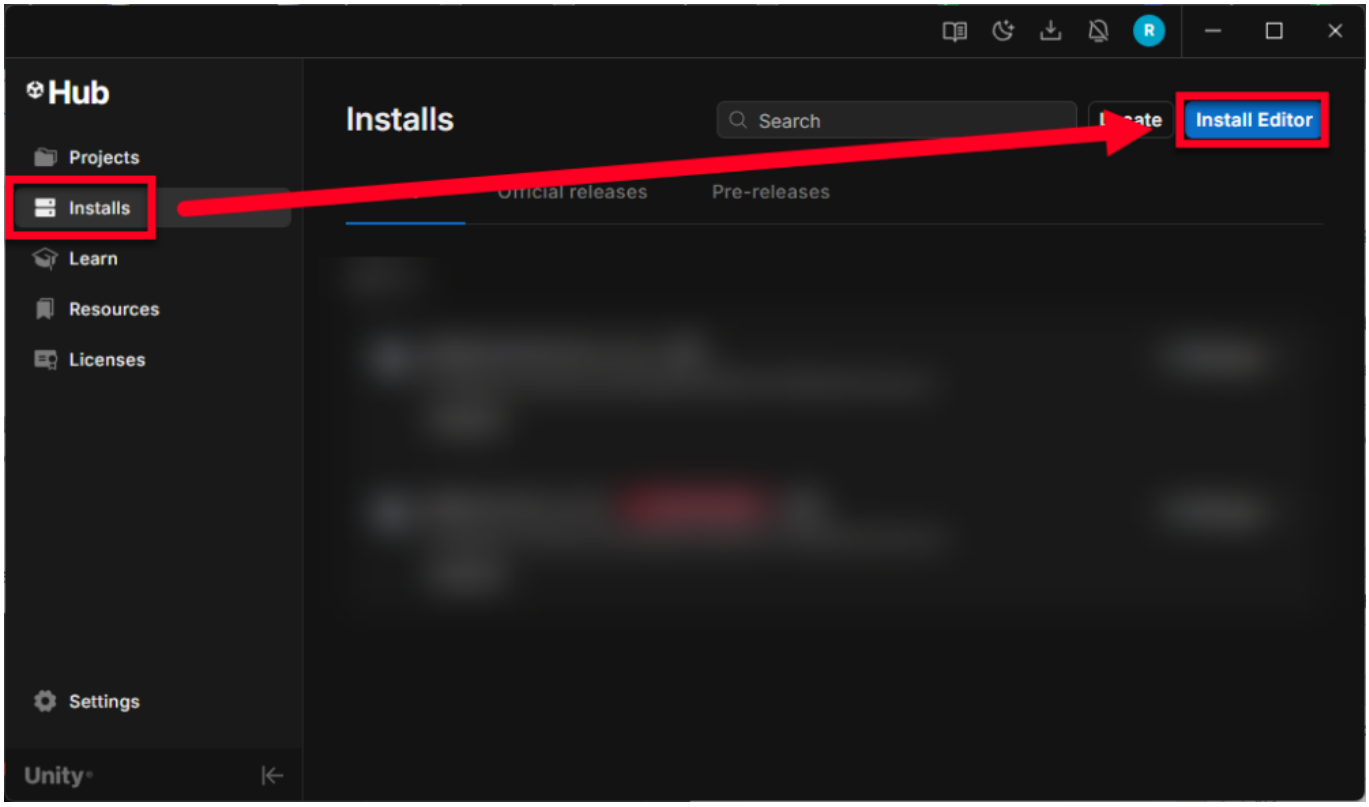
Course Updated to the Unity 6.3 LTS

We've updated the project files to Unity version **6.3 LTS** for this course. This is a long-term support version, which Unity recommends.

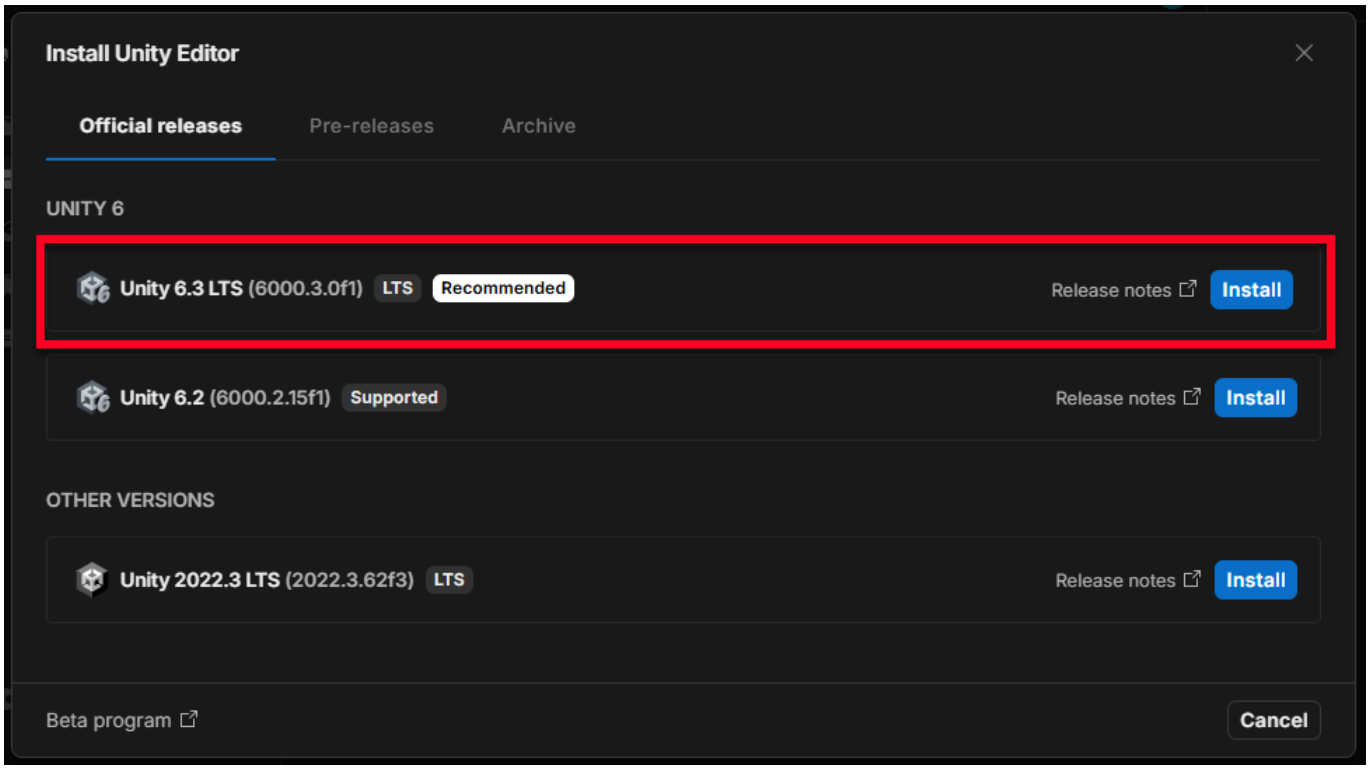
LTS versions are released each year and are a stable foundation for building your projects. For students, this means consistent project files across all courses – no matter when created. So, no more downloading different Unity versions or opening old projects filled with errors!

How to Install the Unity 6.3 LTS

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



A smaller window with a list of Unity versions will open. At the top of the **Official releases** list, in the 'Unity 6' section, we want to select **Unity 6.3 [LTS]** (Recommended version). From here, click **Install** and follow the installation process.

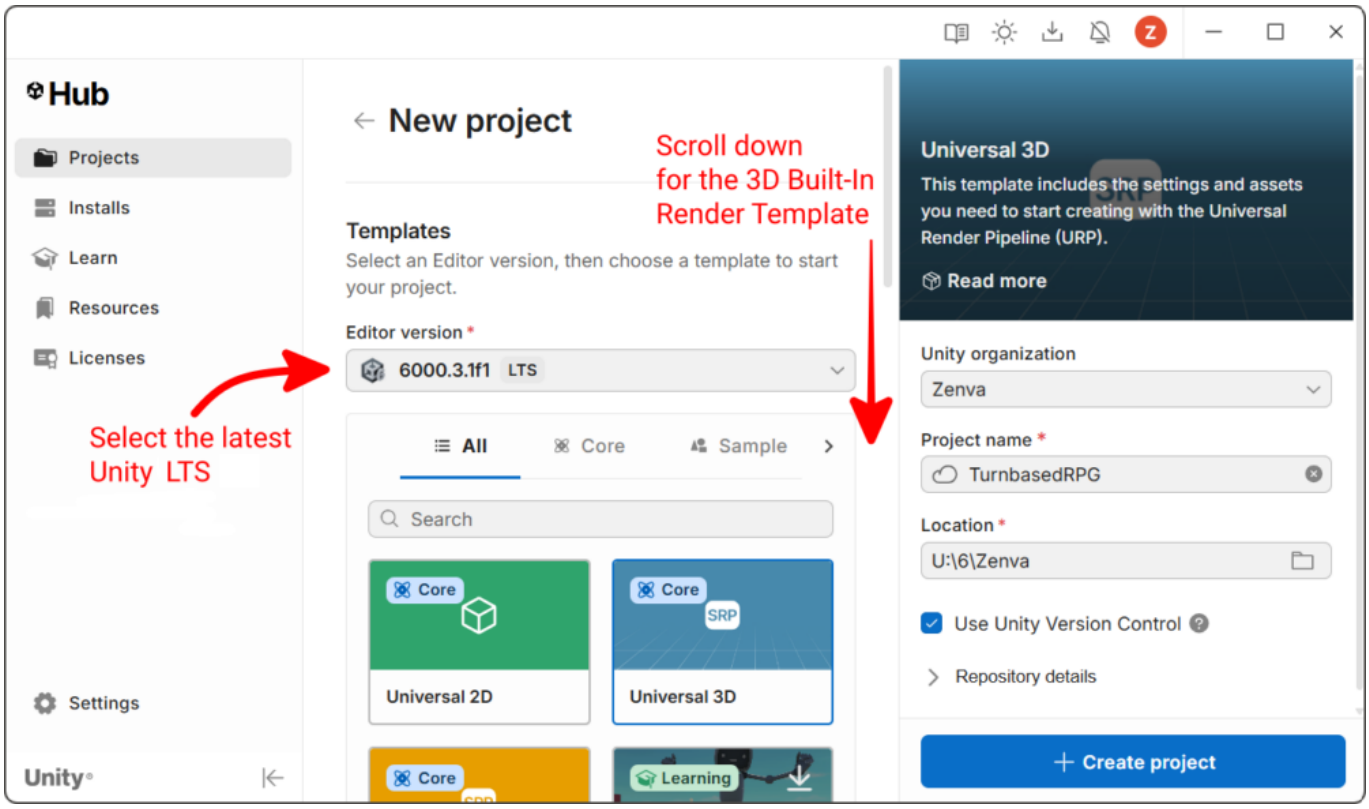


As of Unity 6.3 LTS, the new default render pipeline is set to Universal RP (URP). However, this course was created using an earlier version of Unity that used the Built-In render pipeline (now legacy). When creating a new project to follow along in the course, we'll therefore need to select the correct project template.

Use the following instructions to create a new project with the correct template:

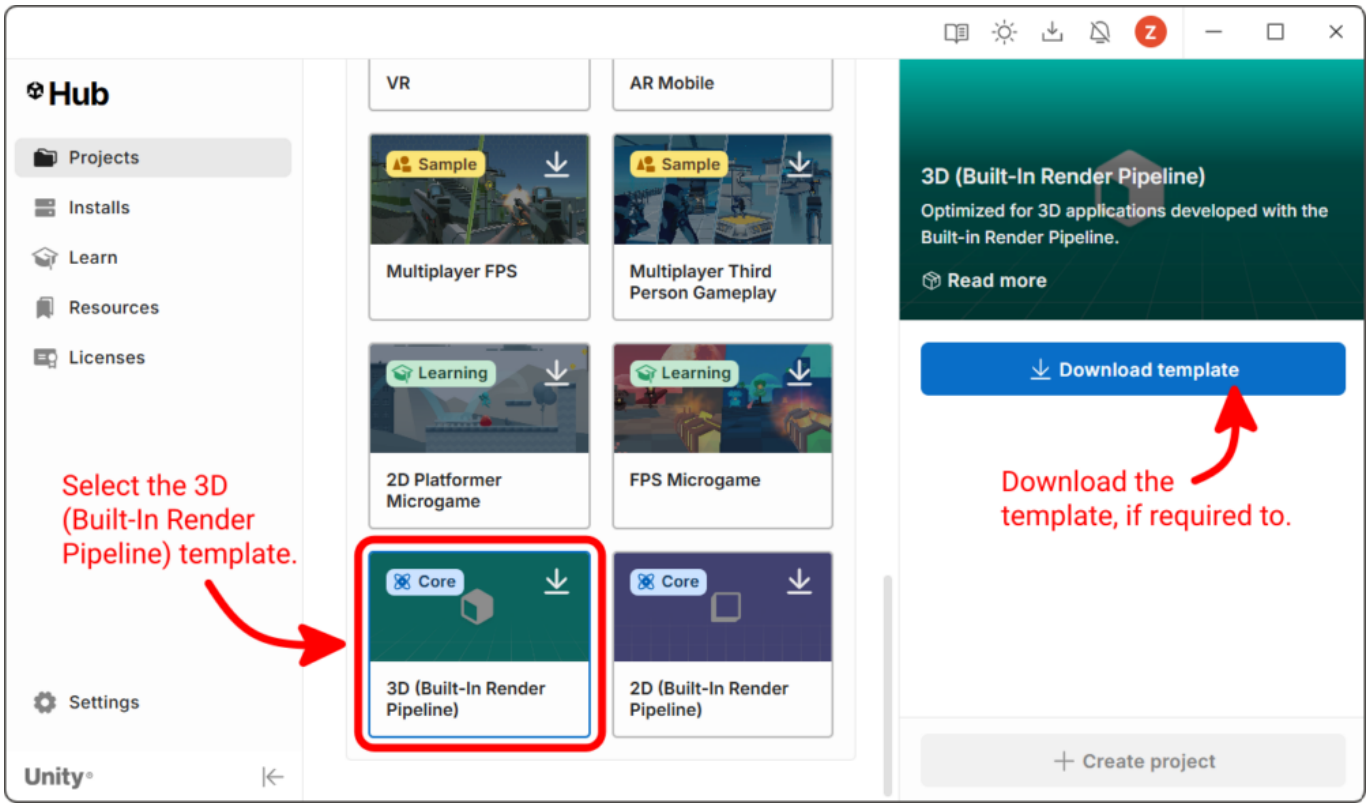
1) In Unity Hub, on the **Projects** screen (the default view when Hub opens), click the **+ New Project** button (top-right of the window).

2) After ensuring you've selected the latest LTS **Editor version**, scroll down the list of **Templates**.



3) Near, or at the bottom, you'll find the **3D (Built-In Render Pipeline)** template.

4) Select the template and click the **Download template** button if required to download it.



You can now continue to fill out the required fields for **Project name** and **Location**. Then, click **Create project** to get started!

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

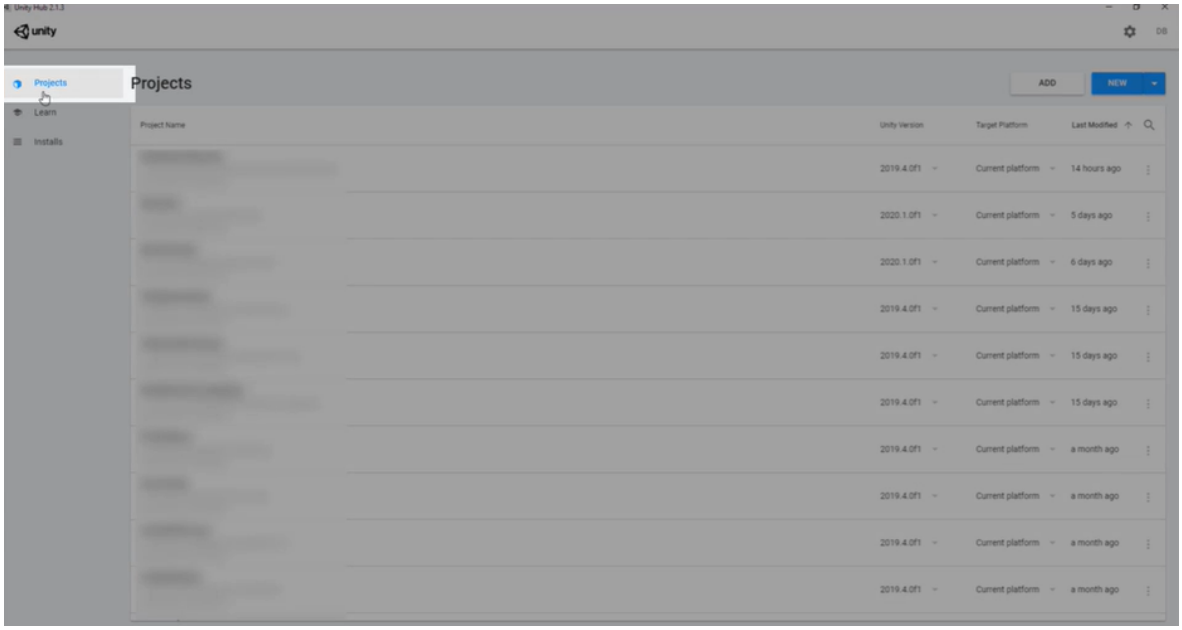
In this lesson, we will be setting up a Survival Game in Unity.

Creating Unity Project

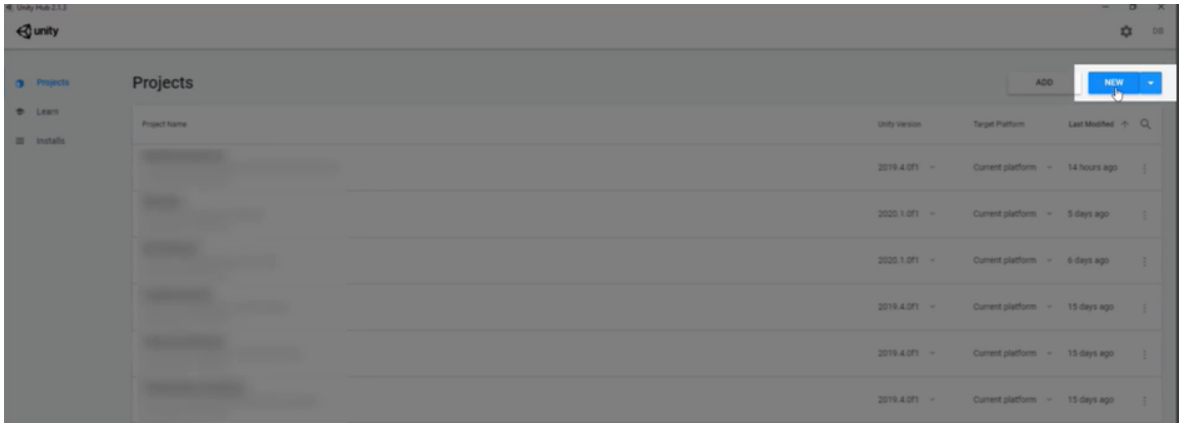
First of all, let's open up a new empty 3D project in Unity 2020.1.0f1 or higher.

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

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

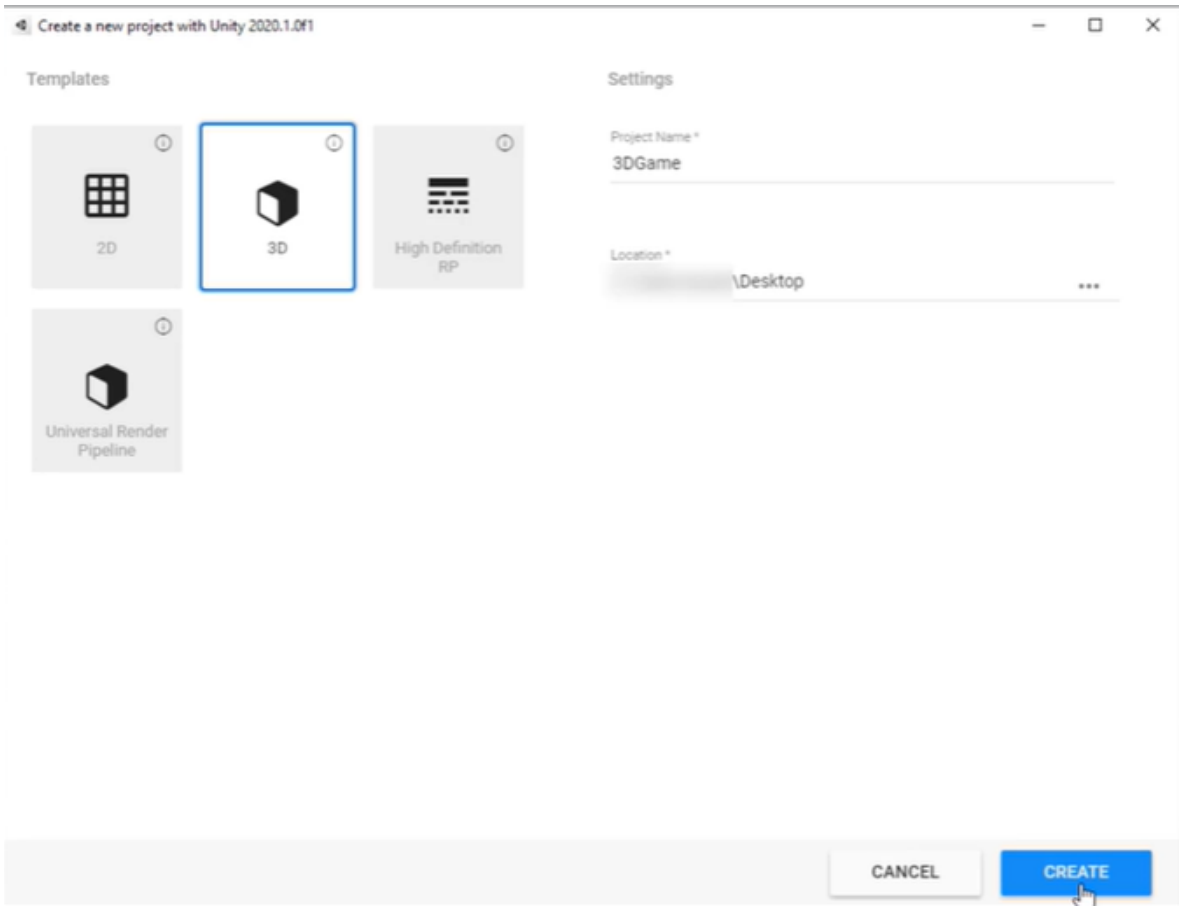


... and click on the **New** button**:**

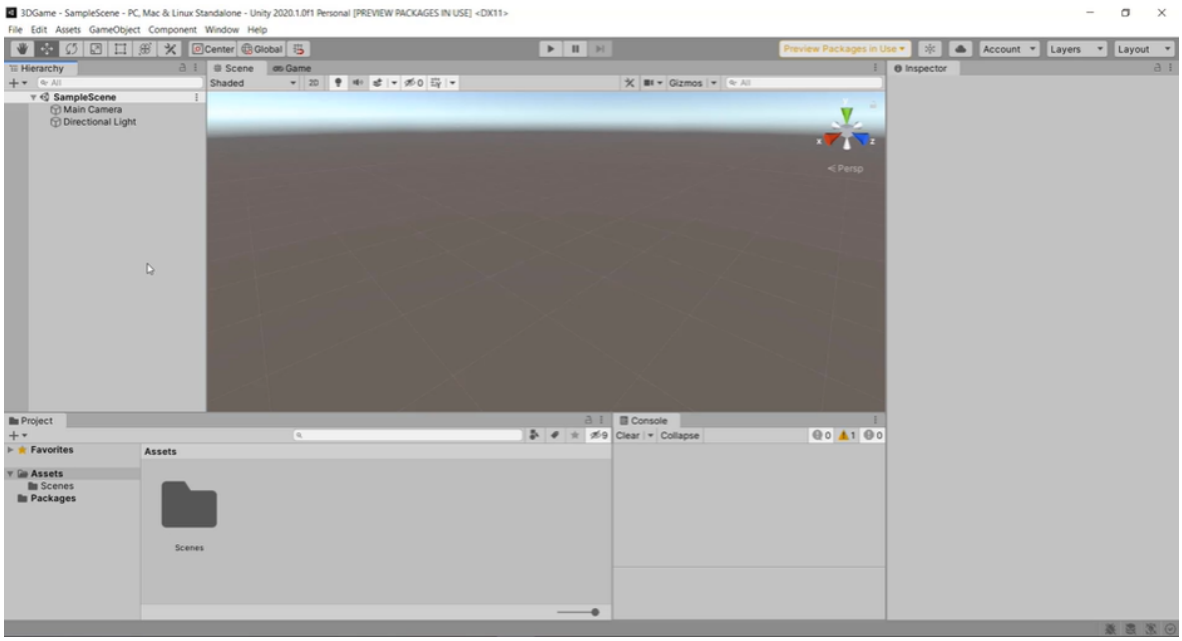


Select the **3D template**, and set the project's name and location. Then click on the '**Create**' button.

The following instructions have been updated, and differ from the video: please ensure you select the **"3D (Built-In Render Pipeline)"** template because we will be using the **Built-in Renderer** specifically for this project. *Note that selecting a template other than this will result in not being able to complete making the survival game.*



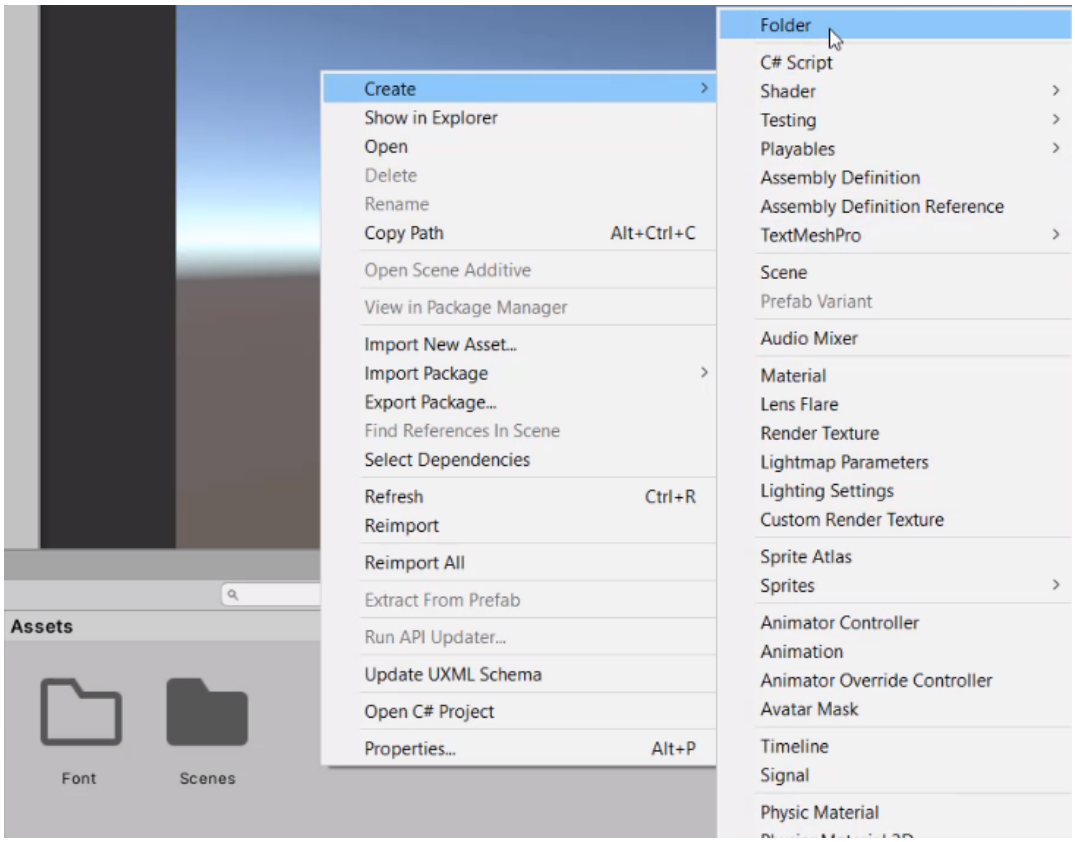
We now have a basic 3D scene open inside of our brand new project.



Importing Resources

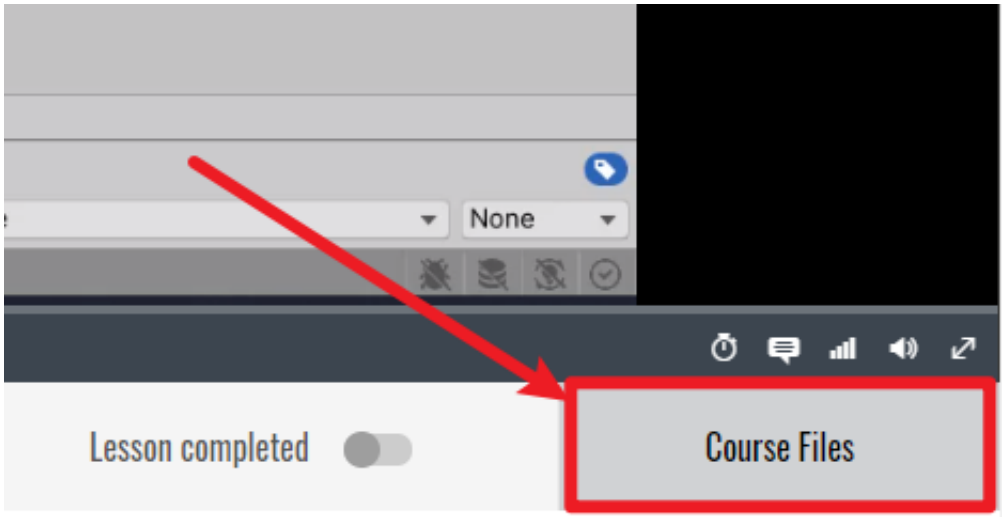
First of all, we're going to **create new folders** to store our assets.

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

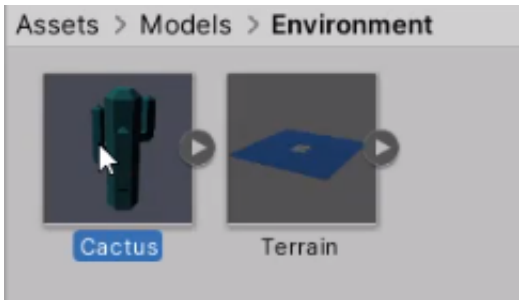


We'll create folders for "**Prefabs**", **"Materials", "Sprites", "Textures",** "**Scripts**" and **"Other"**.

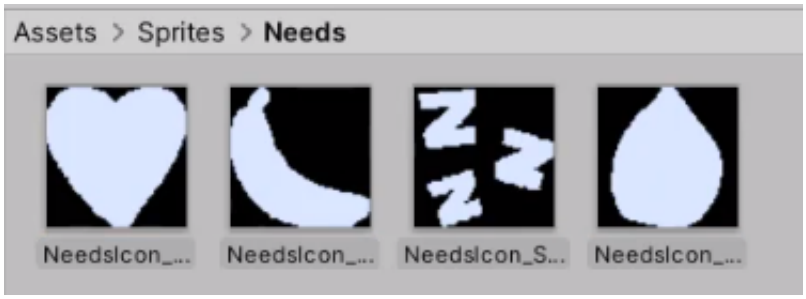
Next, we're going to download the **Assets.zip** file from the **Course Files** tab.

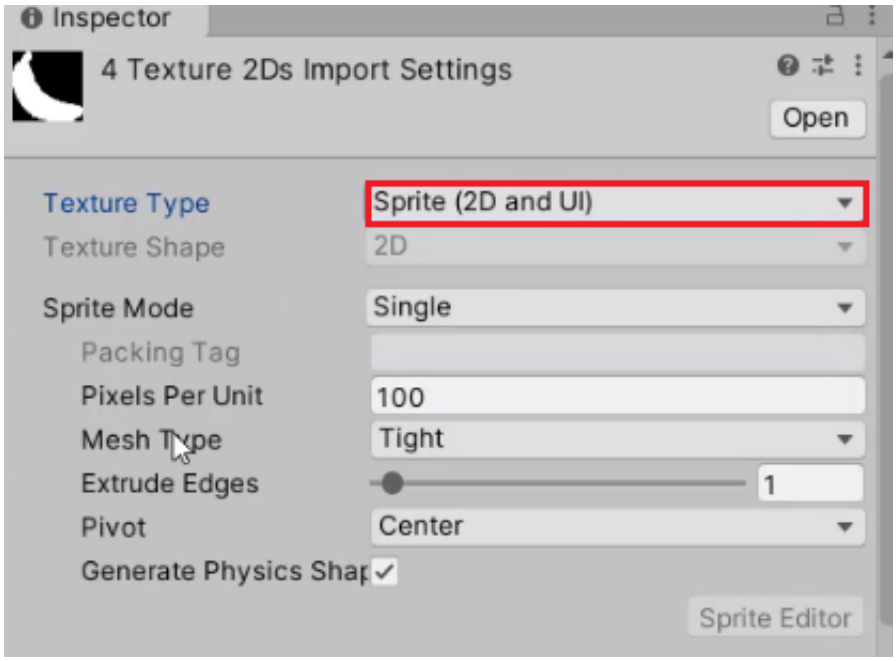


Then we're going to select the **Models** folder inside the extracted .zip file and drag them into the **Models** folder in Unity Editor. We can do the same for our **Sprites** and **Textures**.

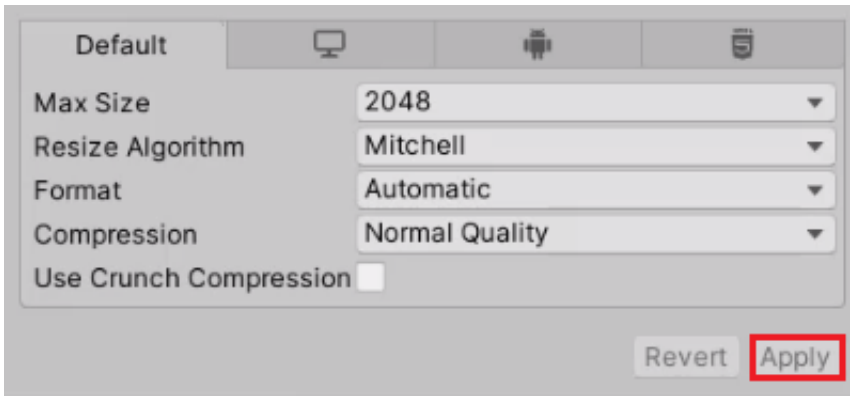


Select all the **Sprites**, and set the **Texture Type** to **Sprite (2D and UI)** in the Inspector.





Make sure to scroll down and click on the **Apply** button to save the changes.

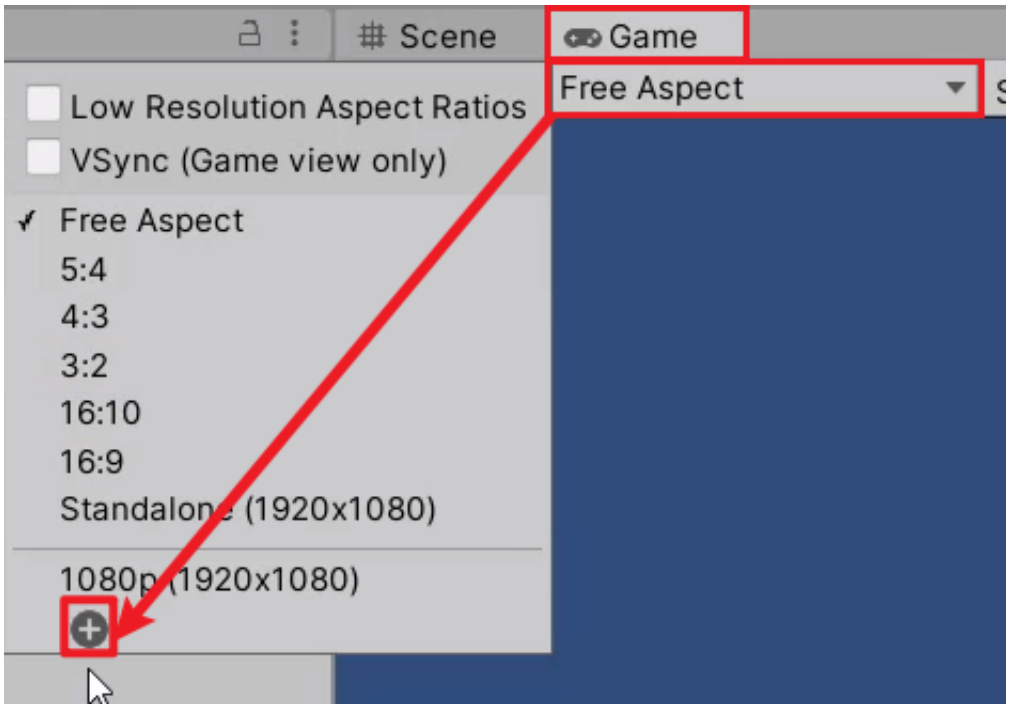


Now we've successfully imported all the assets into our project.

Screen Resolution

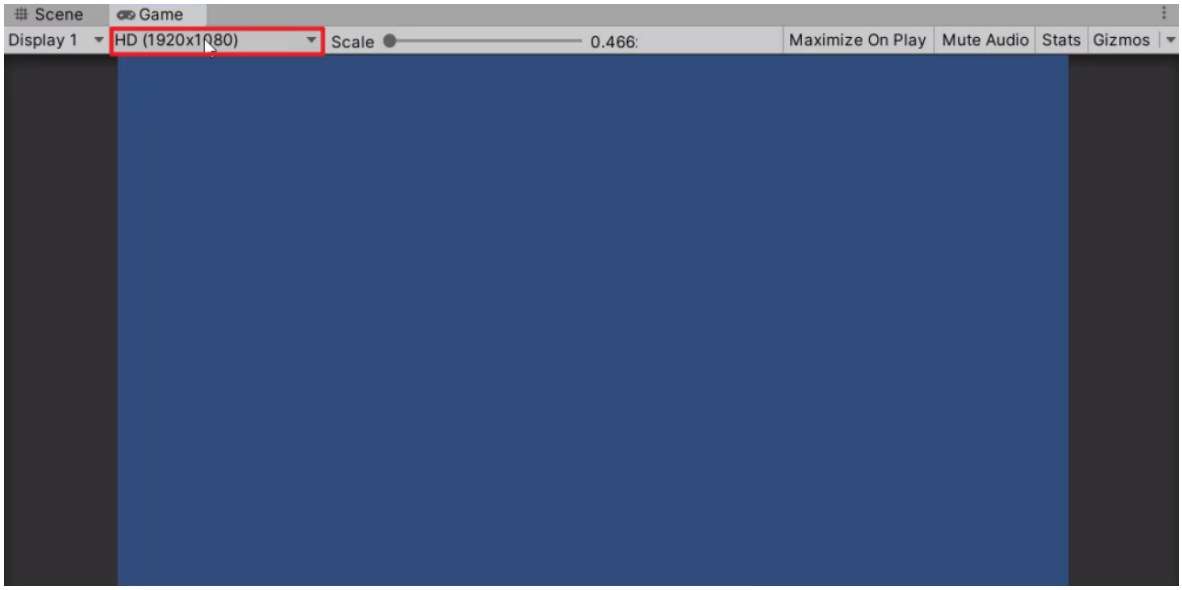
When you click on the **Game** view and go to **Free Aspect**, you can change the **aspect ratio** of your game preview window.

Let's create a new **HD** resolution (**1920 x 1080**) template by clicking on the **plus** icon.



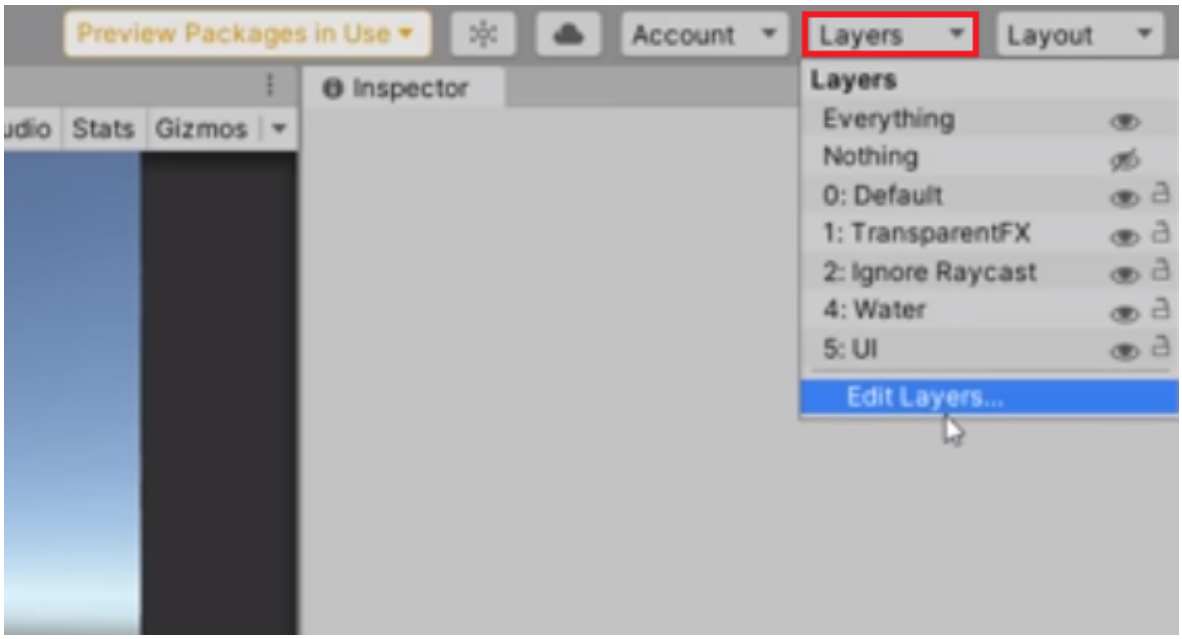
We can then hit **OK** to save and apply the new resolution settings.

Instead of being the size of the game window (**Free Aspect**), our screen is now **fixed** into the resolution of **1920** by **1080**.



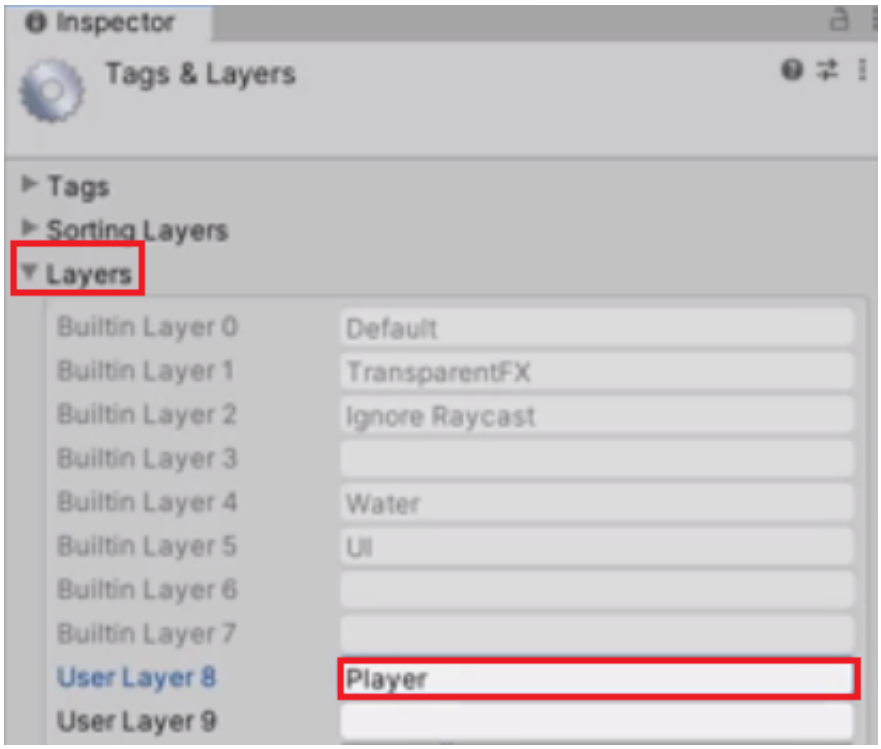
Setting Up Layers

At the top right corner of the screen is a **Layers** dropdown. Click on **Layers > Edit Layers...**



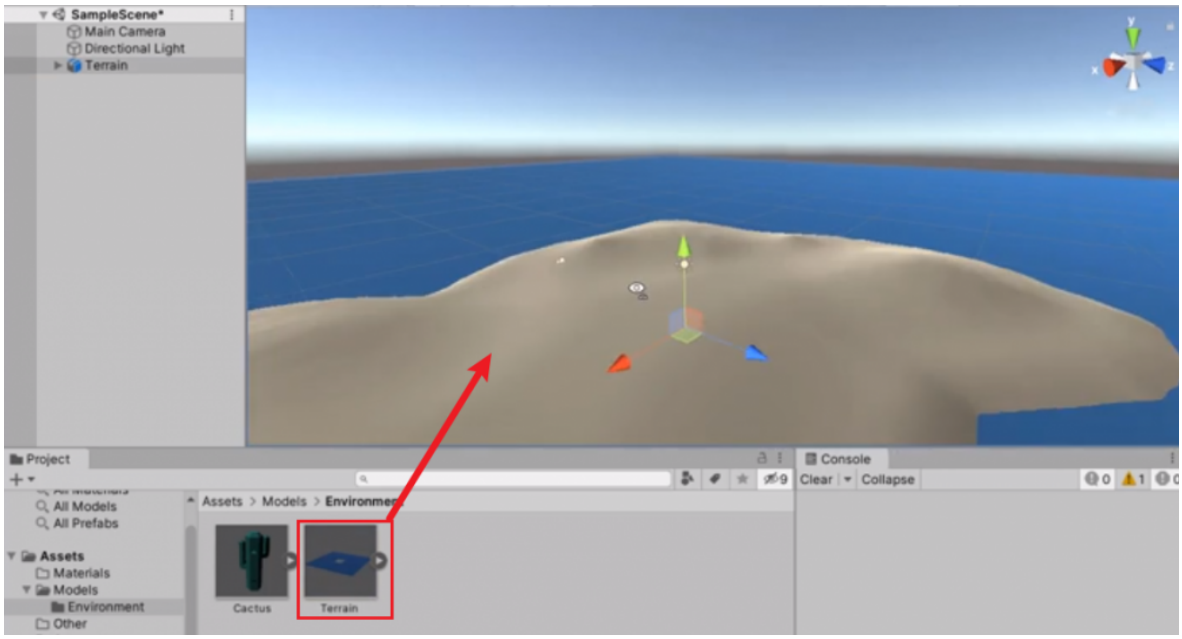
We can then expand the **Layers** list and create a new layer called "**Player**".

When we are detecting to see if the player is standing on the ground, we're going to be shooting a **Raycast** down below the player's feet. But the ray shouldn't detect the player collider itself, so we will be using the player layer to ignore the ray's collision with the player.

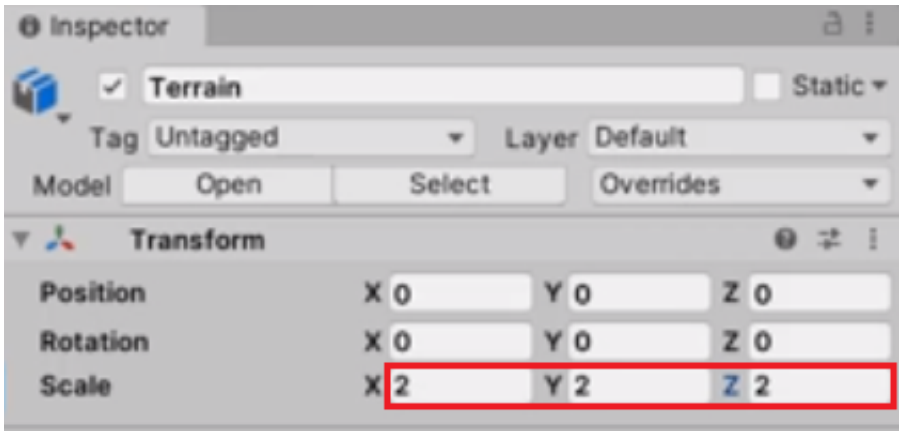


The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected version.

To set up our environment, we're going to select the **Terrain** model in the Environment > Models folder and drag it into the scene.

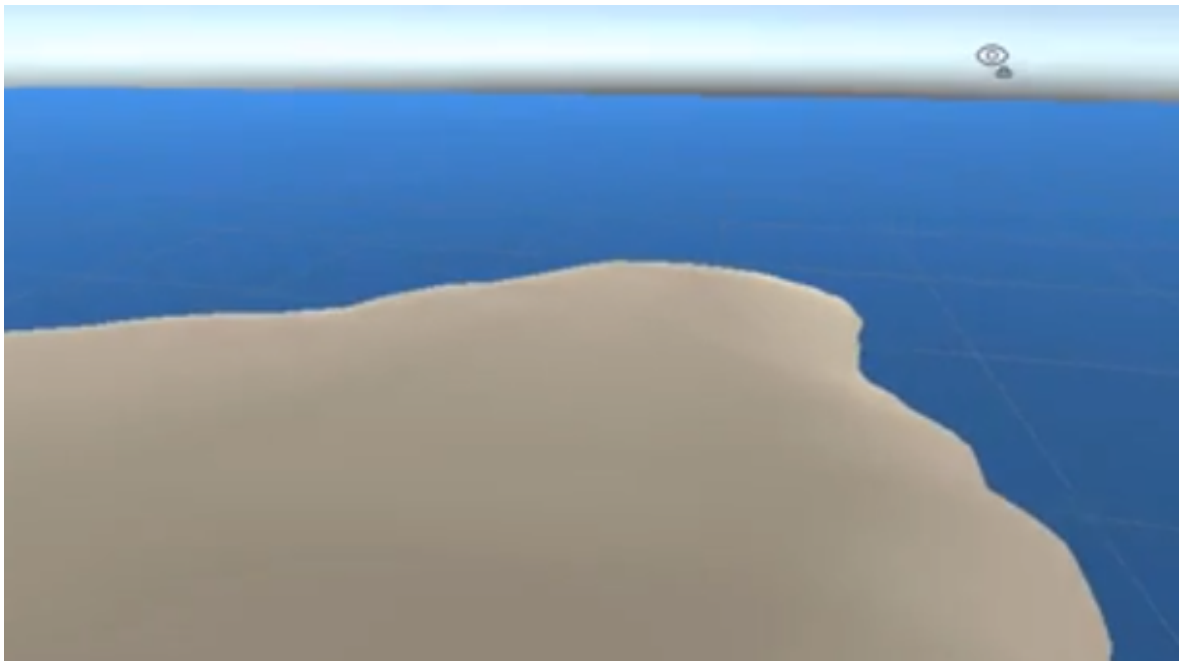


We'll increase the **Scale** of the terrain to **(2, 2, 2).**

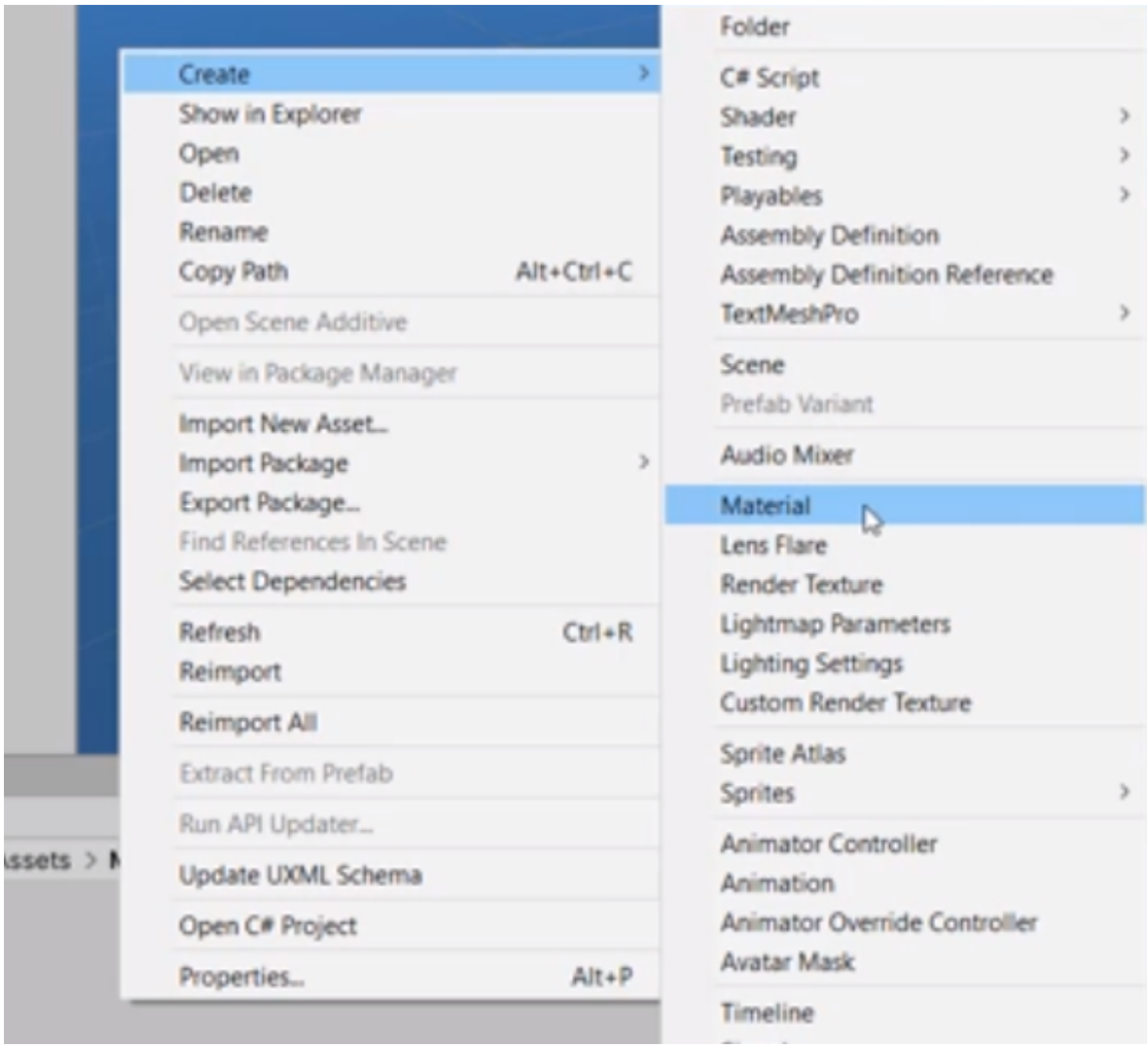


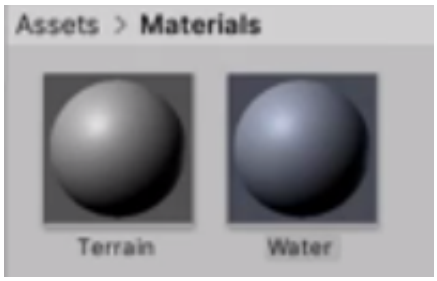
Setting Up Materials

As you can see in the image below, the water texture of the terrain is not reflective at all.

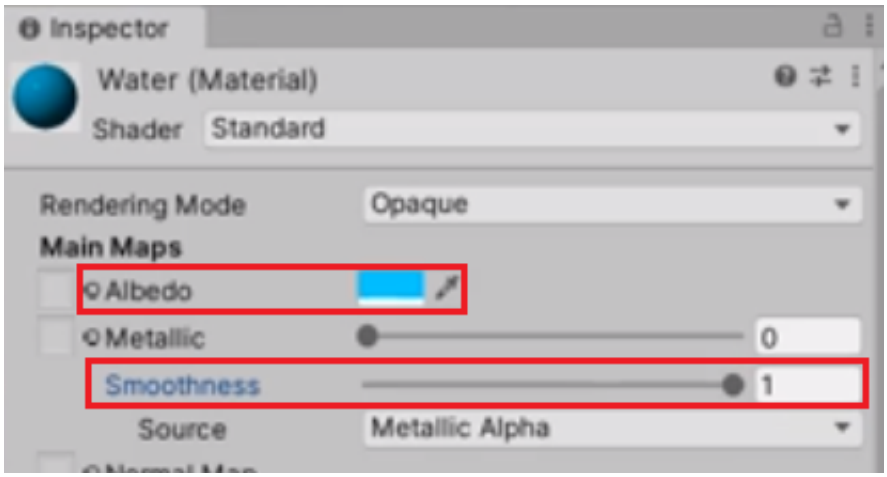


To fix this, navigate to the **Materials** folder, and **Right-click > Create > Material** to create two new materials. Name them '**Terrain**' and '**Water**' respectively.

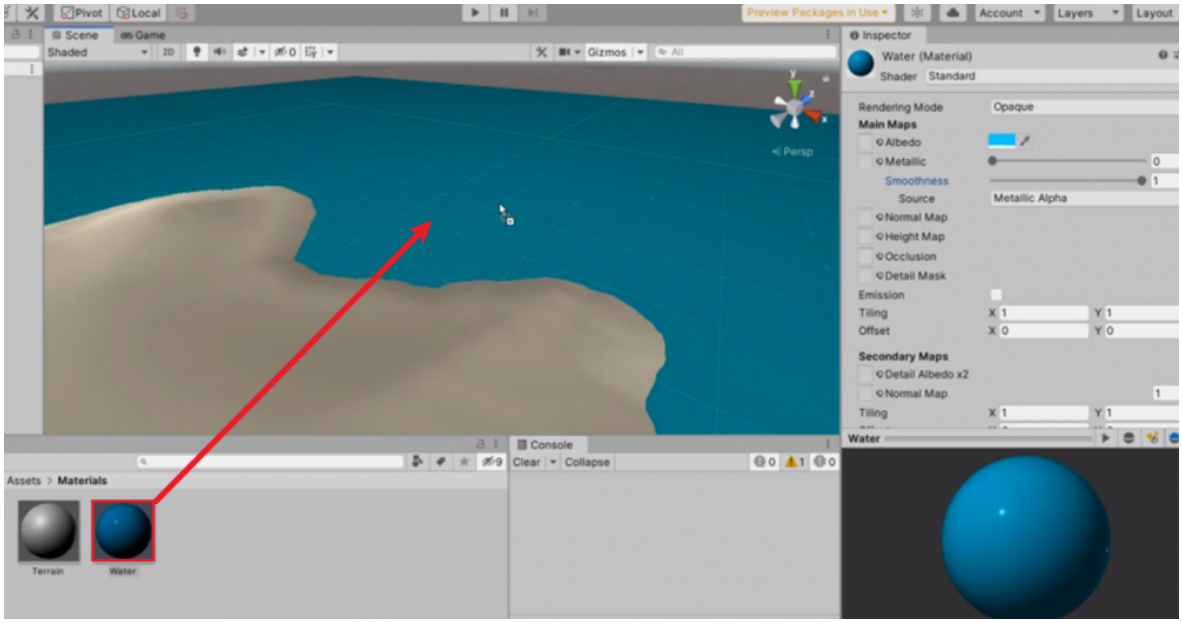




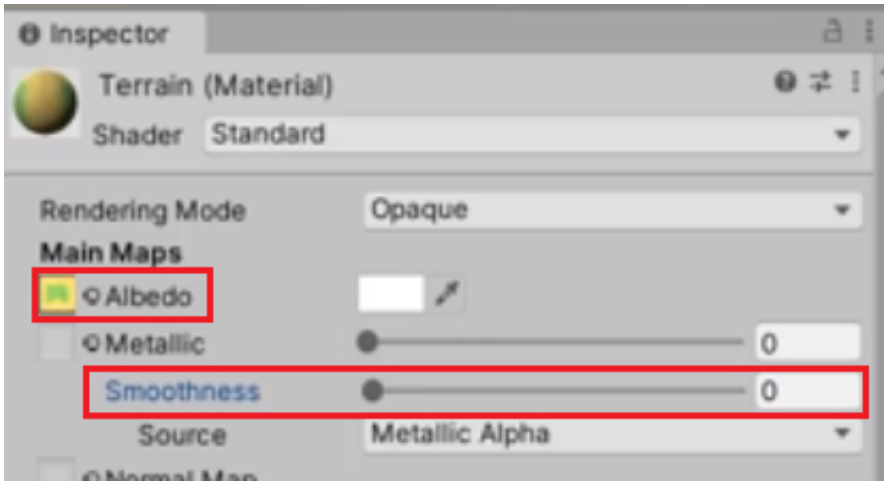
In the inspector, set the **Albedo** color to be blue, and set the **Smoothness** to 1. The smoothness defines how reflective the material looks.



Drag the material onto the water to apply the material.

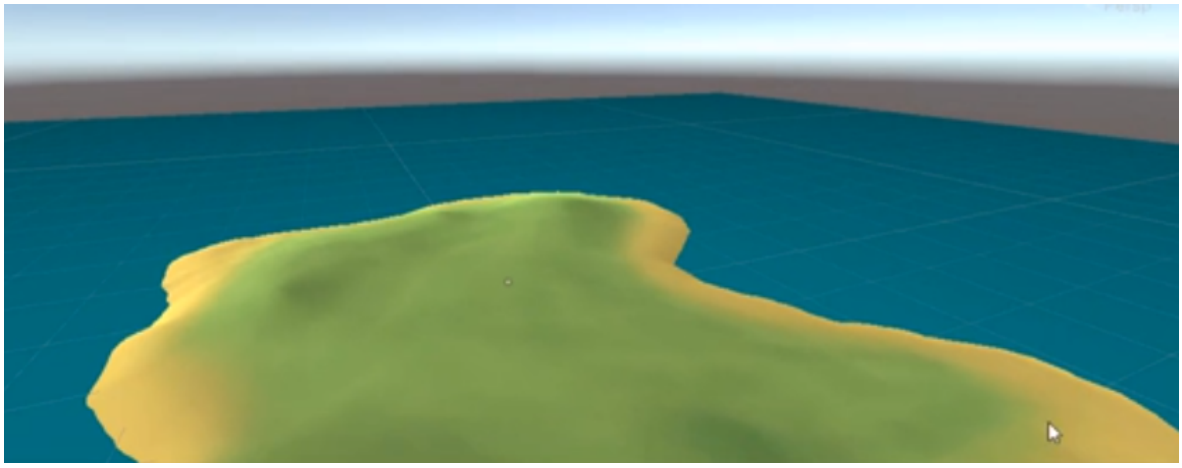


Select the **Terrain** material, and set the **Albedo** to a **Texture** called “Terrain”. Set the **Smoothness** to 0, as the island does not need to appear reflective.



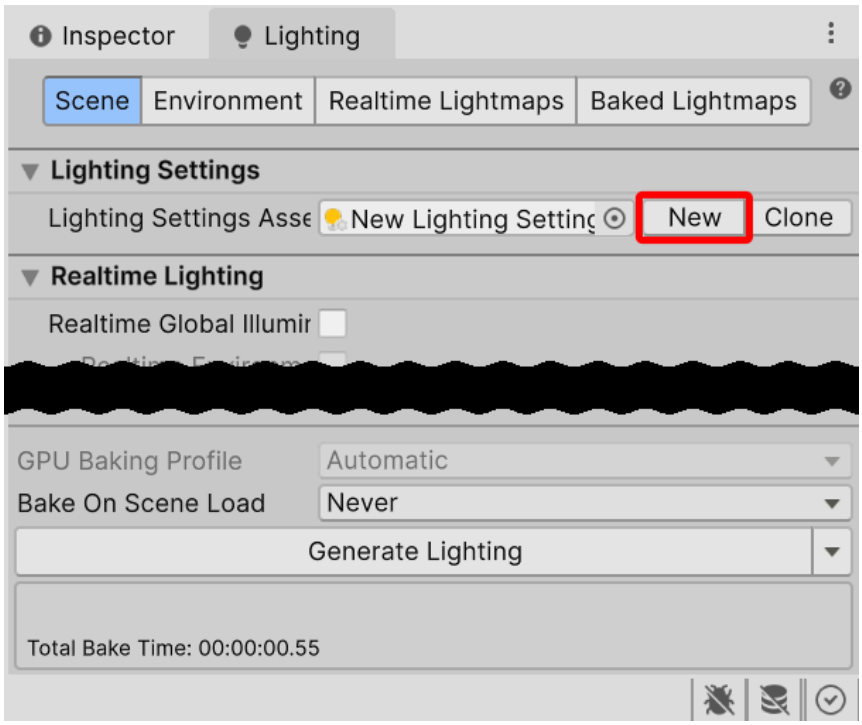
Auto Generate lighting

You may notice that the scene is quite dark, despite having a Directional Light object in the scene. This is because the **Auto Generate Lighting** is set to off by default.



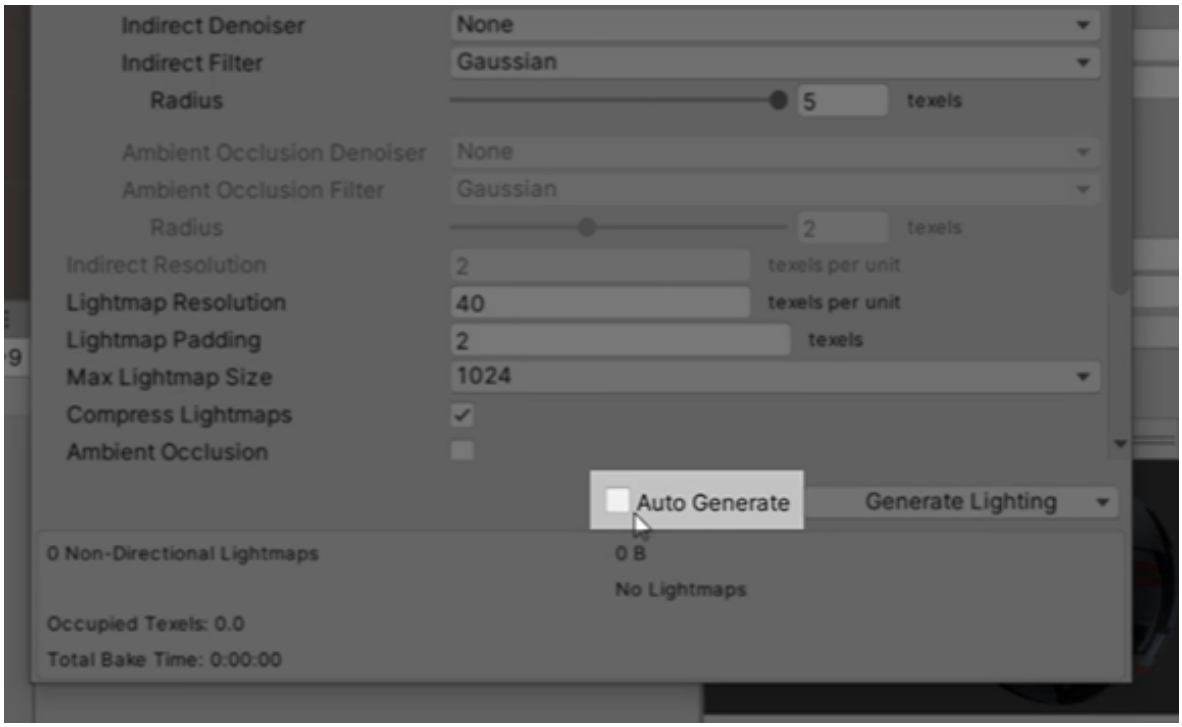
The following instructions have been updated, and differ from the video:

In newer versions of Unity, the lighting settings have changed. We now have to create a new **Lighting Settings Asset** before we can change the lighting settings. Click the **New** button at the top of the panel. You can now click **Generate Lighting** and, optionally, change the **Bake On Scene Load** setting to your preference.

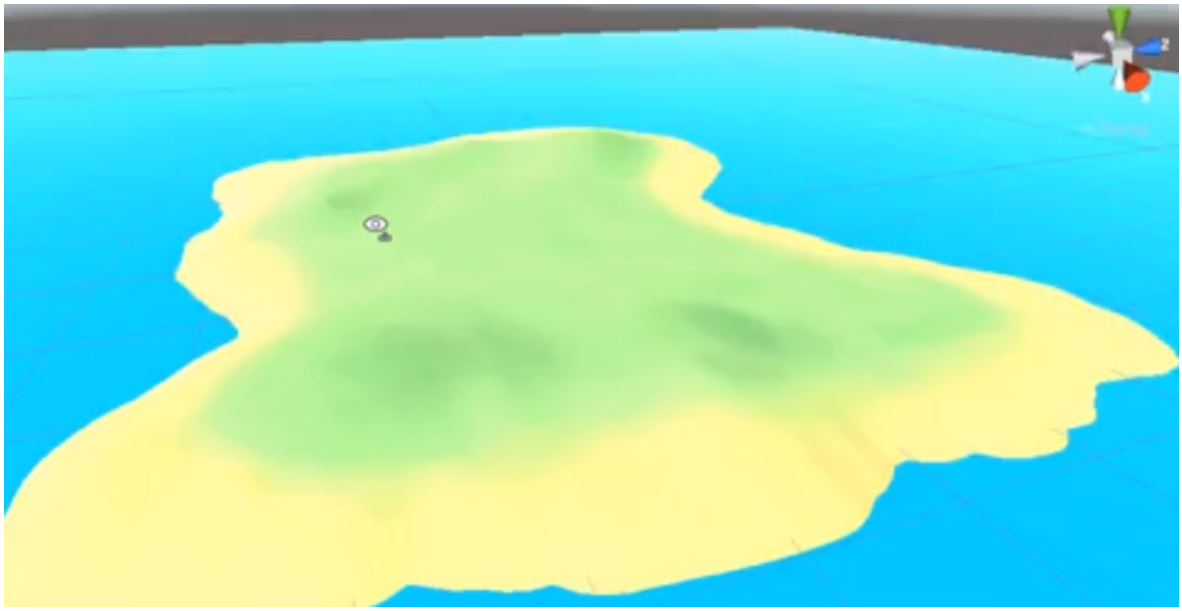


If you're on an older version of Unity, continue with these steps; otherwise, you can skip down to the **Setting Up Skybox** section below.

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

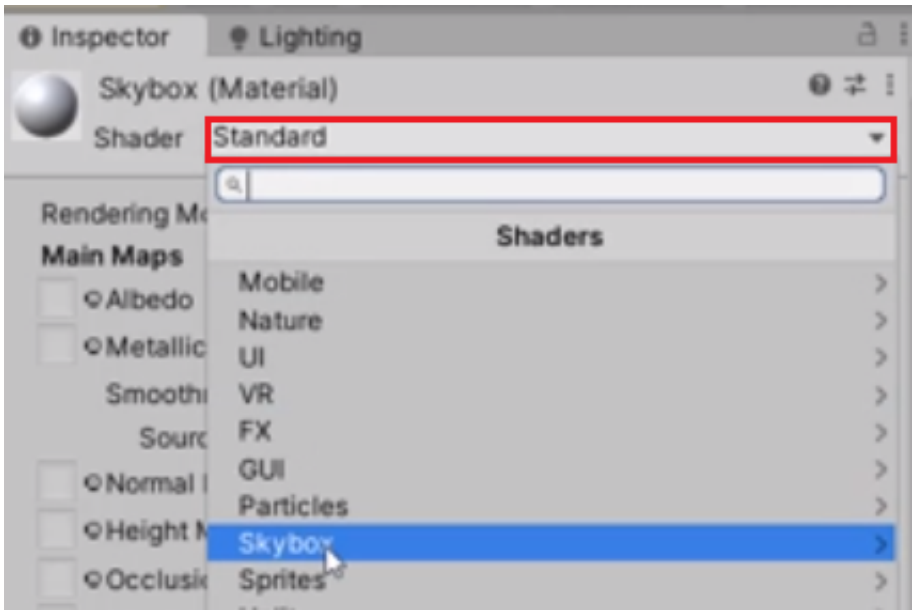


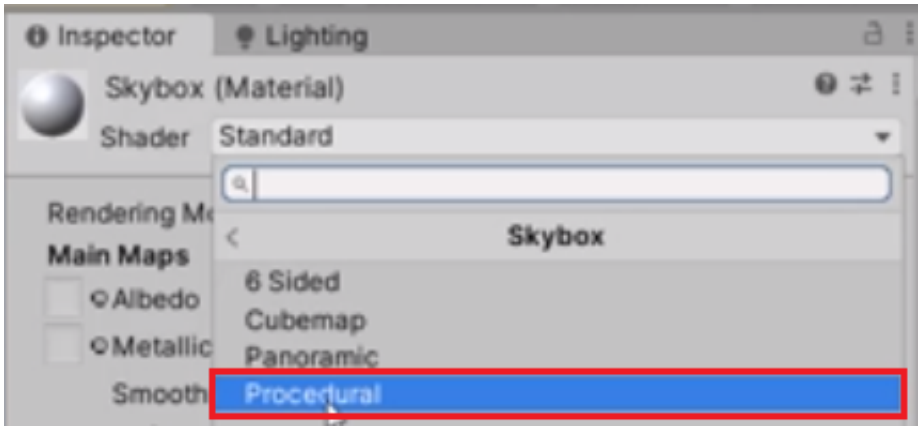
You can see that our terrain is now being lit correctly.



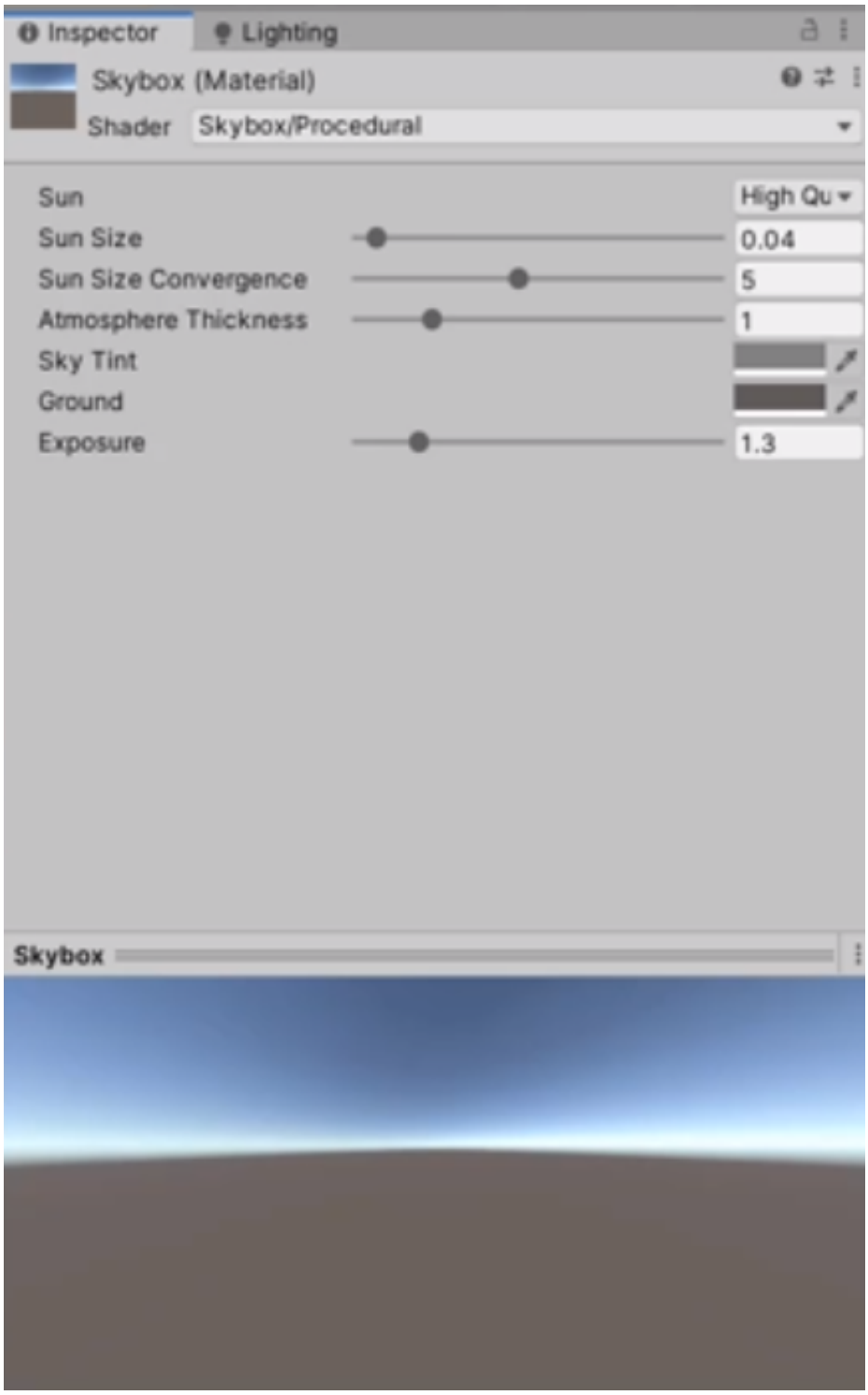
Setting Up Skybox

Now we're going to create a new material (**Right-click > Create > Material**) named "Skybox", and change its shader to **Skybox/Procedural** in the Inspector.

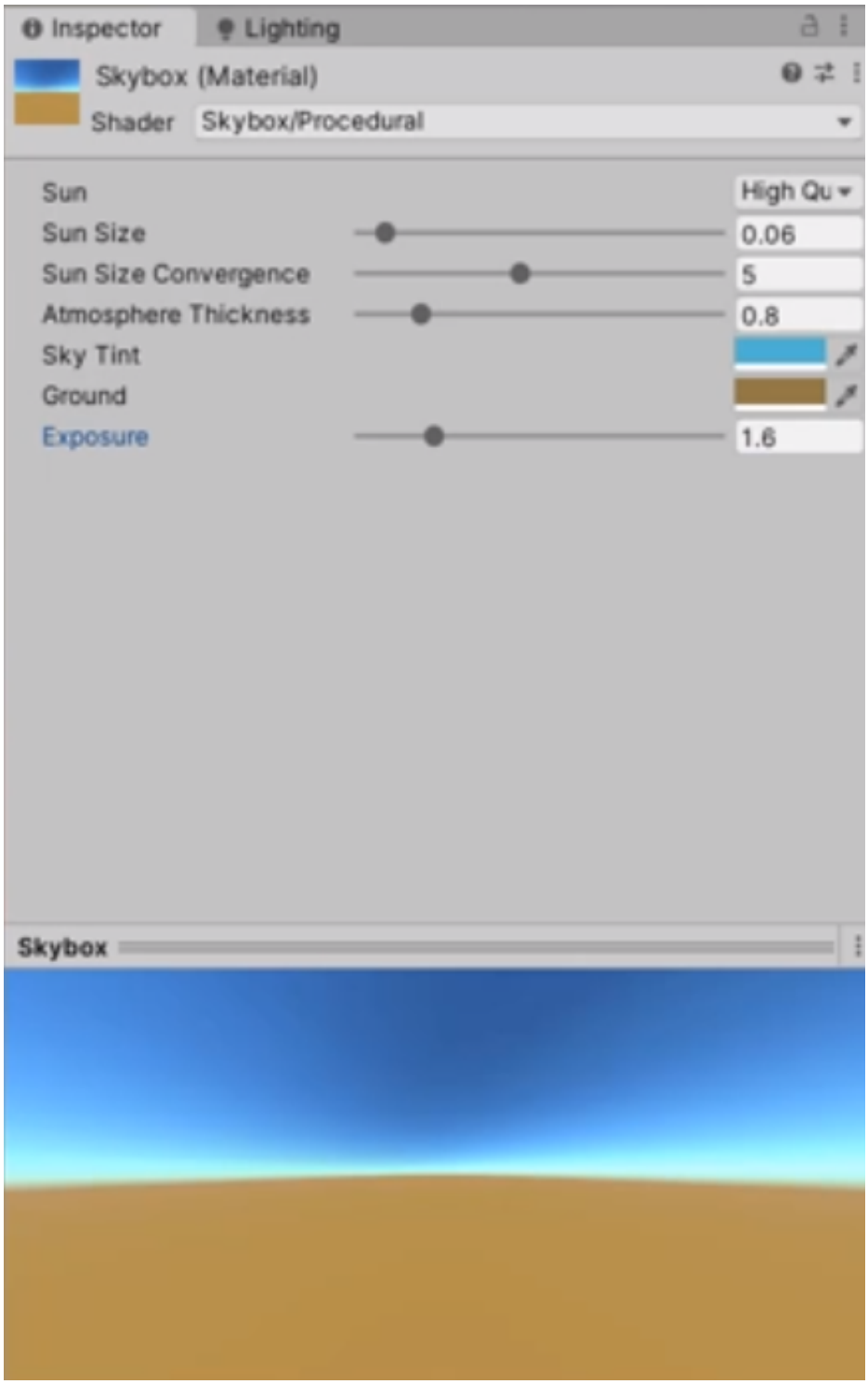




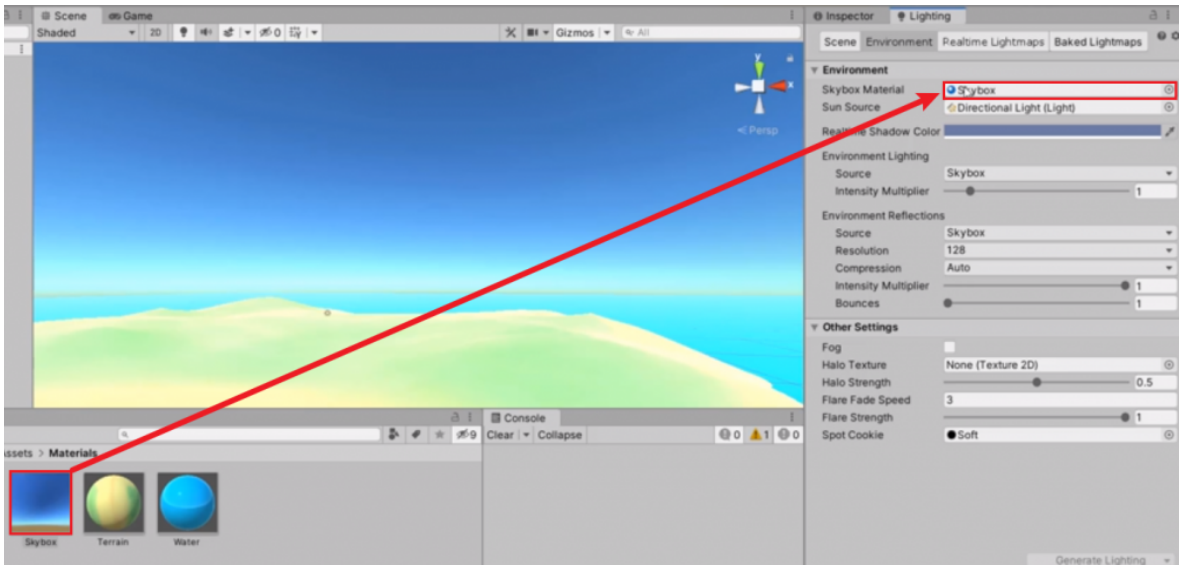
This gives us a new skybox material that is identical to the default skybox, except that it can be customized in the Inspector.



Once you have modified the skybox to however you wish, you can apply it to the scene by going over to the **Lighting Window** again.



Click on the **Environment** tab, and drag the skybox into the **Skybox Material** property.

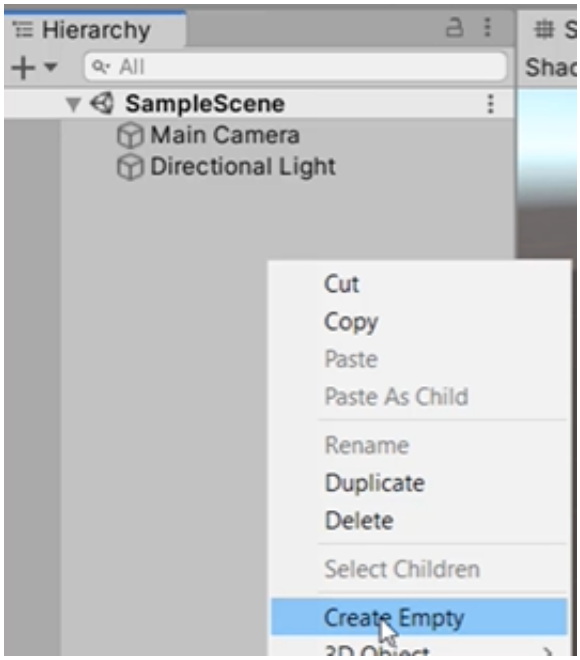


In this lesson, we're going to be creating a Player **GameObject*.**

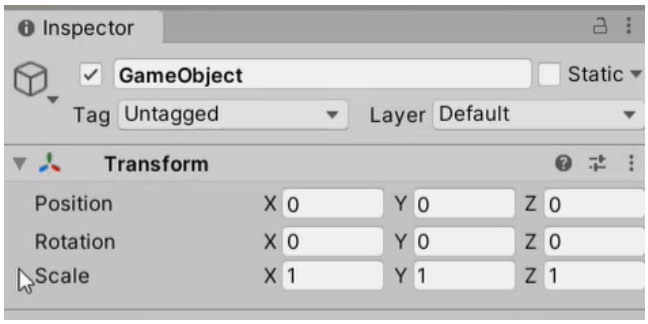
*(A **GameObject** is the base class for all entities in Unity that you can attach components onto.)

Creating A GameObject

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

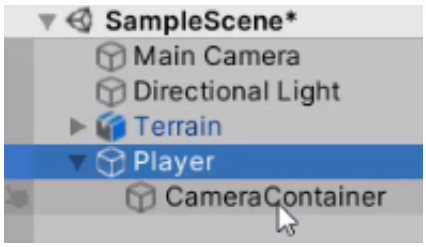


This is going to create a brand new GameObject. In the **Inspector** window, you can see that it has a component called **Transform**.



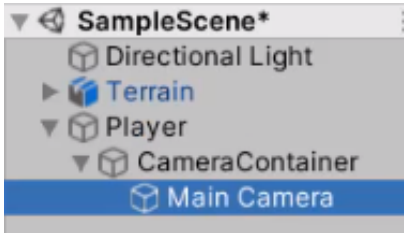
This component defines the **position, rotation,** and **scale** of our GameObject in 3D space.

Let's **rename** the object as "Player", and create another game object as a **child**.



This child object is going to contain our **Main Camera,** and be placed at the location of the player's

eyes. (Y-position of 1.7)



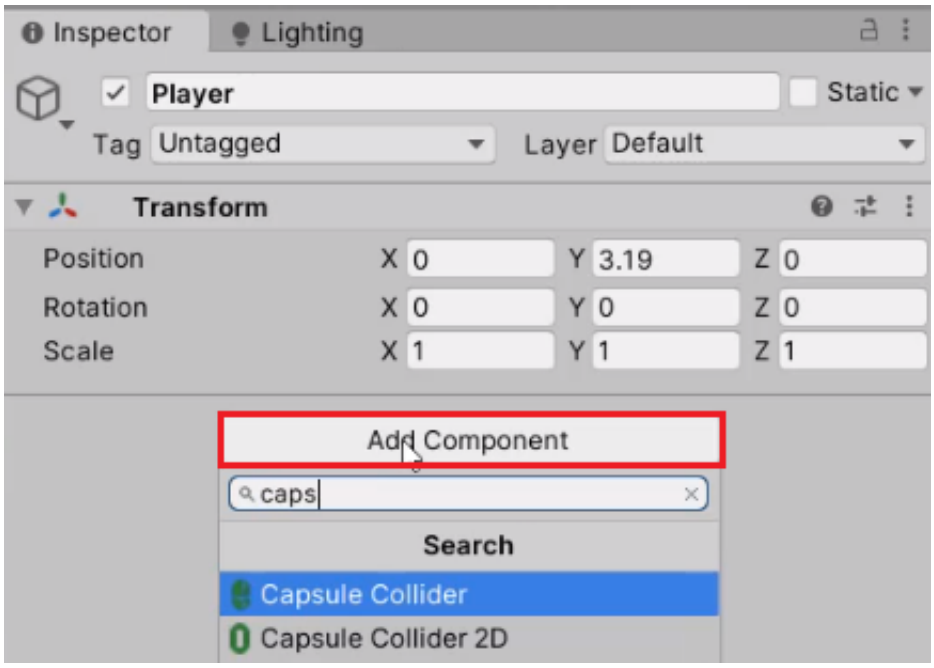
We will now add the following components to our Player GameObject:

- Capsule Collider
- Rigidbody
- A script that allows us to move, jump and collect items.

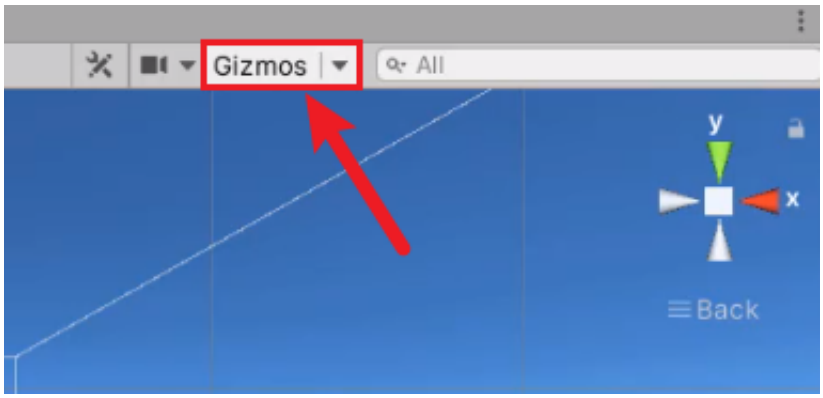
Adding A Collider

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

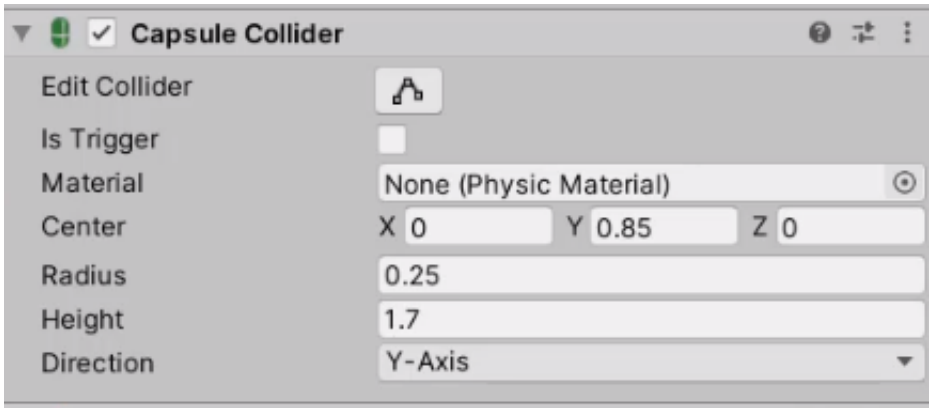
To add a capsule-shaped collider, go to the Inspector and click on **Add Component** > (search) **Capsule Collider**.



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 **0.25**, the **height** to **1.7**, and the **y-center** value to **0.85**, so it matches the shape of our Player model.



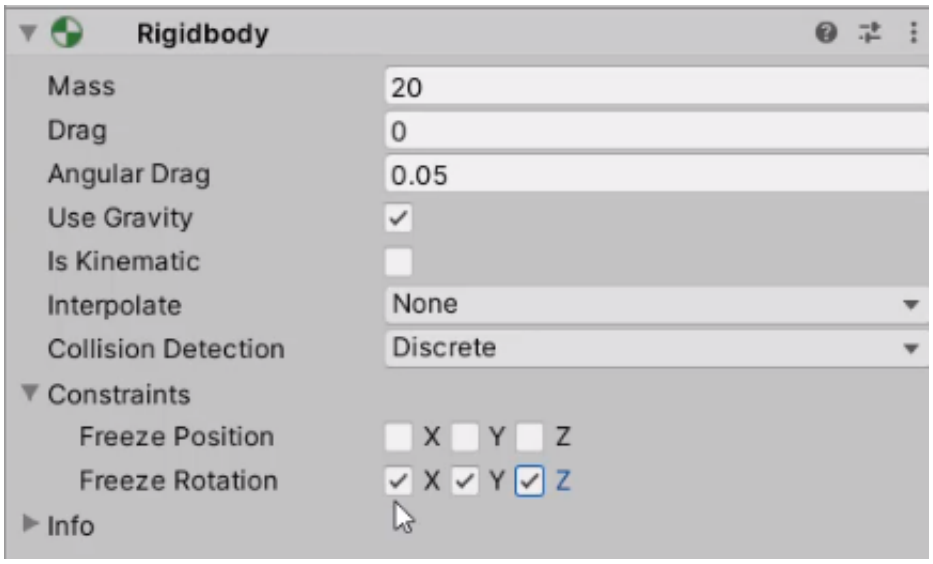
Adding Rigidbody

In order for the collider to work properly, we need a **Rigidbody** component. Rigidbody is basically a component that applies physics such as gravity, drag, mass, etc.

To add a Rigidbody component, click on **Add Component >** (search) **Rigidbody**.

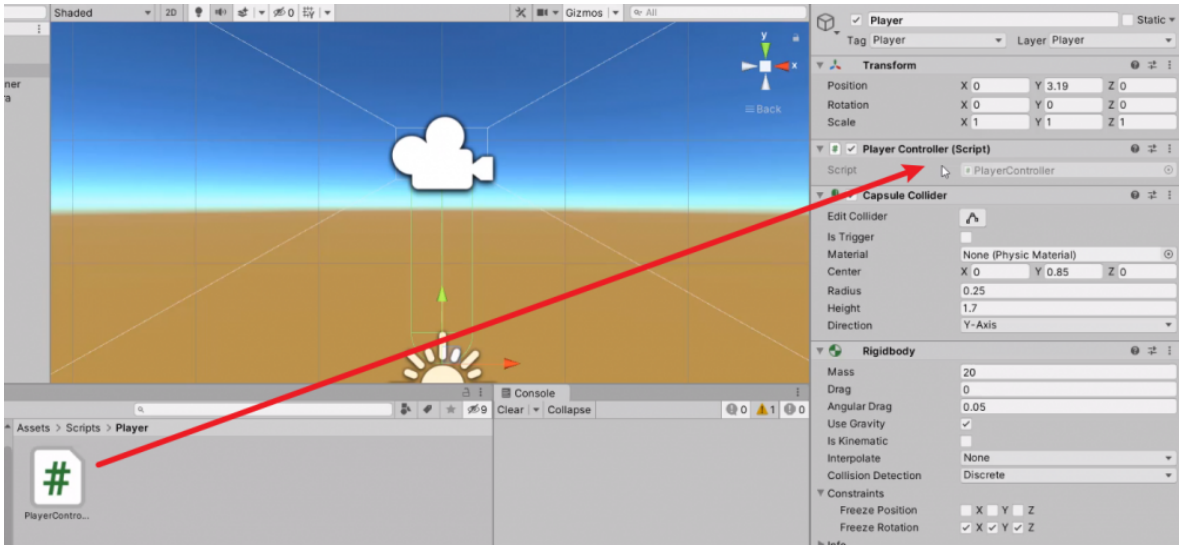
Let's set the **Mass** to be 20, and enable everything inside of **Freeze Rotation** inside the

Constraints drop-down, so our player object won't tip over and fall on its side.

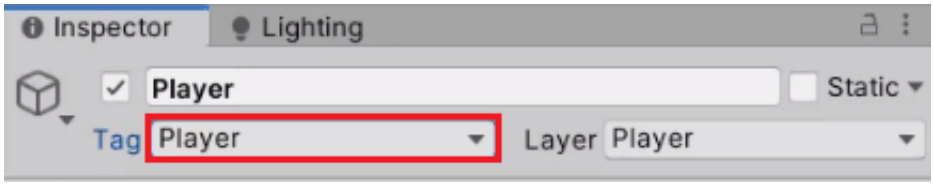


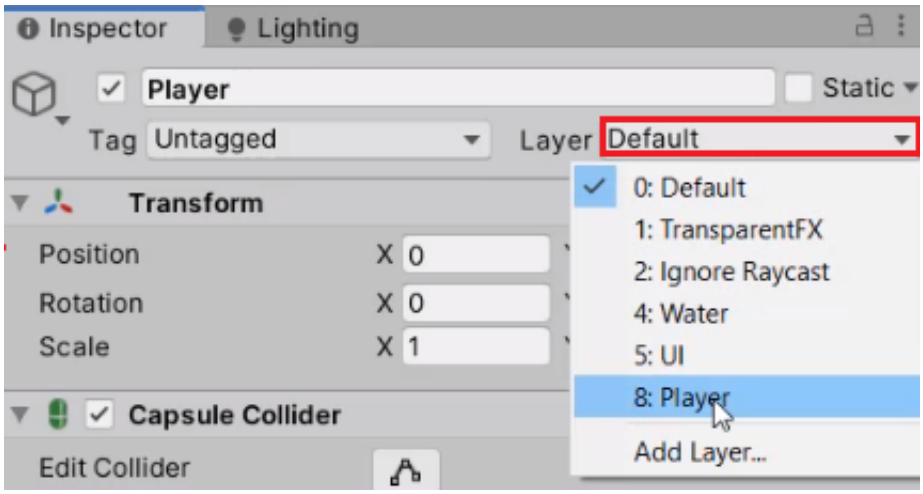
Adding Script, Tag, and Layer

Lastly, we need to create a new C# script called "**PlayerController**" and attach it to our player object.

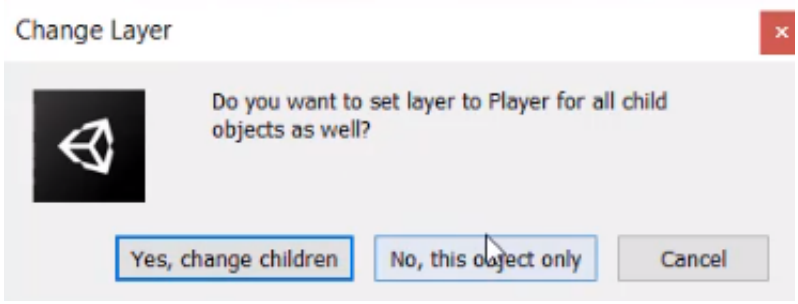


To finish off setting up the player object, let's set the **Layer** of the object as "**8: Player**", and the **Tag** to be "Player" as well.





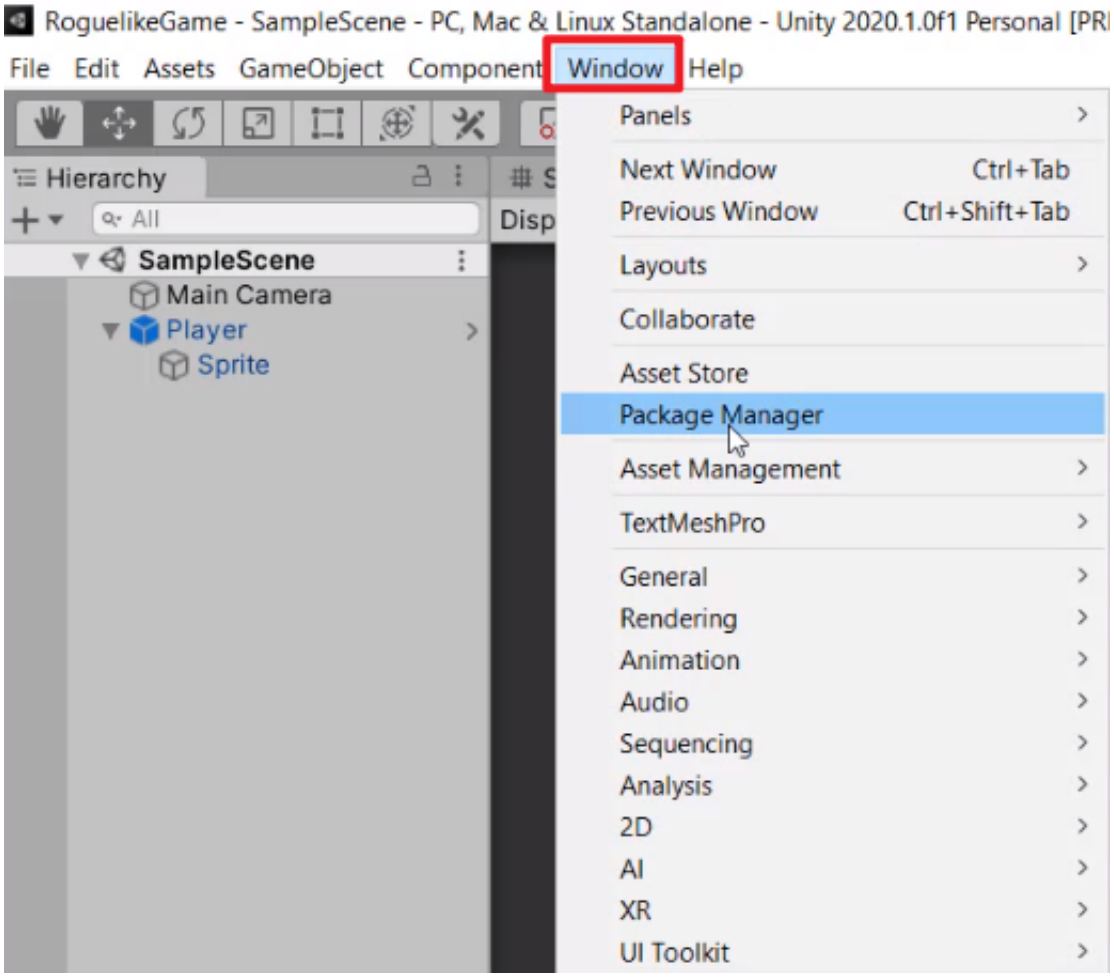
If you are asked about changing layer for all child objects, select "**Yes, change children**".



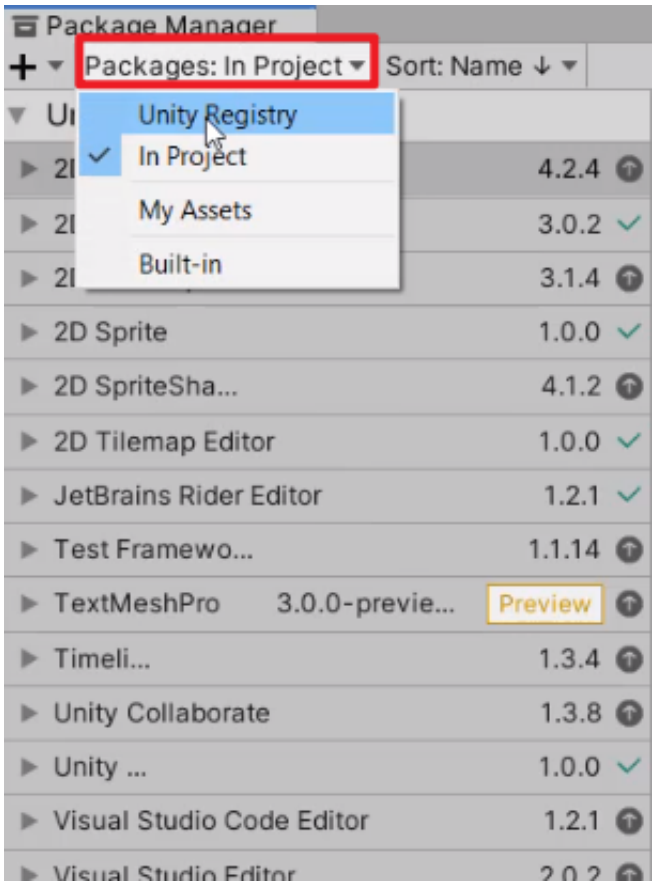
Now, we're going to be setting up our **inputs** using Unity's **Input System** package.

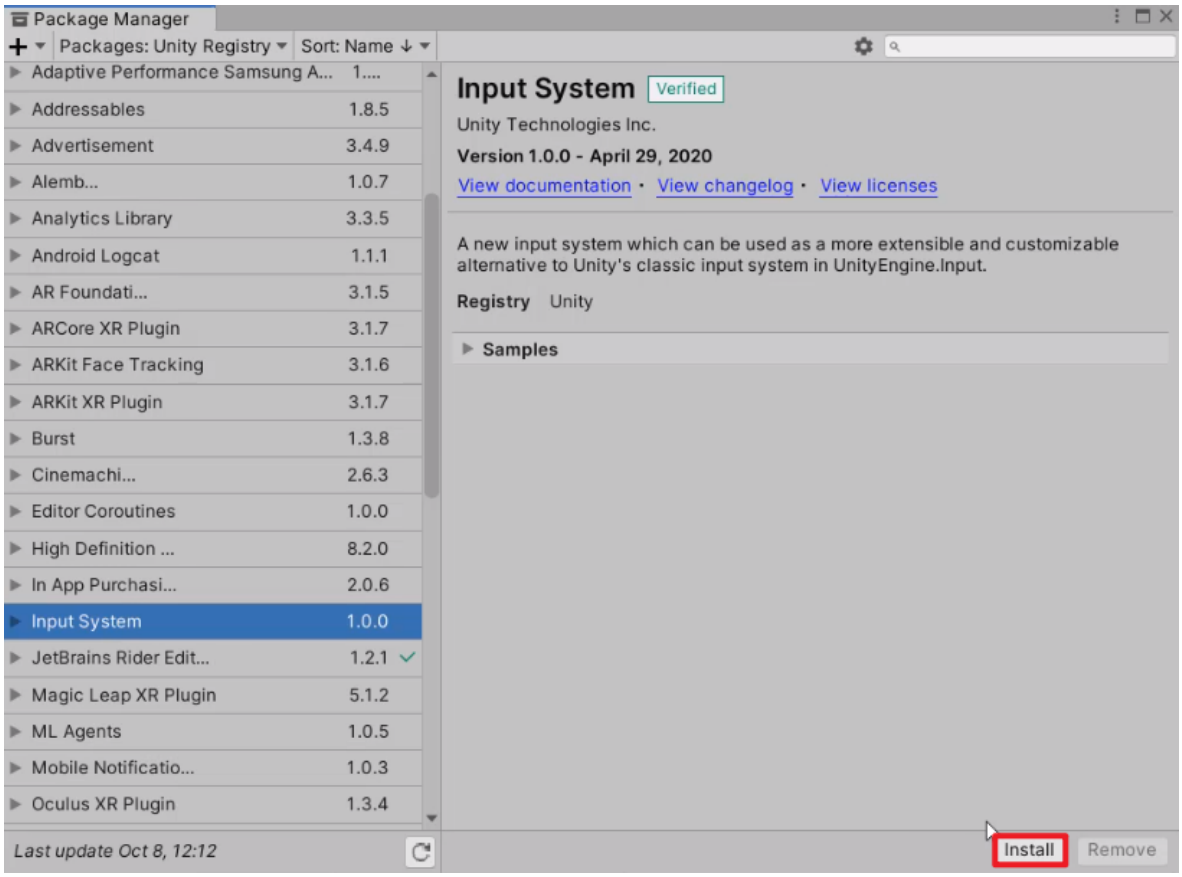
Installing Input System Package

Let's open up the **Package Manager** window (**Window > Package Manager**).



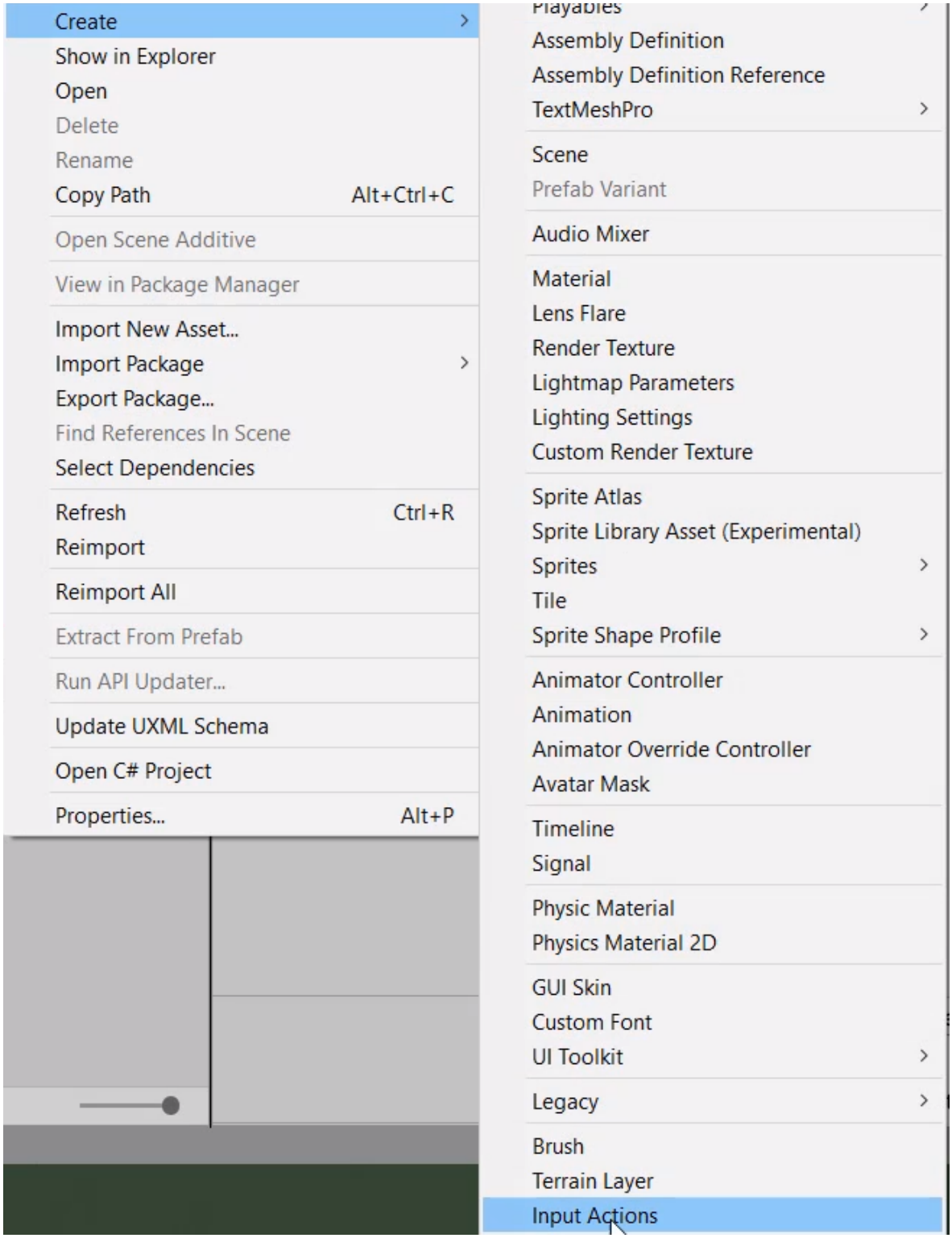
Select **Packages > Unity Registry** to access all the Unity packages, and install the **Input System**.



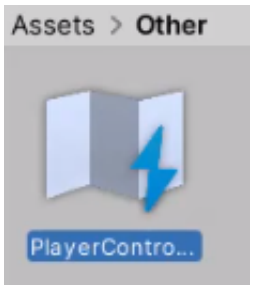


Creating Input Actions

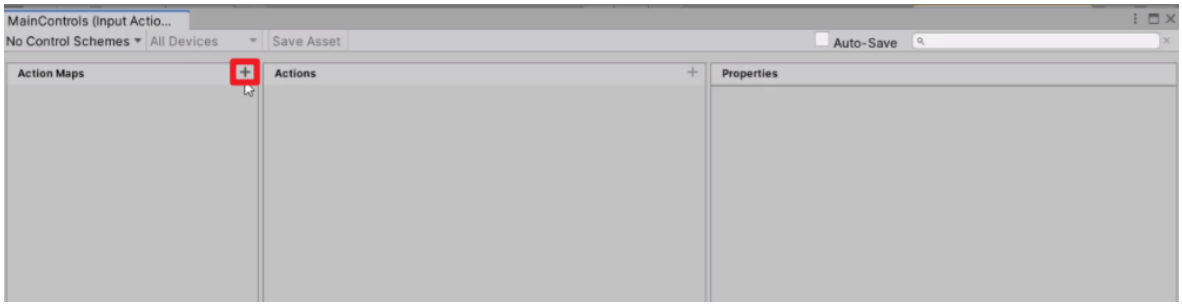
Once that's done, we can **right-click** on the Project window and click on **Create > Input Actions**.



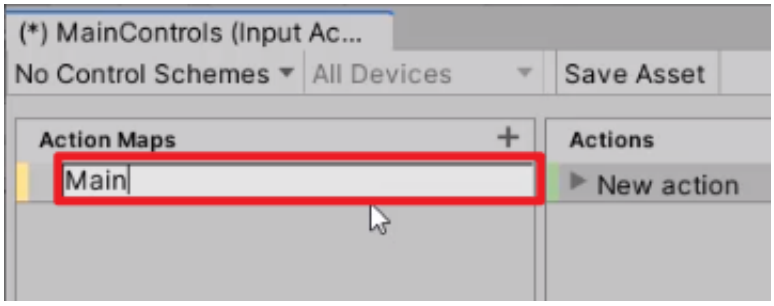
We're going to call this "**PlayerControls**" and **double-click** to open it up.



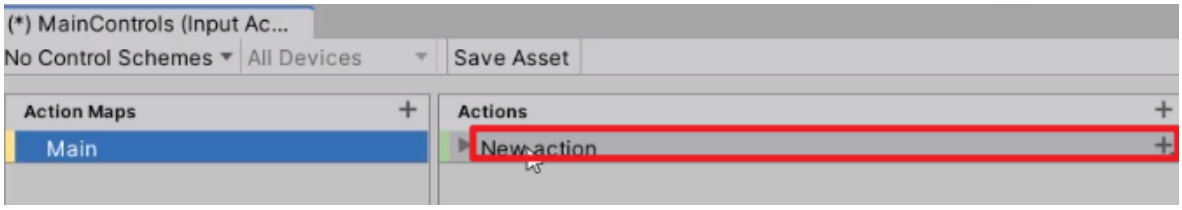
This is going to open up the **Input Actions** window. On the left panel, we can create an **Action Map** by clicking on the **+** button. This is basically a category for all your different inputs.



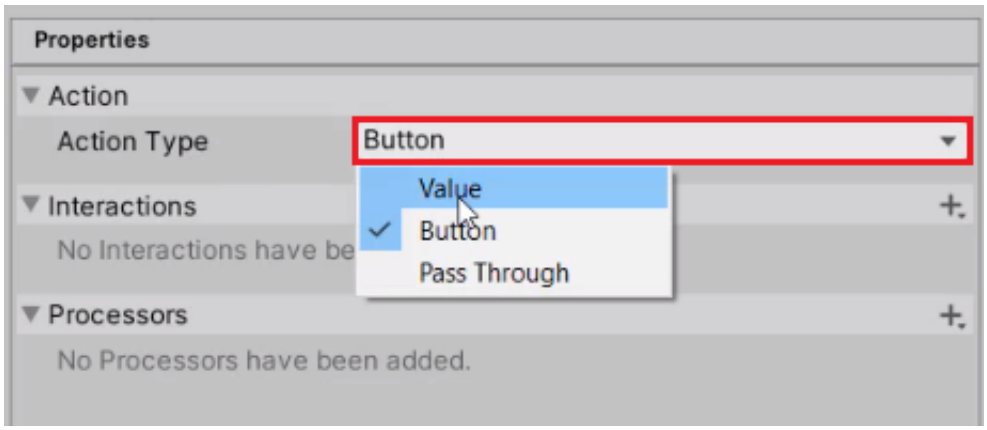
We'll call the first action map "**Main**", and create a new **Action** connected to the "Main" action map. Actions define what we want to do when pressing certain keys.



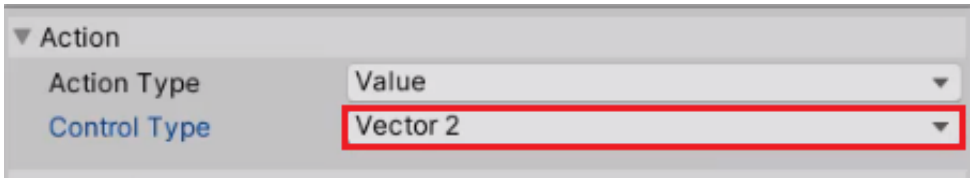
So first of all, we're going to create an action called "**Move**".



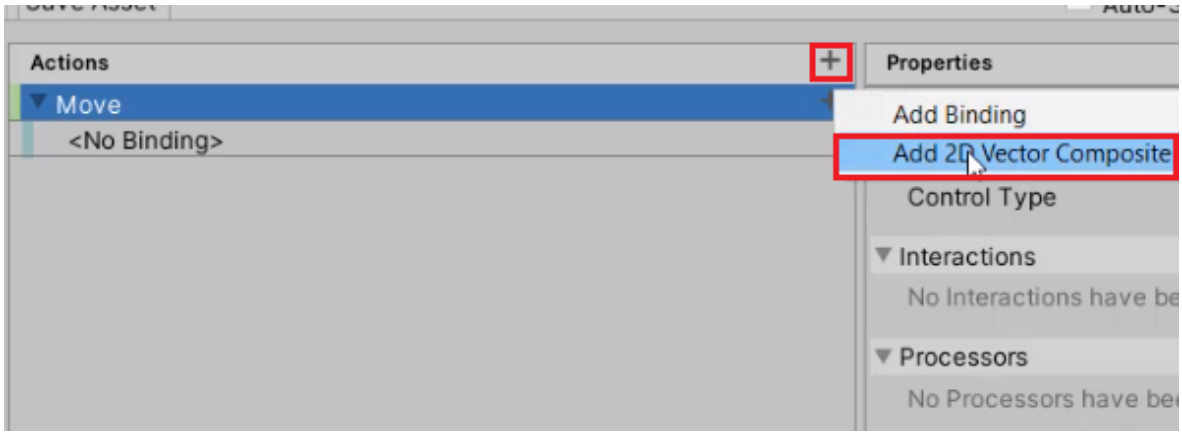
Then we want to go over to the **Properties** panel and change the **Action type** to **Value**.

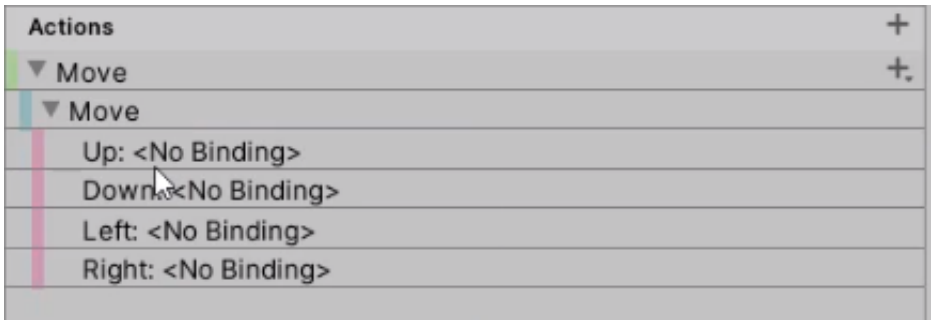


We want our movement to be represented with a **Vector2** value between -1 and 1, so we want to change the **Control Type** to **Vector2**.

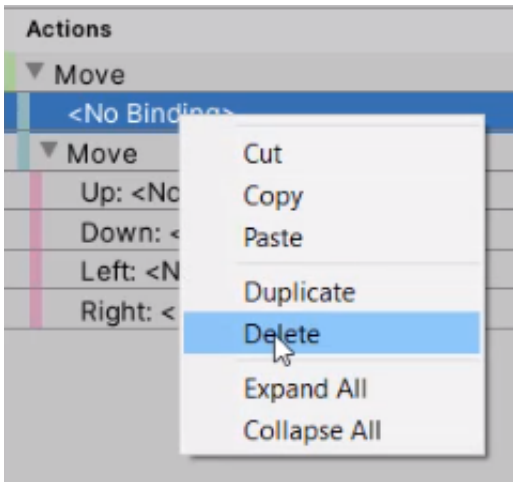


To determine which direction the player needs to move in, we can add **2D Vector Composite** to the Move action. This will create a new binding with four children bindings called Up, Down, Left, and Right.

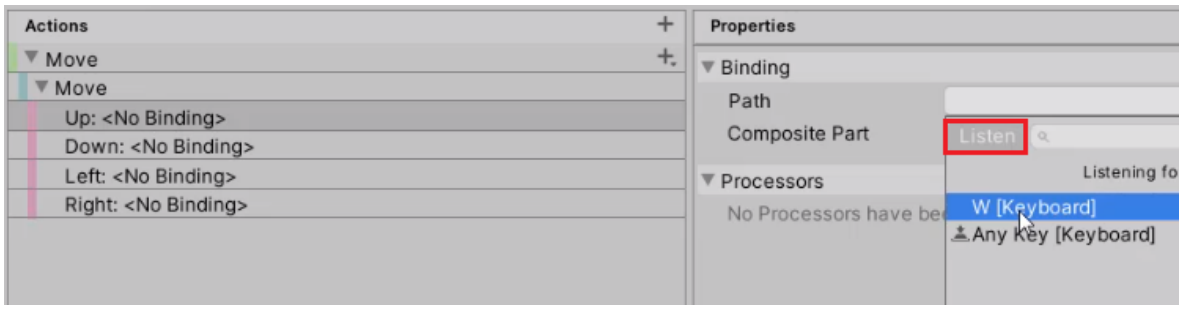




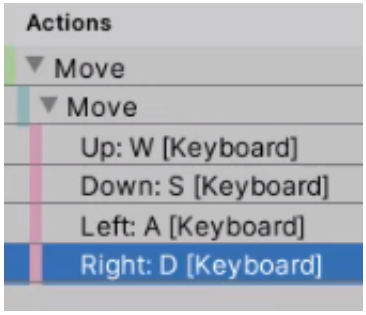
We can now **delete** the existing <No Binding> option, as we won't be needing this anymore.



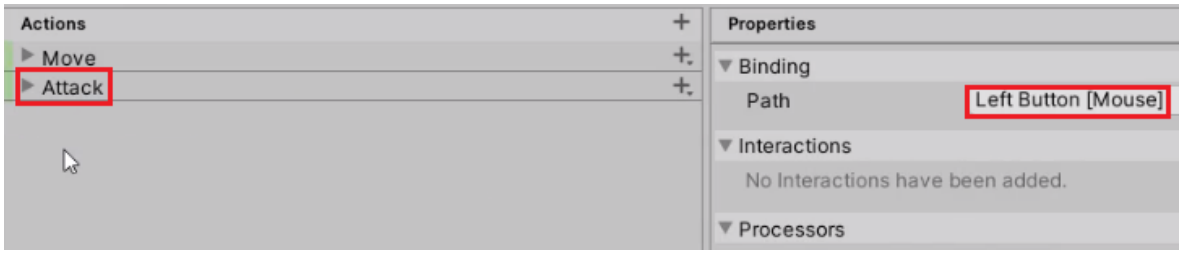
We can then bind a key to each direction by clicking on **Path > Listen**, and hit the corresponding keyboard button (W, A, S, and D).



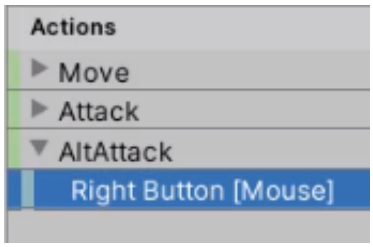
Make sure that all four directions are assigned a key.



We also need to add a new action called "**Attack**", which should be triggered by a **Left Mouse Button**.

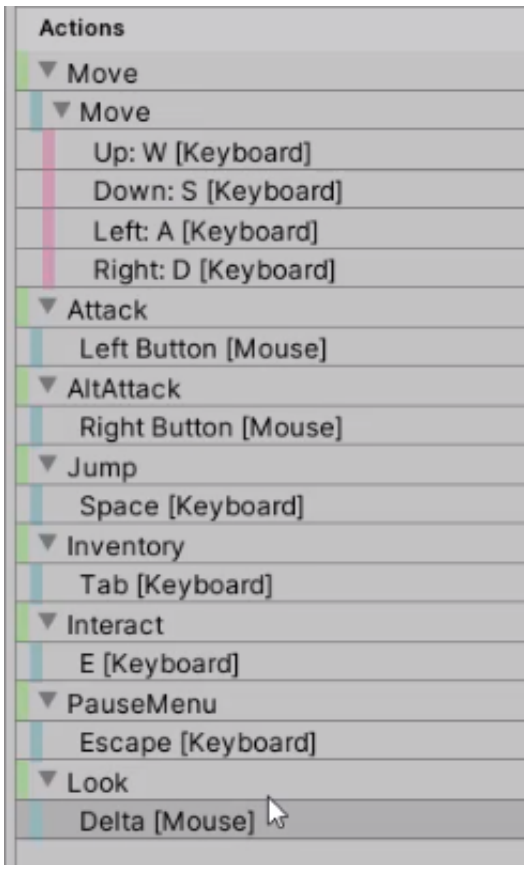


And similarly, we want to have a new action called **AltAttack** that is triggered by a **Right Mouse Button**.

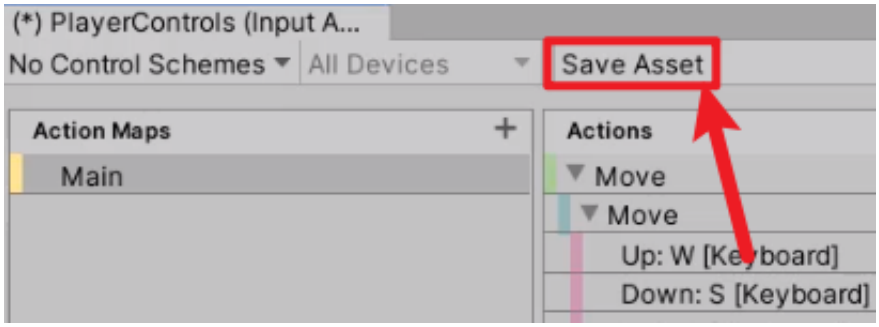


Refer to the table below to set up the rest of the actions:

Jump	Space [Keyboard]
Inventory	E [Keyboard]
PauseMenu	Escape [Keyboard]
Look	Delta [Mouse]

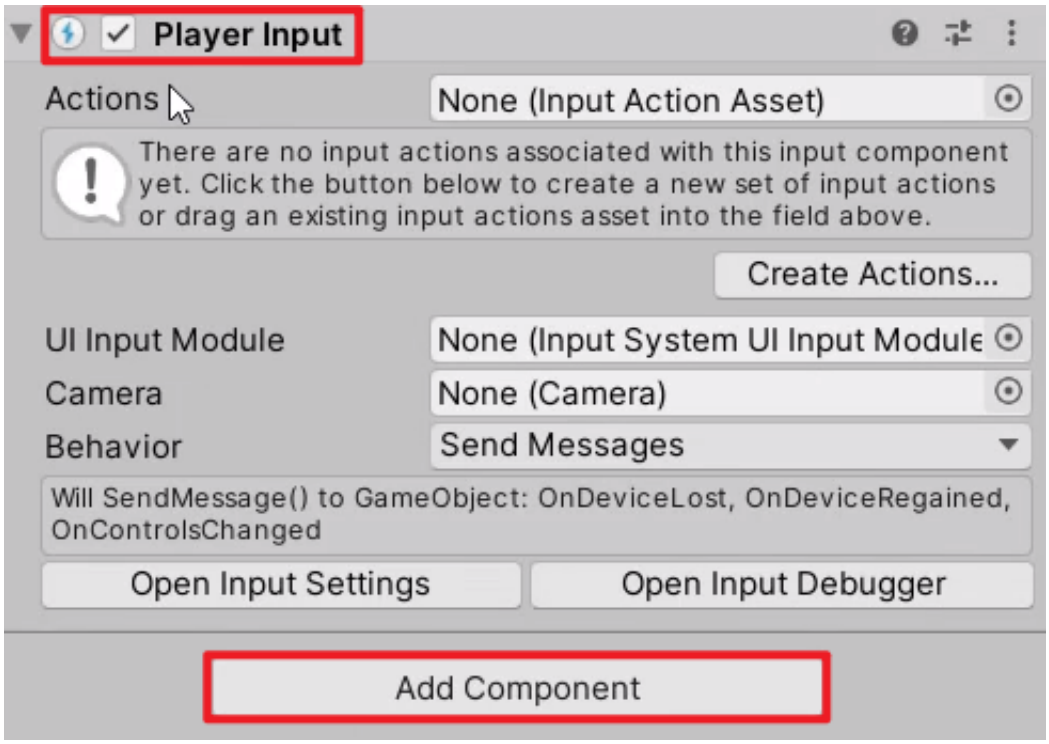


Make sure to click **Save Asset** before closing the window.

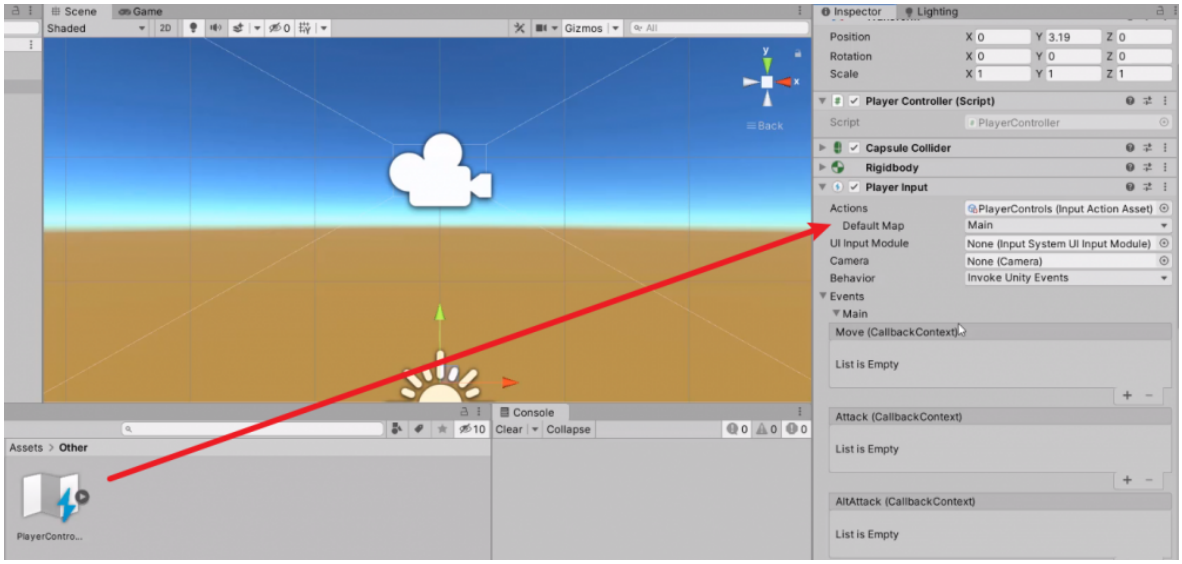


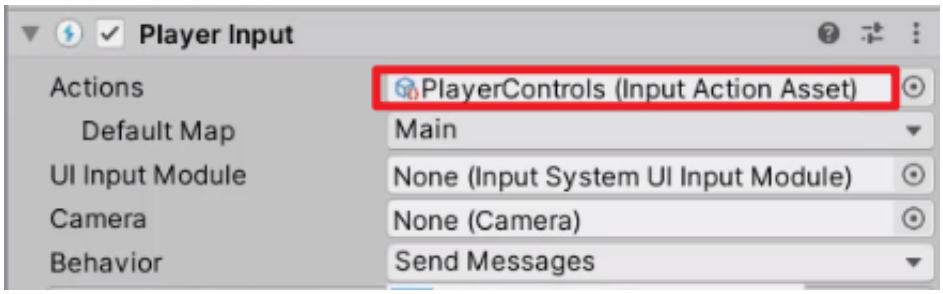
Detecting Inputs

In order to detect and make use of these inputs, we need to select our **Player** object and add a new component called "**Player Input**".

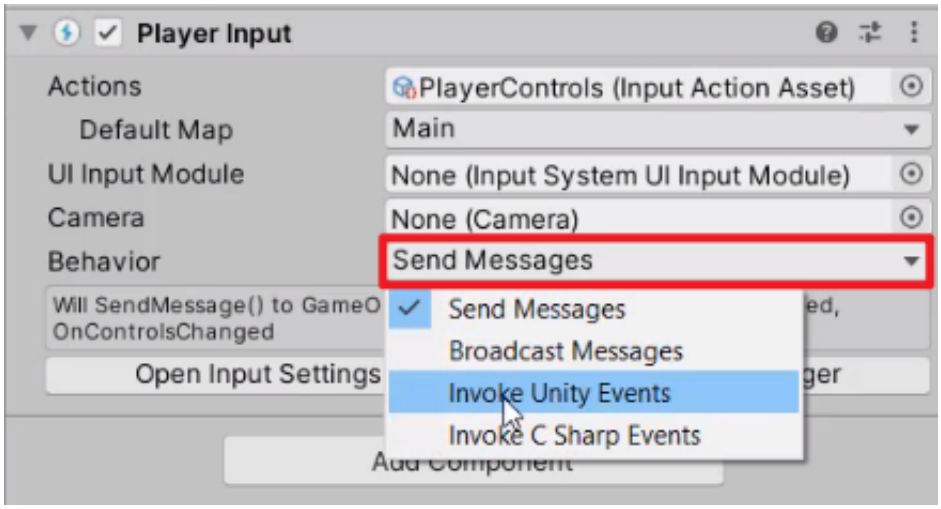


Then, drag the **Input Action** asset into the **Actions** property of the Player Input component.

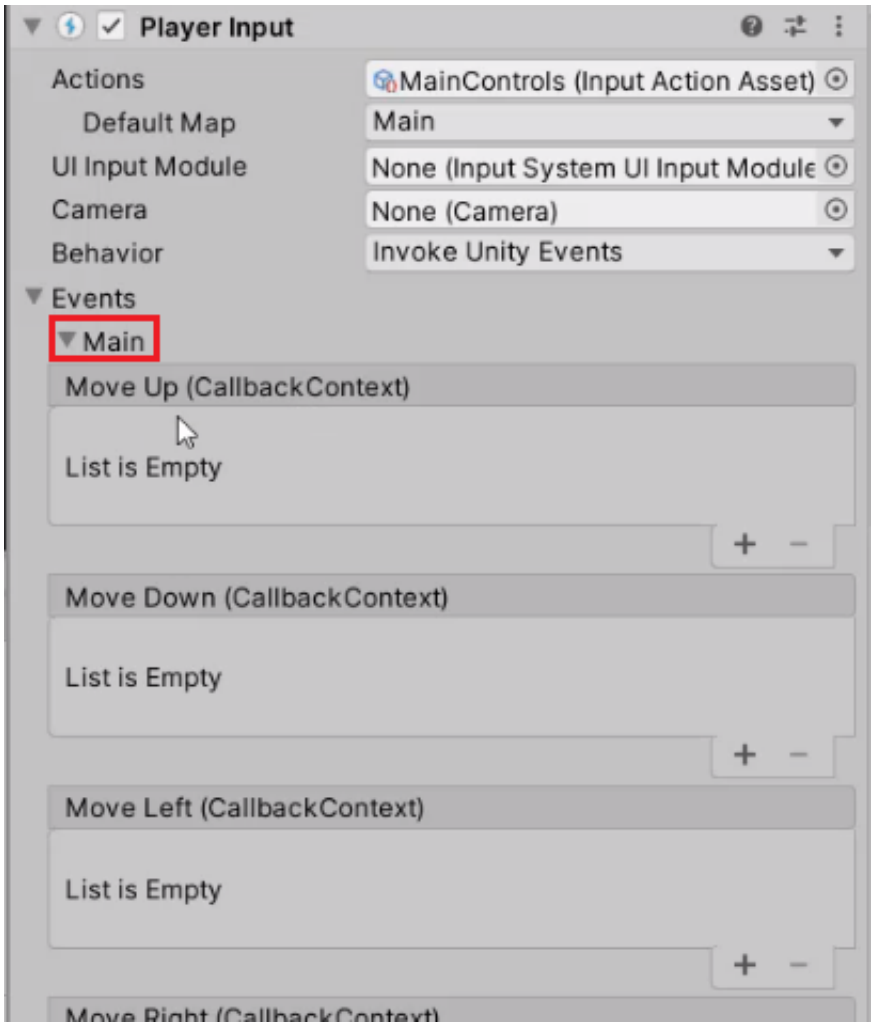




The **Behavior** property defines how we're going to be detecting when we press our buttons. Let's change the Behaviour property from '**Send Messages**' to '**Invoke Unity Events**'.



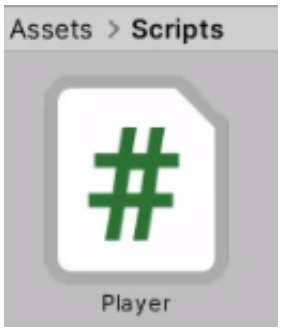
Now you'll get access to the Action Map, which contains all the actions that we created earlier. This means we can now create a function inside of a C# script and link it up here so that the function is called when the key is detected.



Scripting Camera Rotation

In the previous lesson, we set up our player's inputs. One of those inputs was the "Look" action, which had a binding of **mouse delta**. The mouse delta is basically the distance and direction that our mouse has moved each frame. In this lesson, We're going to be using that information in order to rotate our camera around.

Let's open up **Assets > Scripts**, and double-click on the **PlayerController** script.



First of all, we need to import the library package to get access to the **InputSystem** at the top of the script.

```
using UnityEngine.InputSystem;
```

Then we're going to create a function called "**OnLookInput**", which is managed by the **Input System** and is going to be called whenever we move our mouse.

```
// called when we move our mouse - managed by the Input System
public void OnLookInput (InputAction.CallbackContext context)
{

}
```

This function will take in a parameter of type **CallbackContext**, which contains various different information about the input, such as if it is pressed or not.

In our case, we only need to read the **Vector2** value. Let's store the value in a local variable called **mouseDelta**.

```
private Vector2 mouseDelta;

// called when we move our mouse - managed by the Input System
public void OnLookInput (InputAction.CallbackContext context)
{
    mouseDelta = context.ReadValue<Vector2>();
}
```

Now, we want to update the camera in the **LateUpdate** function, because we want to rotate the

camera after the player has moved.

```
void LateUpdate ()
{
    CameraLook();
}
```

Inside the **CameraLook** function, we're going to rotate the camera container up and down. To do this, we want to multiply the **mouse delta (y)** value by **look sensitivity**, add it to our **camCurXRot** variable, and clamp it between the min/max values.

```
void CameraLook ()
{
    // rotate the camera container up and down
    camCurXRot += mouseDelta.y * lookSensitivity;
    camCurXRot = Mathf.Clamp(camCurXRot, minXLook, maxXLook);
}
```

Now, we can apply this value to the local Euler angle of our **camera container** along the X-axis.

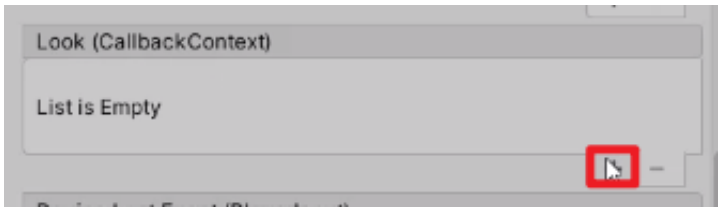
Note that by default the mouse rotation is going to be inverted, which means the camera will look up when the mouse moves downward. To correct the inversion, you can set the **camCurXRot** to be negative.

```
void CameraLook ()
{
    // rotate the camera container up and down
    camCurXRot += mouseDelta.y * lookSensitivity;
    camCurXRot = Mathf.Clamp(camCurXRot, minXLook, maxXLook);
    cameraContainer.localEulerAngles = new Vector3(-camCurXRot, 0, 0);
}
```

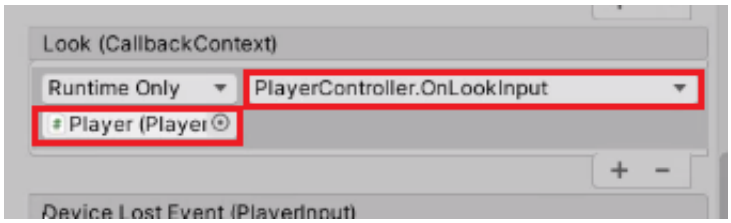
Make sure to save the script before returning to the Editor.

Linking Script To Events

We can now click on the **Add** (+) button on the **Look** section of the **Player Input** component to add a listener ...

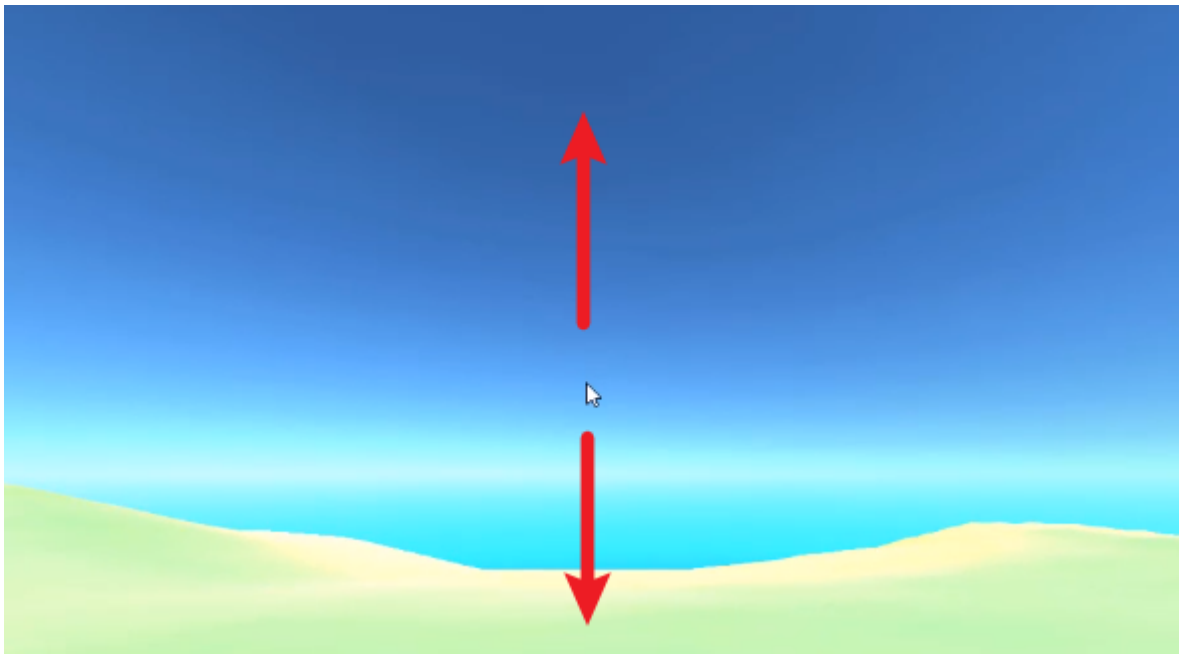


... and drag the **Player** script into the **Object** field. This will allow us to select the **OnLookInput** function inside the script.



(If they're not appearing here, make sure to check that the functions are set to be 'public' and that your script is saved to apply those changes.)

Now if you move your mouse up and down, you'll see your camera rotates along the x-axis.

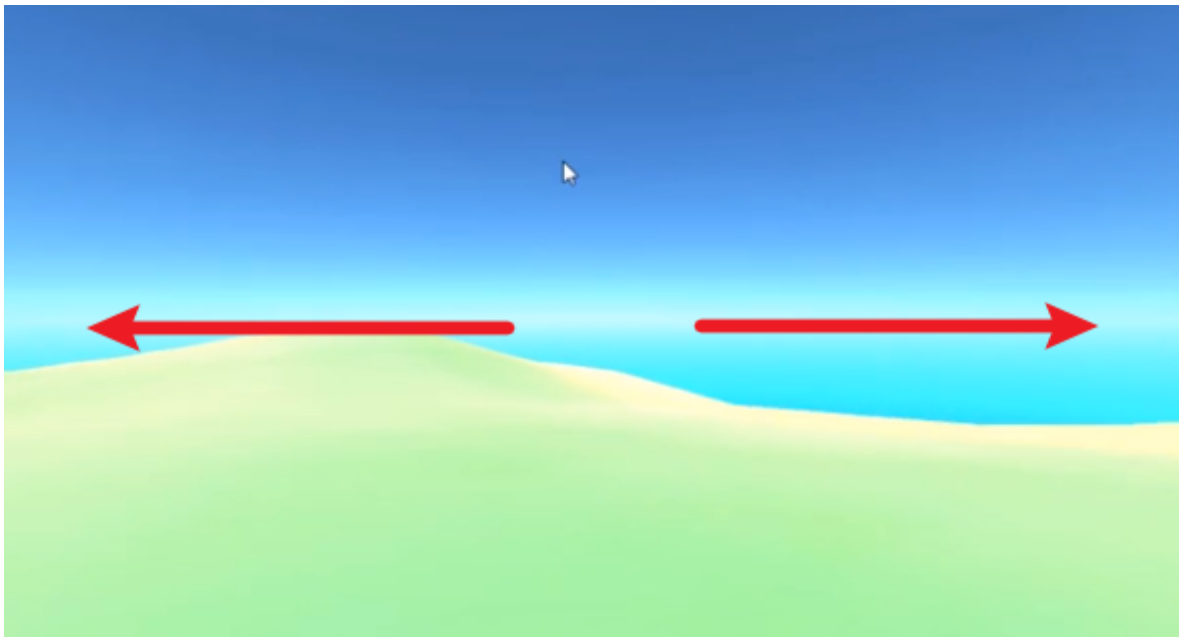


Horizontal Rotation

Let's return to our **PlayerController** script and add this new line at the bottom of the **CameraLook** function:

```
void CameraLook ()
{
    ...

    // rotate the player left and right
    transform.eulerAngles += new Vector3(0, mouseDelta.x * lookSensitivity, 0);
}
```

This will implement camera rotation along the y-axis, which will allow us to look left and right by our mouse.



Hiding The Cursor

Finally, let's lock the cursor into the screen and prevent accidental click outside the application window.

```
void Start ()
{
    // lock the cursor at the start of the game
    Cursor.lockState = CursorLockMode.Locked;
}
```

The cursor should now be hidden and confined inside the screen.

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

Categorizing Variables

Before we start implementing player movement, we want to categorize our existing variables.

All of the variables that we created so far in **PlayerController.cs** are used for looking around, so let's go ahead and create a new header called "Look".

```
[Header("Look")]
public Transform cameraContainer;
public float minXLook;
public float maxXLook;
private float camCurXRot;
public float lookSensitivity;
```

Then let's create a new header called "Movement", and fill it up with the following variables.

```
[Header("Movement")]
public float moveSpeed;
private Vector2 curMovementInput;
```

Detecting Input

The next thing we're going to do is to check if we have pressed the W/A/S/D keys.

Similar to the mouse input detection, we need to create a new function called **OnMoveInput.** Just as the **OnLookInput** function we created in the previous lesson, this function is going to be managed by the **Input System**, and we can use the **CallbackContext** parameter to detect when we press a movement button.

```
// called when we press WASD - managed by the Input System
public void OnMoveInput (InputAction.CallbackContext context)
{
    // are we holding down a movement button?
    if(context.phase == InputActionPhase.Performed)
    {
    }
}
```

If a movement button is detected, we want to read the **Vector2** value from the context and store it in our **curMovementInput** variable. If we let go of that button, then we want to reset the **curMovementInput** variable to be zero and stop moving.

```
// called when we press WASD - managed by the Input System
public void OnMoveInput (InputAction.CallbackContext context)
{
    // are we holding down a movement button?
```



```
    if(context.phase == InputActionPhase.Performed)
    {
        curMovementInput = context.ReadValue<Vector2>();
    }
    // have we let go of a movement button?
    else if(context.phase == InputActionPhase.Canceled)
    {
        curMovementInput = Vector2.zero;
    }
}
```

Inside the **FixedUpdate** function, we're going to call a new function called "**Move**", which we will create now to figure out which direction we want to move in.

```
public class PlayerController : MonoBehaviour
{
   void FixedUpdate ()
   {
       Move();
   }

   void Move ()
   {

   }
}
```

The **Move** function will be taking our forward and right direction into consideration, using a temporary **Vector3** variable called "**dir**". This is so that we can move in the direction that is relative to where we are facing.

```
void Move ()
{
    // calculate the move direction relative to where we're facing.
    Vector3 dir = transform.forward * curMovementInput.y + transform.right * curMovem
entInput.x;
    dir *= moveSpeed;
}
```

When our player is jumping or falling, it will be affected by gravity, which modifies the player object's y-velocity. So we need to get the y-velocity from our **Rigidbody** component, and apply it to the "**dir.y**" value.

```
// components
private Rigidbody rig;

void Awake ()
{
    // get our components
```



```
    rig = GetComponent<Rigidbody>();
}



void Move ()
{
    // calculate the move direction relative to where we're facing.
    Vector3 dir = transform.forward * curMovementInput.y + transform.right * curMovem
entInput.x;
    dir *= moveSpeed;
}
```

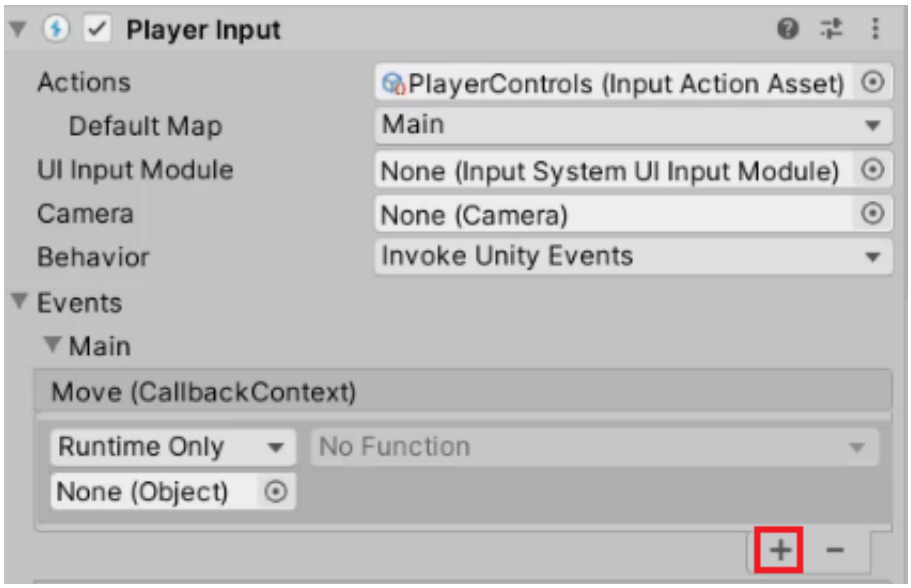
Then the final **dir** value can be assigned back to our Rigidbody velocity.

The following code has been updated, and differs from the video: For **Unity 6** LTS, the scripting API has changed for rig.velocity, now use rig.linearVelocity.

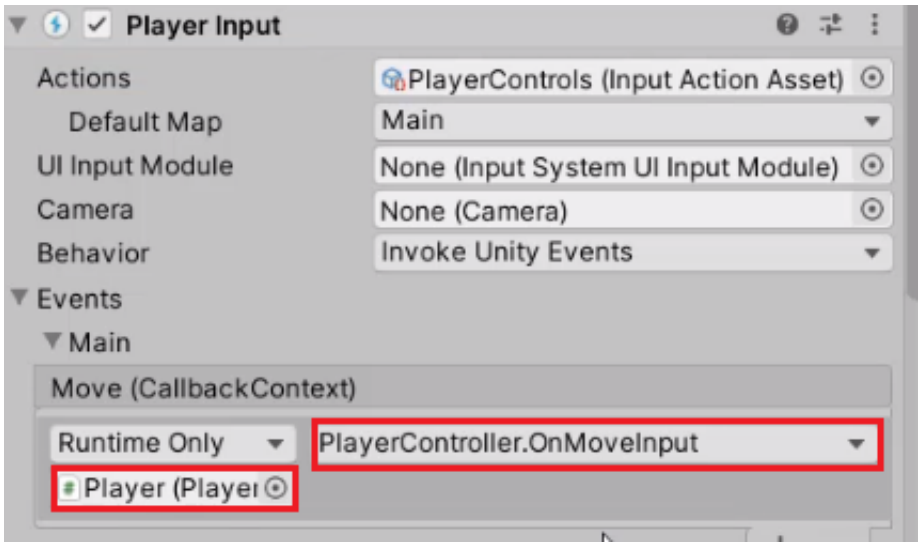
```
void Move ()
{
    // calculate the move direction relative to where we're facing.
    Vector3 dir = transform.forward * curMovementInput.y + transform.right * curMovem
entInput.x;
    dir *= moveSpeed;
    dir.y = rig.linearVelocity.y;

    // assign our Rigidbody velocity
    rig.linearVelocity = dir;
}
```

Linking Script to InputAction Event

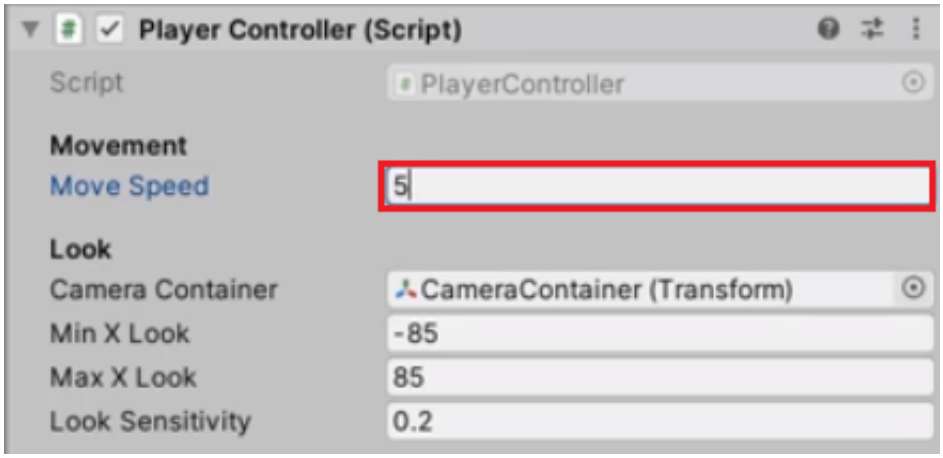
Let’s save the script, open up the Editor, and go to **PlayerInput > Events > Main**. Here, we can add a new listener to link with the **OnMoveInput** function.

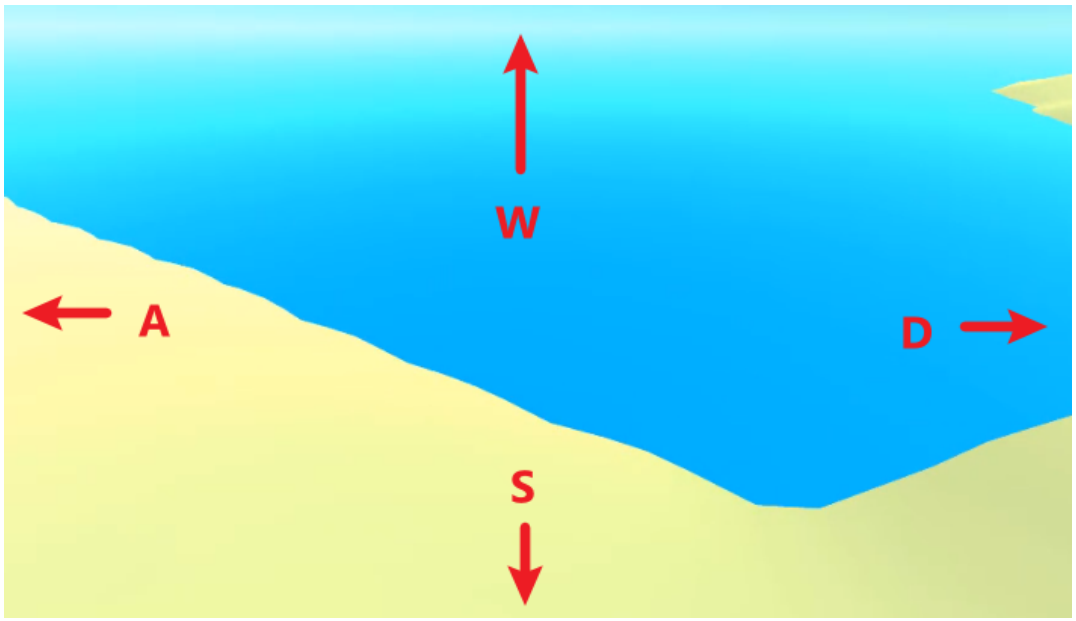


Drag the **Player** object into the '**None (Object)**' field, and set the **OnMoveInput** to be called when the Move action is triggered.



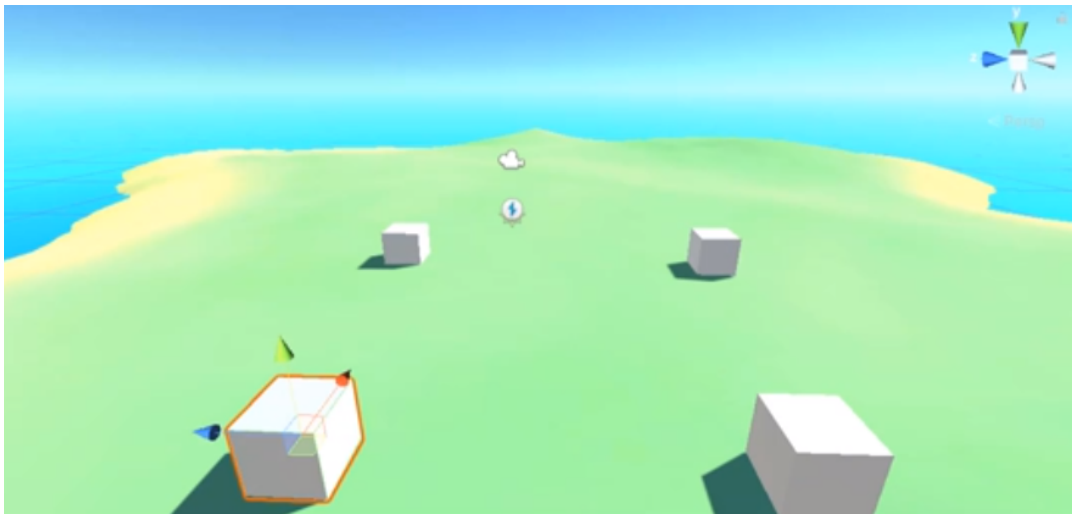
Finally, set a value for the **MoveSpeed** inside the **PlayerController** component, and press Play to test it out. You should now be able to move around by pressing the W/A/S/D key.





Now that we've set up looking and moving around, we're going to start implementing jumping with the space key.

But before we begin, let's add in some testing cubes so that we can actually see where we are in relation to the island.



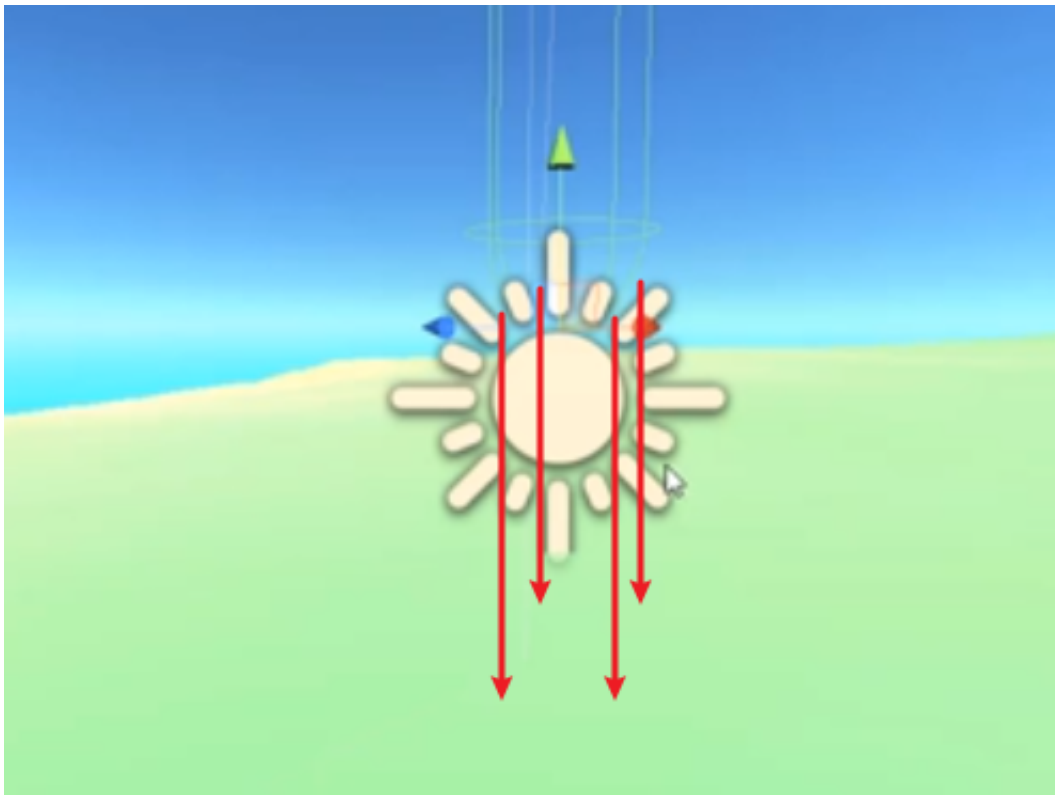
Declaring Variables

Since jumping is related to movement, we're going to add the following variables under the "**Movement**" header.

```
[Header("Movement")]
public float jumpForce;
public LayerMask groundLayerMask;
```

The **LayerMask** variable is going to be used to check if we're standing on a surface. This is going to be done by **Raycasting**, which is basically like shooting lasers that return information about the object that they hit.

So on each corner of the player's base, we're going to be shooting down four **Raycasts** downwards to see if we're standing on the ground.



When we do this, we want to filter out the player object itself, in other words, the “Player” layer is going to be ignored.

Ground Check

Let’s create a new function called **IsGrounded**, that returns a boolean.

```
bool IsGrounded ()
{

}
```

Whenever we press the jump button (i.e. inside the **OnJumpInput** function) we want to call the **IsGrounded** function. If this function returns true, then we can add force upwards using **Rigidbody.AddForce**.

```
// called when we press down on the spacebar - managed by the Input System
public void OnJumpInput (InputAction.CallbackContext context)
{
    // is this the first frame we're pressing the button?
    if(context.phase == InputActionPhase.Started)
    {
        // are we standing on the ground?
        if(IsGrounded())
        {
            // add force updwards
            rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
        }
```



```
    }
}
```

When using **Rigidbody.AddForce**, we can set a certain **ForceMode** to specify how the force should behave. 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.

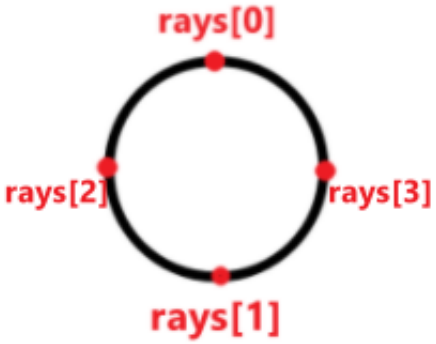
For more information, refer to:

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

Inside the **IsGrounded** function, we'll be shooting four **Rays** from the player's feet, setting the direction as **Vector3.down**.

```
bool IsGrounded ()
{
    Ray[] rays = new Ray[4]
    {
        new Ray(transform.position + (transform.forward * 0.2f), Vector3.down),
        new Ray(transform.position + (-transform.forward * 0.2f), Vector3.down),
        new Ray(transform.position + (-transform.right * 0.2f), Vector3.down),
        new Ray(transform.position + (transform.right * 0.2f), Vector3.down)
    };
}
```

In the diagram below, the large circle represents the bottom of the player capsule, and the red dots show us where each ray is located at.



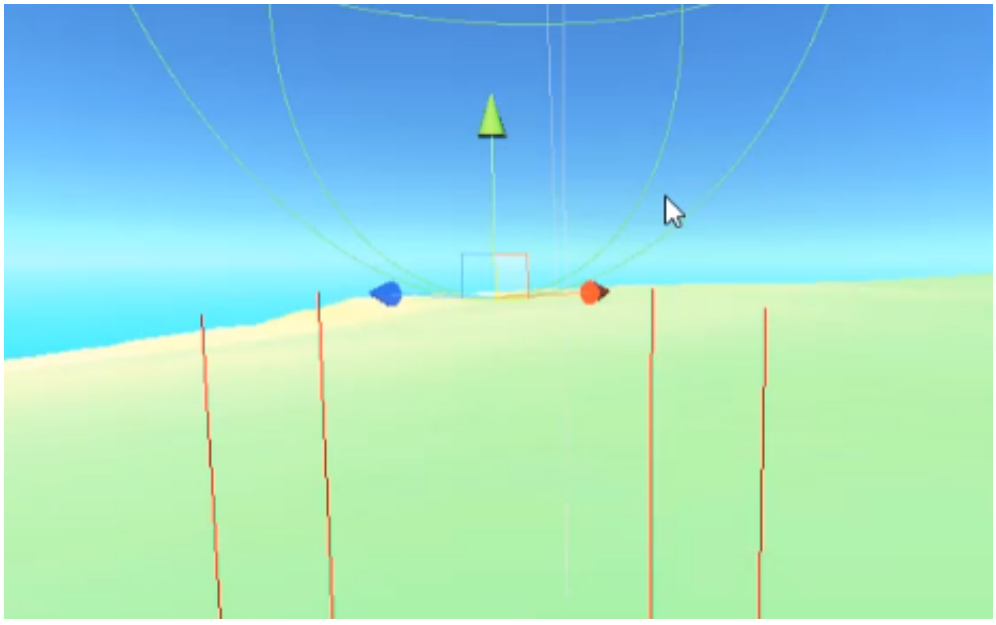
Drawing Rays

To see the four rays shooting downwards in the scene view, we're going to create a function called **OnDrawGizmo**. Inside here, we're going to set the color of the gizmo to be red:

```
private void OnDrawGizmos ()
{
    Gizmos.color = Color.red;
}
```

Then we can call **Gizmos.DrawRay** four times inside the function, sending the Vector3 values used to create each ray as an argument.

```
private void OnDrawGizmos ()
{
    Gizmos.color = Color.red;

    Gizmos.DrawRay(transform.position + (transform.forward * 0.2f), Vector3.down);
    Gizmos.DrawRay(transform.position + (-transform.forward * 0.2f), Vector3.down);
    Gizmos.DrawRay(transform.position + (transform.right * 0.2f), Vector3.down);
    Gizmos.DrawRay(transform.position + (-transform.right * 0.2f), Vector3.down);
}
```



Detecting Ground

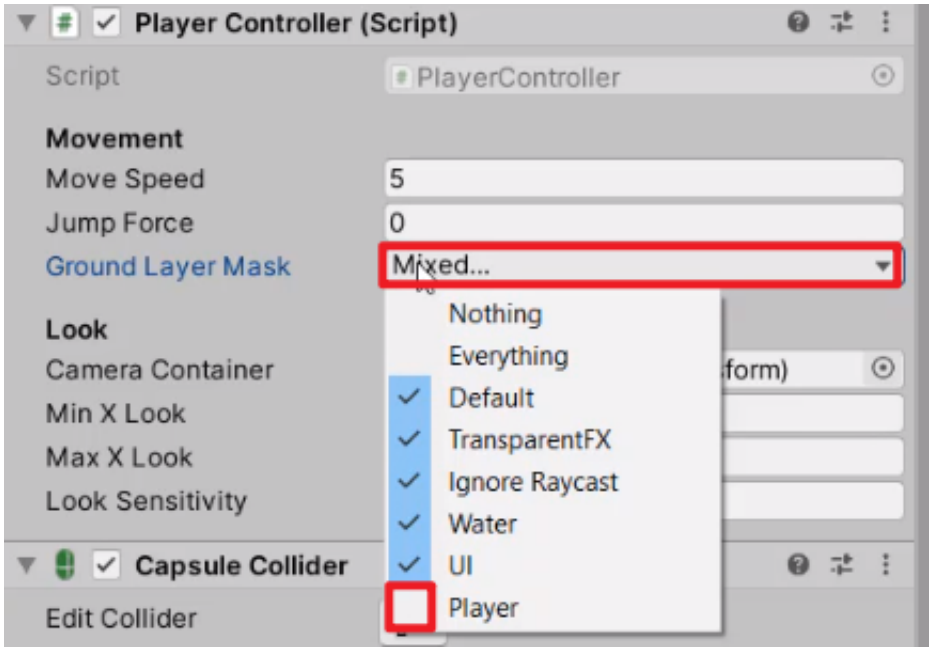
For each of the rays that we created inside the **IsGrounded** function, we're going to actually shoot them using **Physics.Raycast**, setting the maximum distance as **0.1** units (=10 centimeters).

```
bool IsGrounded ()
{
    Ray[] rays = new Ray[4]
```

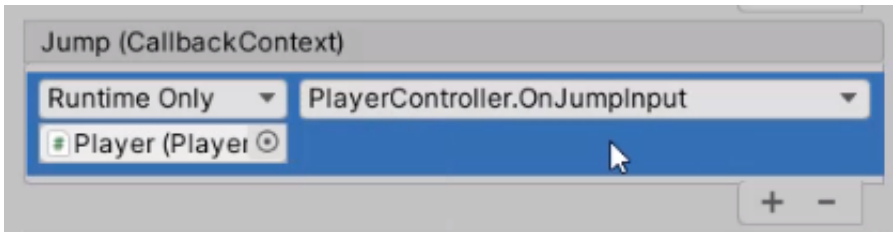
```
    {
        new Ray(transform.position + (transform.forward * 0.2f), Vector3.down),
        new Ray(transform.position + (-transform.forward * 0.2f), Vector3.down),
        new Ray(transform.position + (transform.right * 0.2f), Vector3.down),
        new Ray(transform.position + (-transform.right * 0.2f), Vector3.down)
    };

    for(int i = 0; i < rays.Length; i++)
    {
        if(Physics.Raycast(rays[i], 0.1f, groundLayerMask))
        {
            return true;
        }
    }

    return false;
}
```

If the raycast hits an object that is labeled as '**groundLayerMask**', the function will return true. Otherwise, it will move on to the next iteration. If none of them have hit anything, then the function will return false.

Let's save the script, and define **Ground LayerMask** in the Inspector. We should select everything except for 'Player', so the ray can ignore the player object.



We also need to set a value for **Jump Force** and link the player script with **Jump Input Action** in the **Player Input** component.

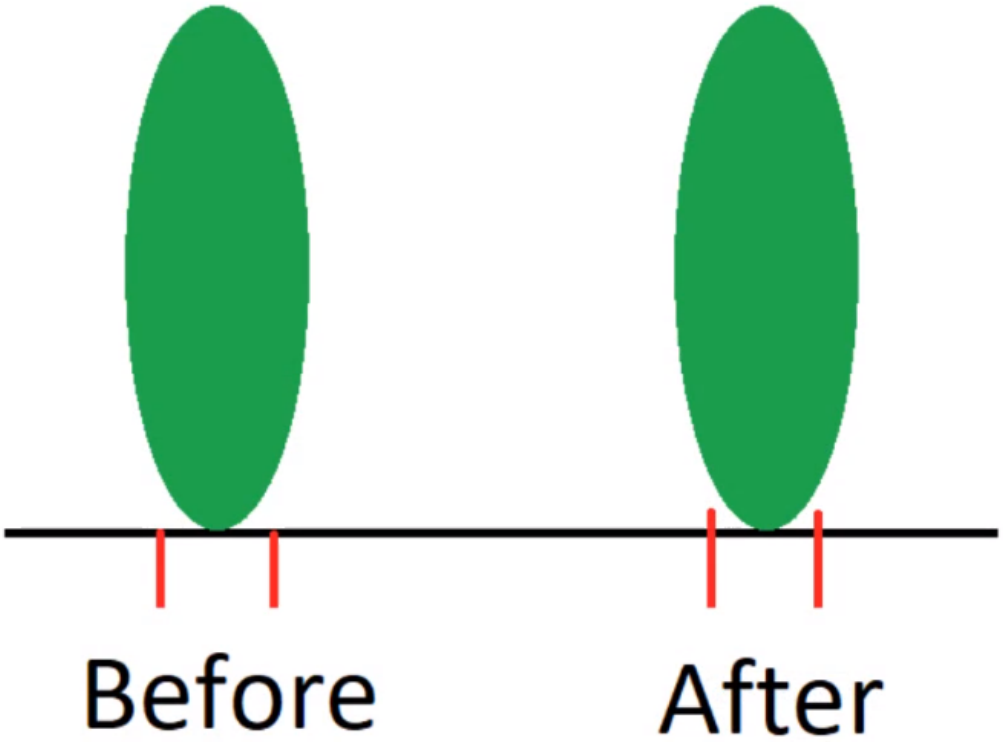


Testing Out

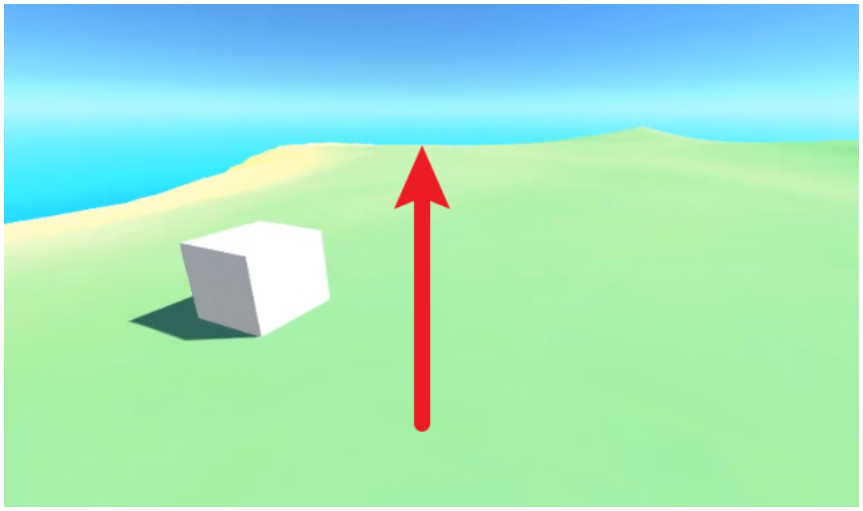
Now you can jump by hitting the Space key, but you will notice that it is buggy sometimes the jump key is ignored. This is because ray casts are not going to get detected properly if they start within a collider.

So we're going to move the rays upwards a bit so they are starting to shoot down from slightly above the feet.

```
bool IsGrounded ()
{
    Ray[] rays = new Ray[4]
    {
        new Ray(transform.position + (transform.forward * 0.2f) + (Vector3.up * 0.01f
), Vector3.down),
        new Ray(transform.position + (-transform.forward * 0.2f) + (Vector3.up * 0.01
f), Vector3.down),
        new Ray(transform.position + (transform.right * 0.2f) + (Vector3.up * 0.01f),
 Vector3.down),
        new Ray(transform.position + (-transform.right * 0.2f) + (Vector3.up * 0.01f)
, Vector3.down)
    };

    for(int i = 0; i < rays.Length; i++)
    {
        if(Physics.Raycast(rays[i], 0.1f, groundLayerMask))
        {
            return true;
        }
    }

    return false;
}
```

Now you should be able to jump properly by hitting the space key.



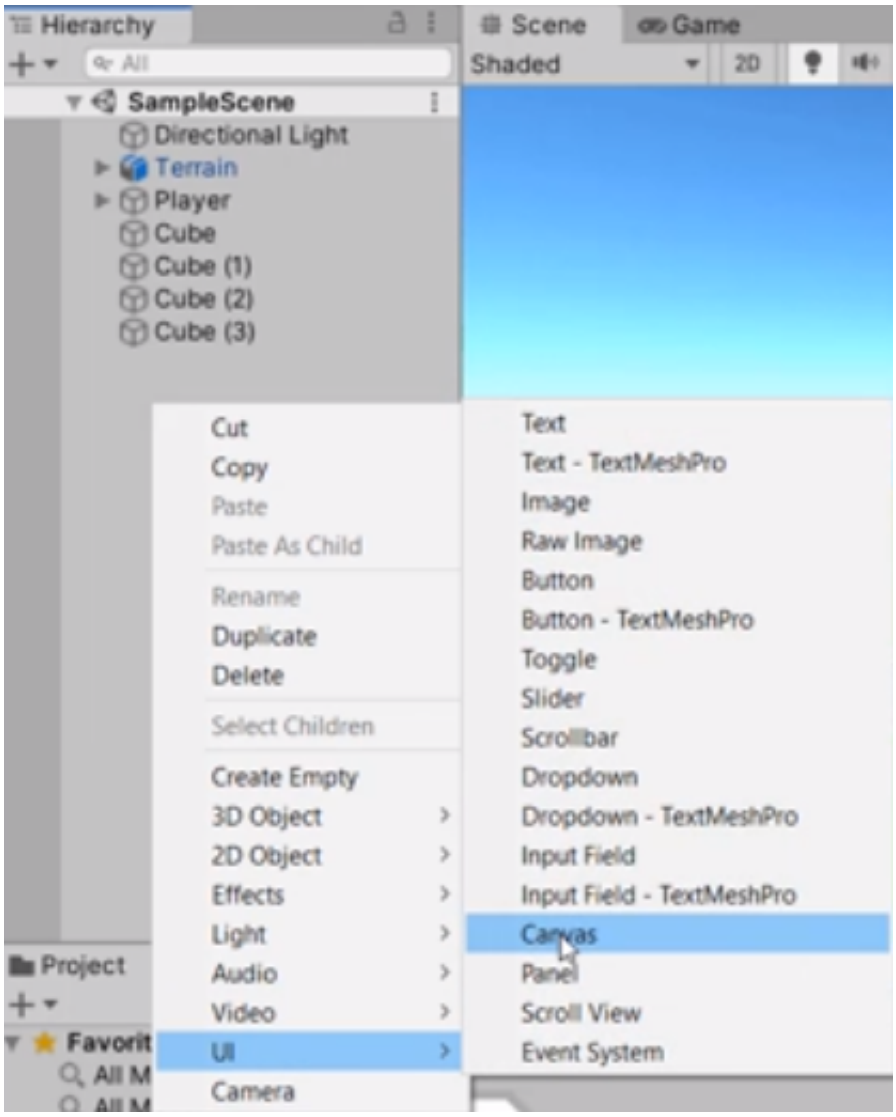
The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected code.

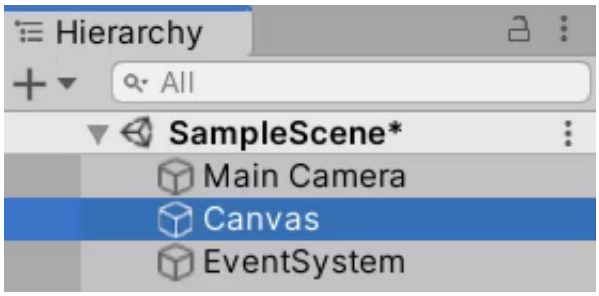
In this lesson, we will be setting up the visual **UI** (User Interface) of the game.

Creating UI Canvas

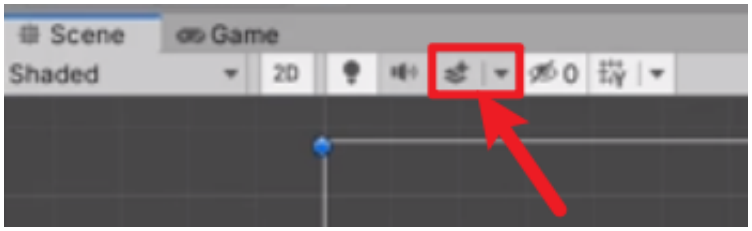
First of all, we're going to create a new gameObject called **Canvas,** which contains all UI elements.

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

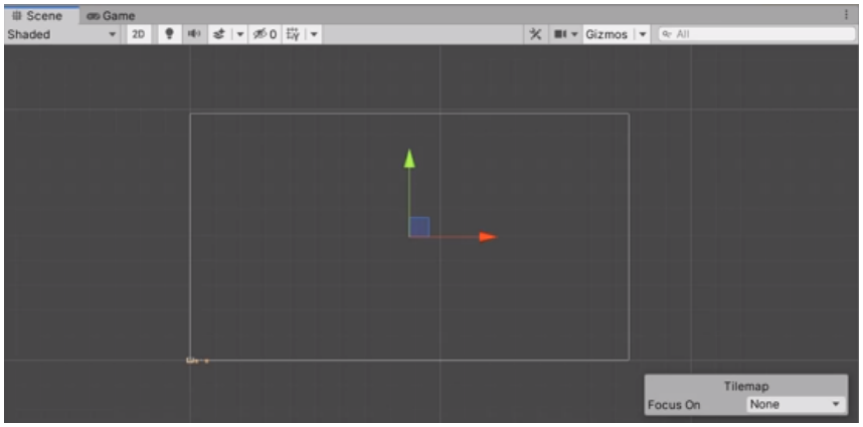




The skybox can be quite distracting when working on UI. For now, let's switch it off by clicking on the **Toggle Effects** button.

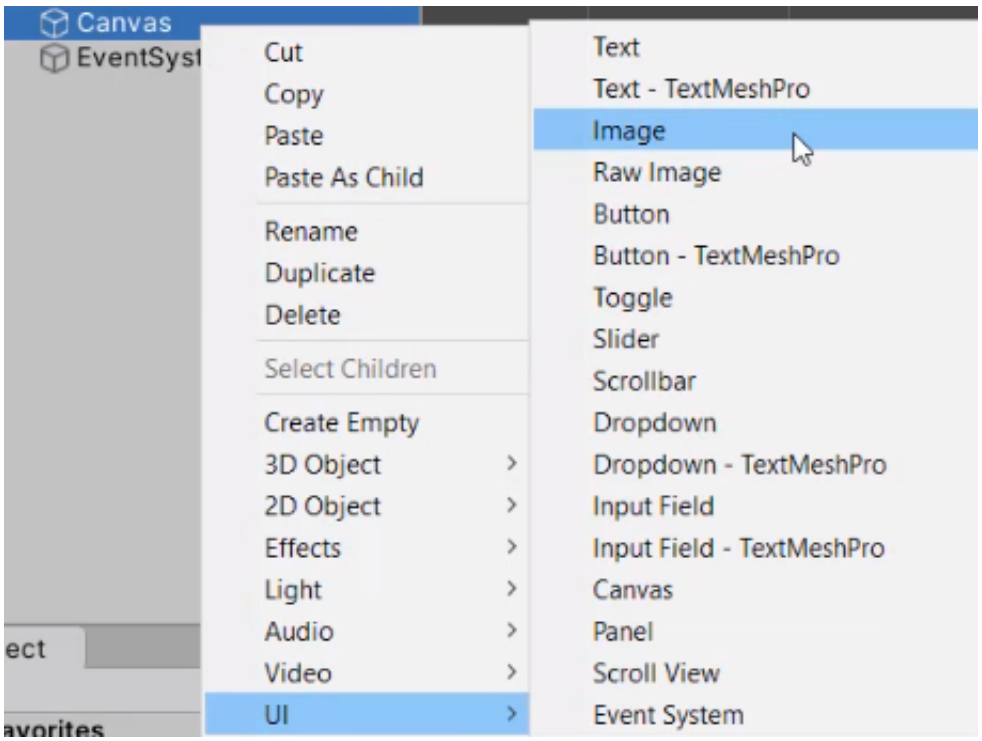


If you select the Canvas and **press F**, the screen will be zoomed out to **focus** on it.

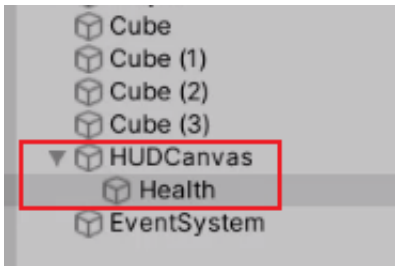


Adding Image To Canvas

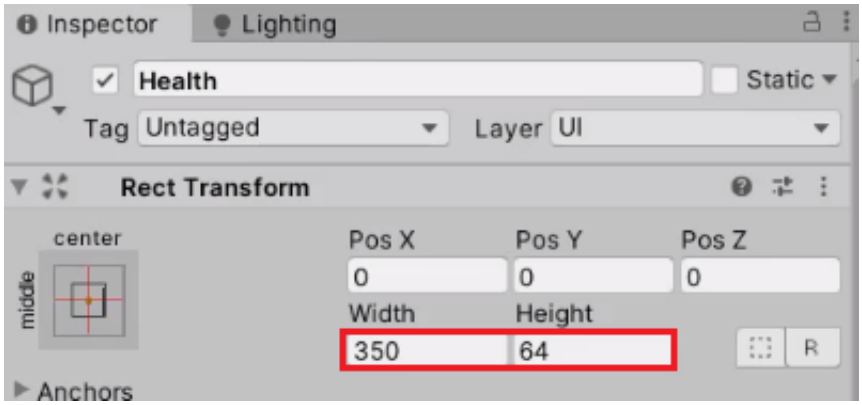
To add an **Image** object to the Canvas, we're going to **right-click** on Canvas and go to **UI > Image**.



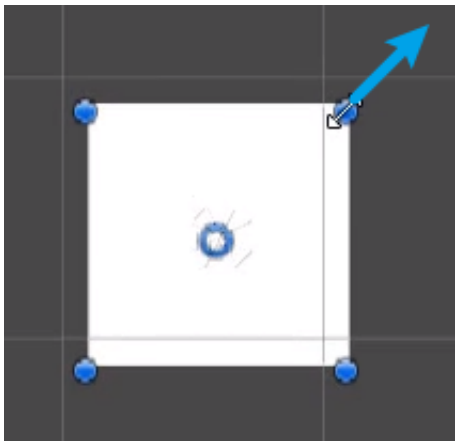
As you can see, it has created a brand new **Image** object, which is a **child** of Canvas. Let's **rename** this Image to '**Health**' and rename the canvas to '**HUDCanvas**'.



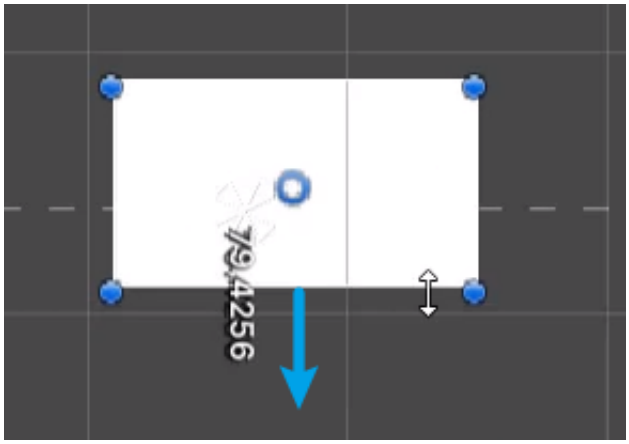
We're going to set the **Width** to 350 and **Height** to 64 in the **Rect Transform** component:



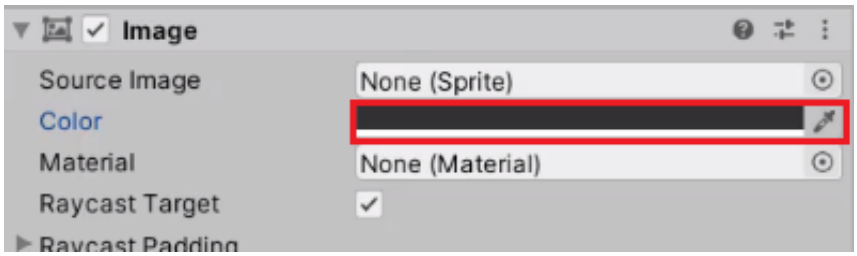
Or, you can click and drag on the **blue circles** of the health bar to increase the size.



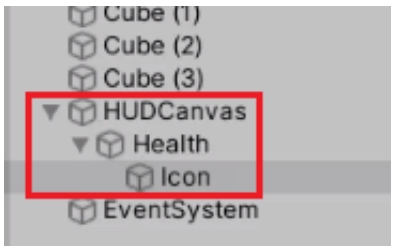
You can also click and drag on the **sides** to resize them along that specific axis.

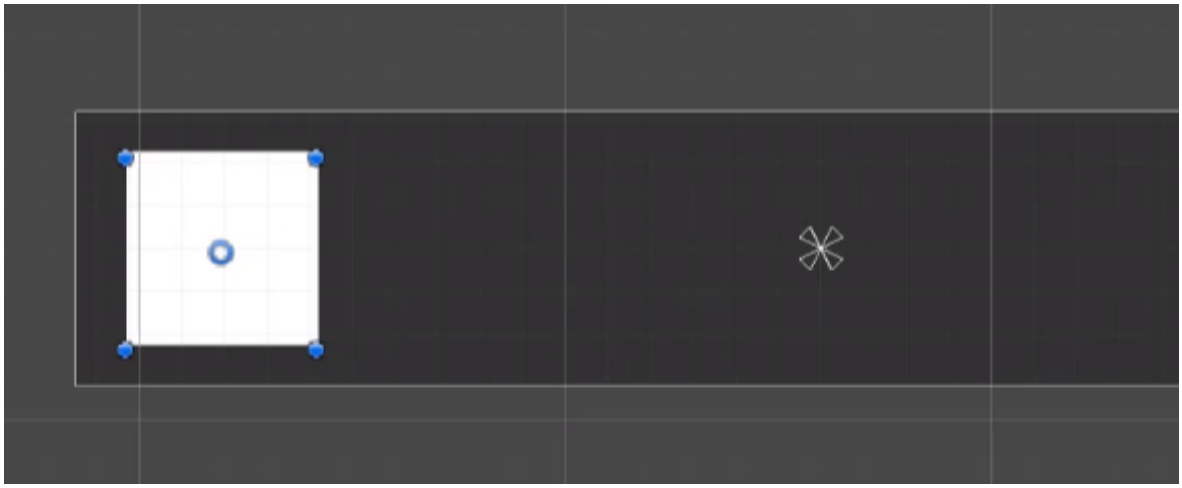


This is going to be the background for our health bar. Let's change the **Color** to be a dark gray color:

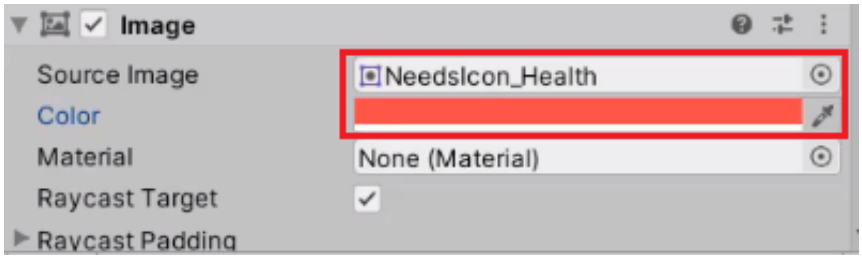


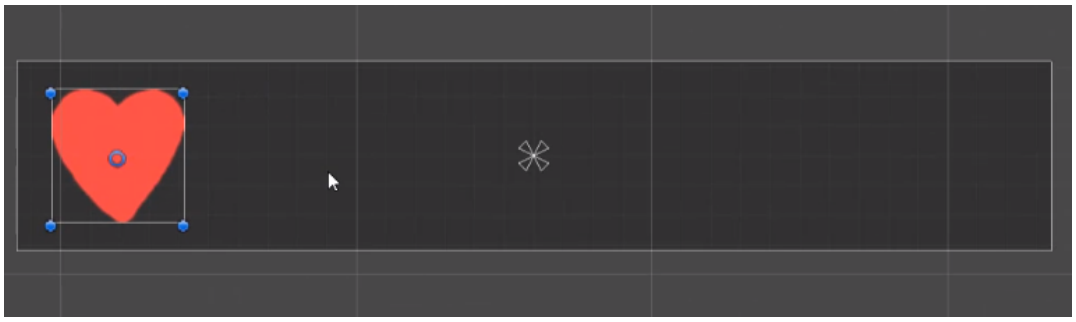
We need to add another image to the **Health** object as a child and rename it to "**Icon**." This is going to be the placeholder for our health icon.





Set the **Source Image** to be "NeedsIcon_Health", and set the **Color** to be red.





Creating A Bar

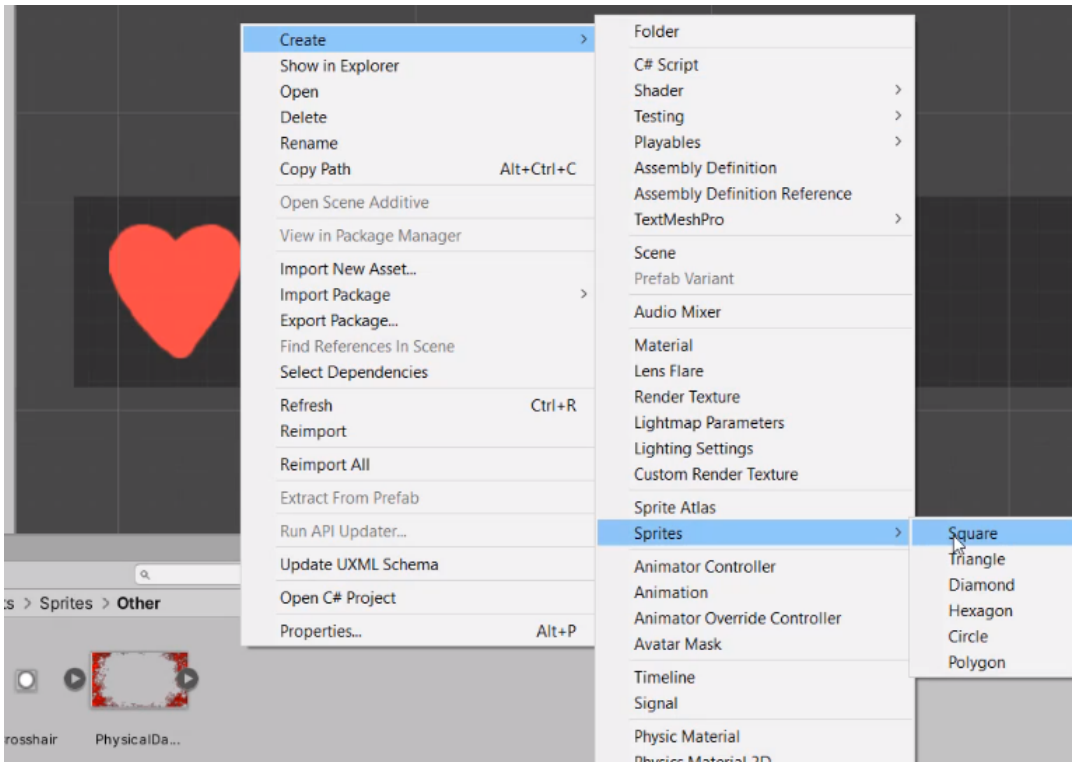
Now, we need to create an actual bar that goes up and down.

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

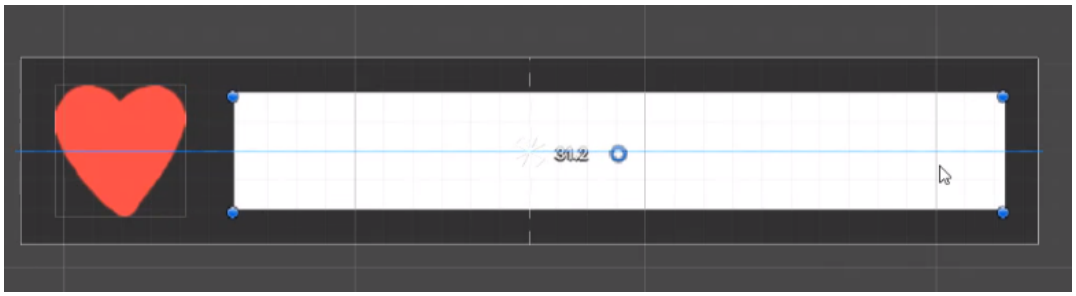
For **Unity 2020.2 and newer versions**, to create a **Square Sprite**, you will first have to install the **'2D Sprite' package** by going to **Window > Package Manager**, selecting **"Unity Registry"** from the **'Packages:' dropdown** (upper-left corner), selecting **'2D Sprite'** in the list, and finally clicking **Install** (bottom-right corner).

You can then create a new **Square Sprite** from **Create > 2D > Sprites > Square**.

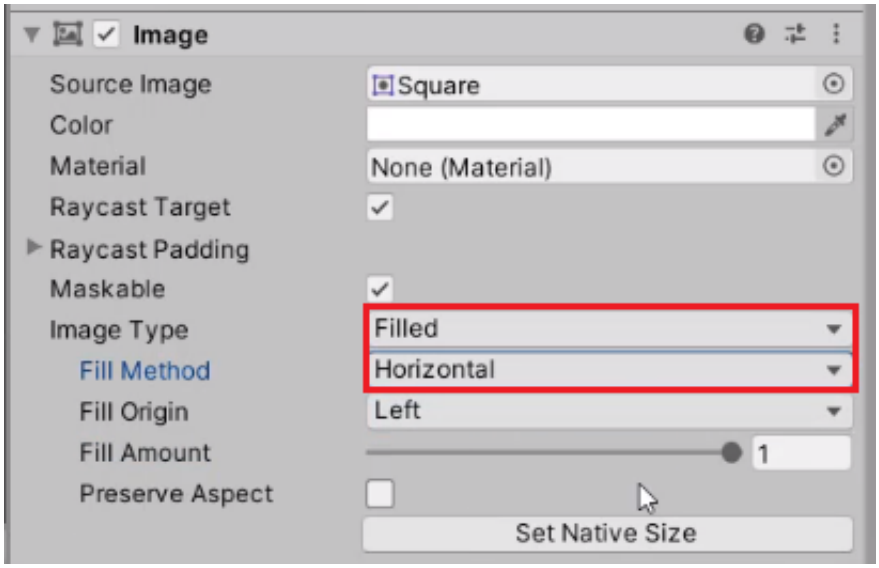
For earlier versions of Unity, let's go to **Assets > Sprites > Other** and create a new **Square Sprite**.



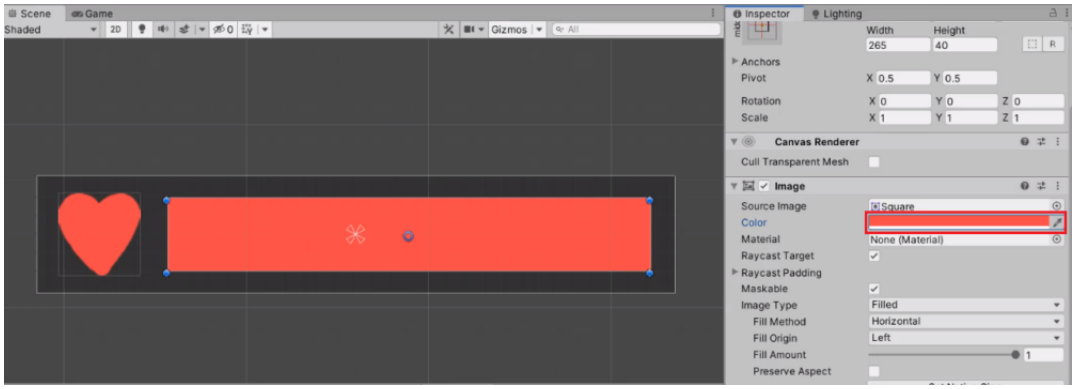
Place it in the scene as a child of the **Health** object, and set the **width** and **height** to be 265 and 40, respectively.



Make sure that the Image Type is set as **Filled**, and the fill method to be **Horizontal**.

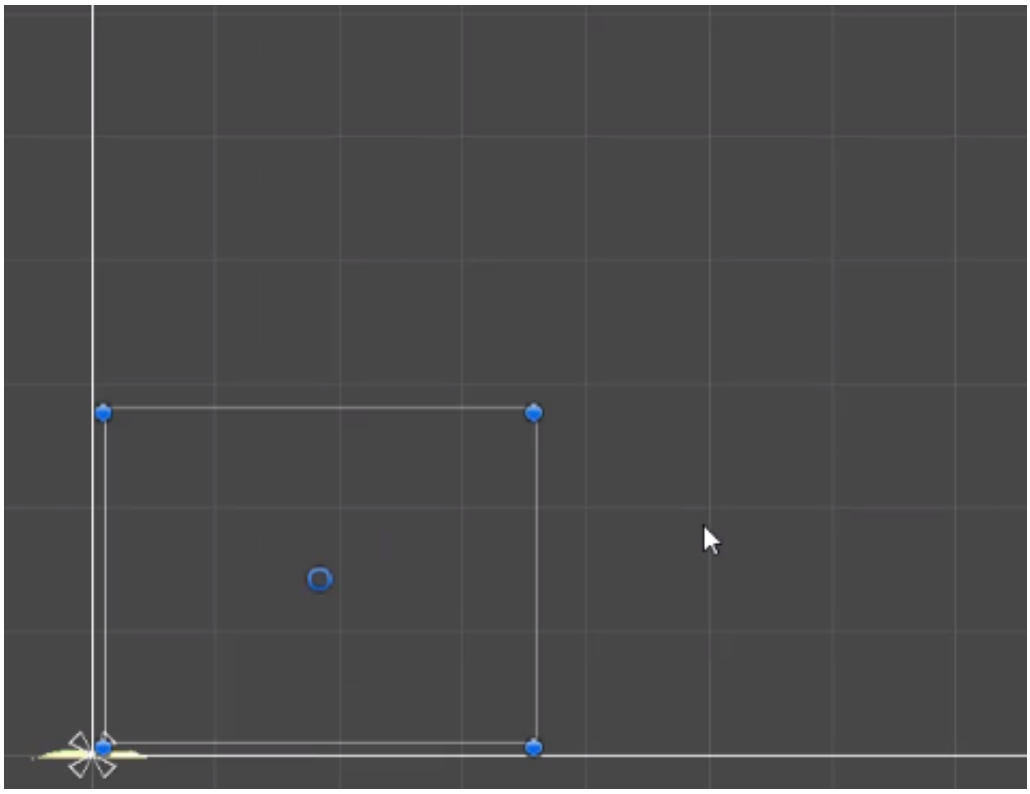


Finally, set the **Color** to be the same red as the heart icon.



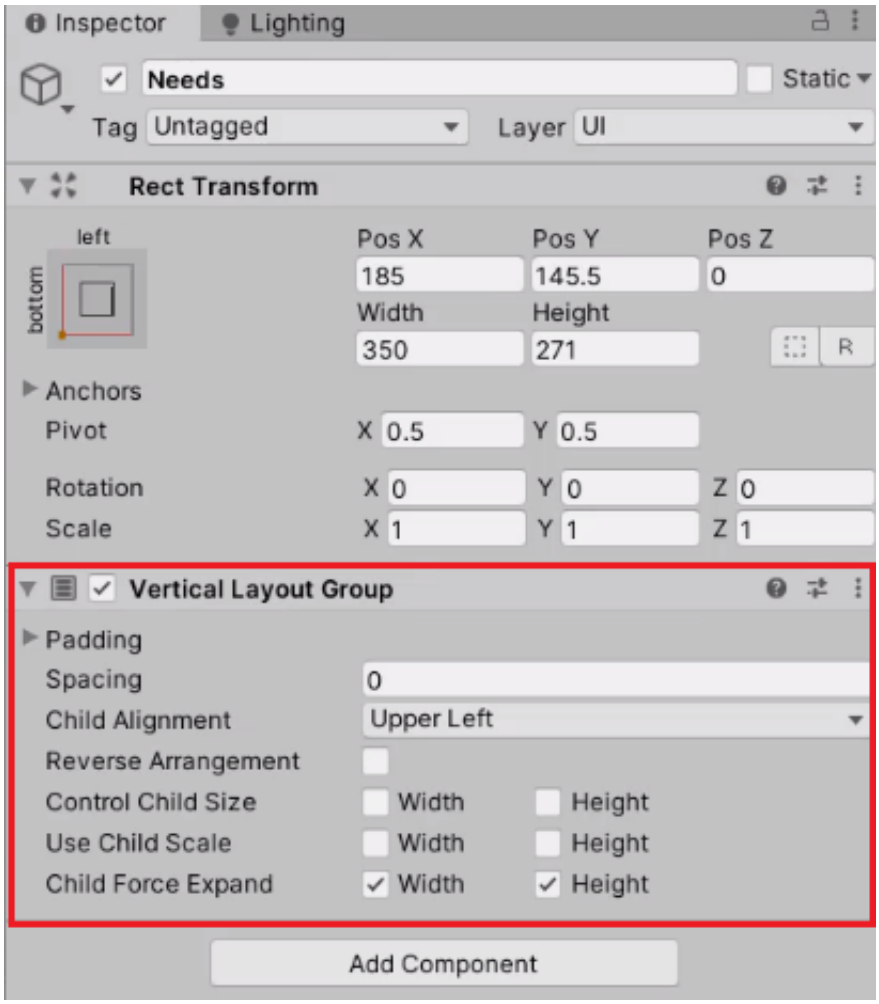
Vertical Layout Group

Now we're going to create an empty object inside Canvas, and rename it to "**Needs**". As the name suggests, this will contain all the UIs that display player's needs.

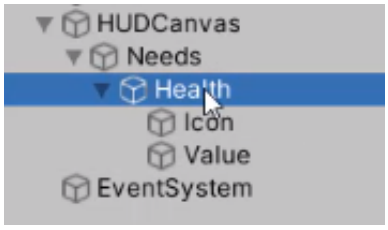


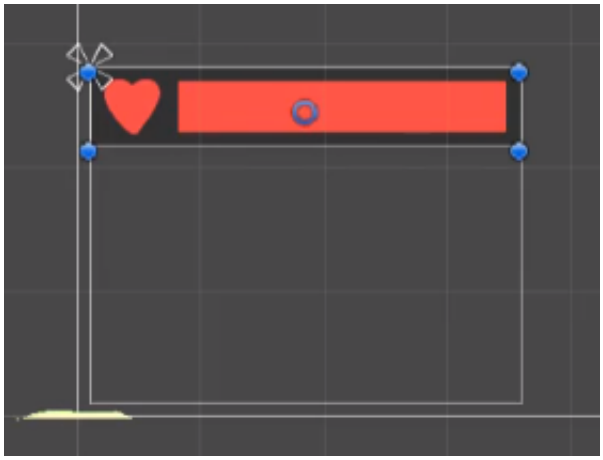
We will then click **Add Component > Vertical Layout Group**.

Vertical Layout Group is a component that automatically arranges all the children of the object that it is attached to.

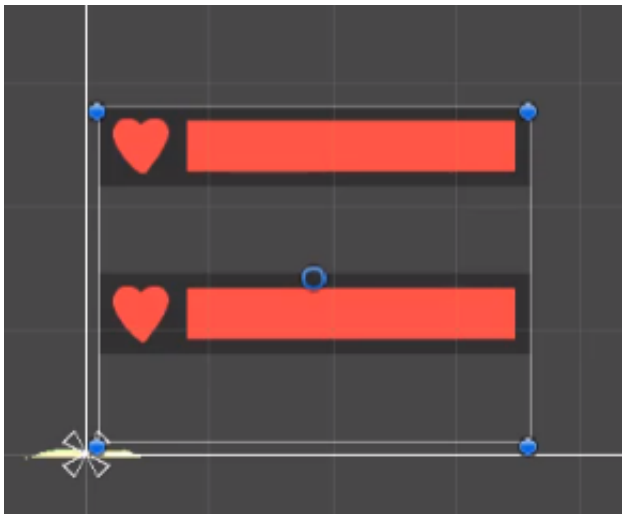


So if you drag the Health object to Needs, you'll see that it automatically gets placed at the top.

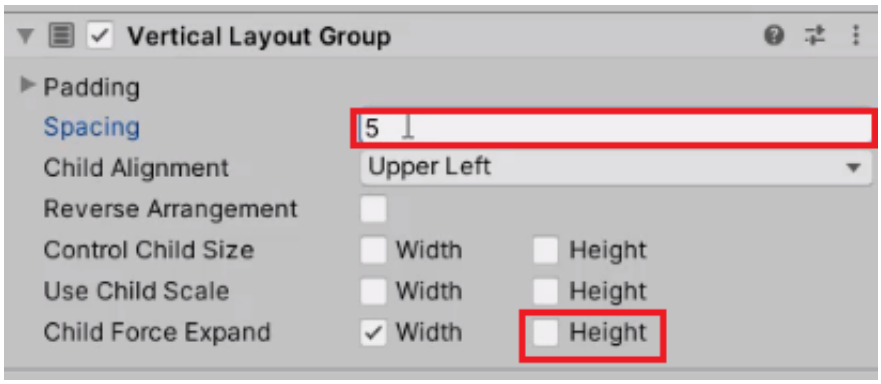


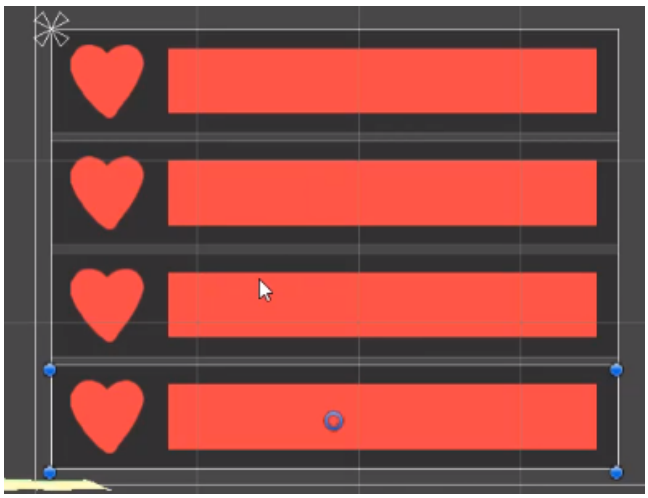


We can **duplicate** (Ctrl+D or Cmd+D) this bar, and have the copies stacked up against each other.

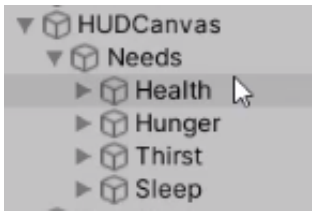


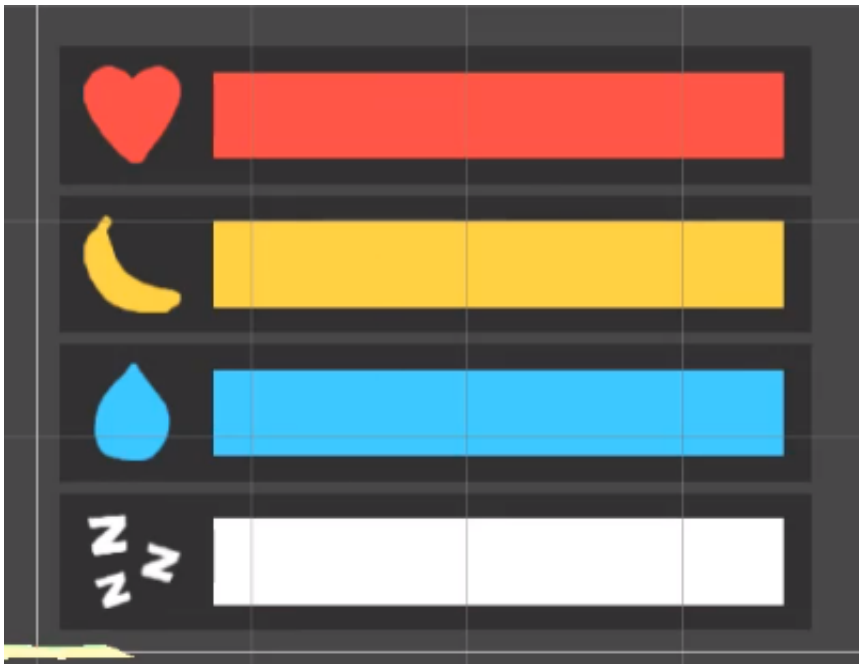
Disable the **Height** option next to **Child Force Expand** property, and manually set the **Spacing** between the copies.





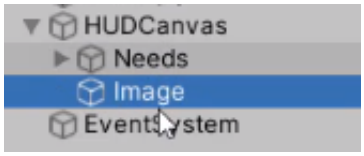
We can now set the appropriate icons, colors, and names for each need– **Health**, **Hunger**, **Thirst**, and **Sleep**.



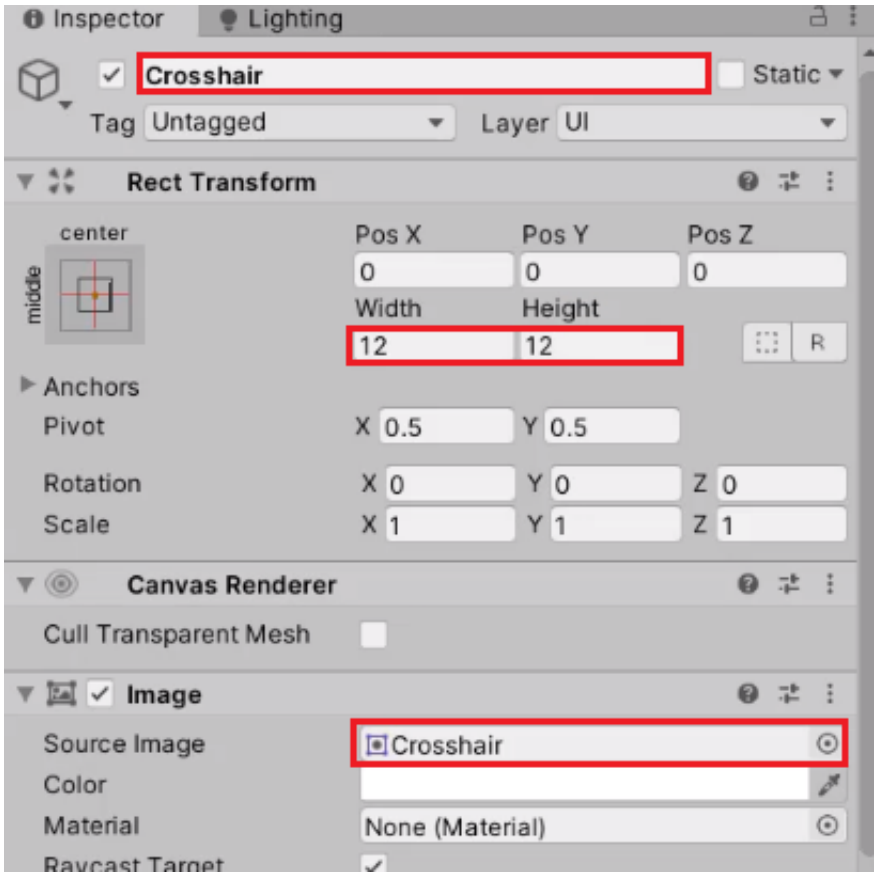


Adding Crosshair

Let’s create another **Image** below Needs, and rename it to “**Crosshair**“.

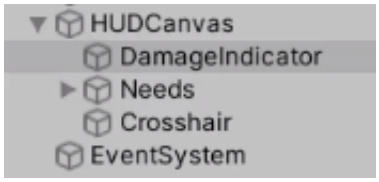


Set the **Width** and **Height** to be 12, and set the **Source Image** to “Crosshair”.



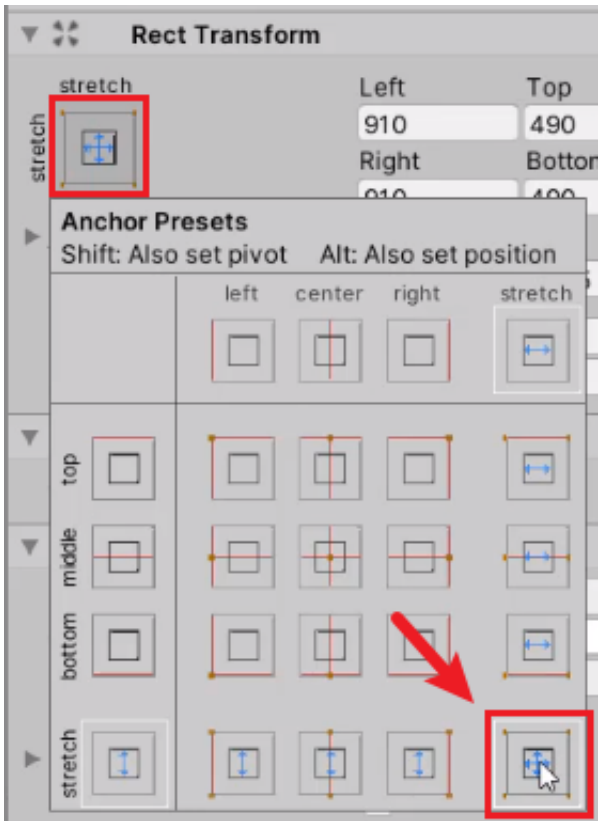
Damage Flash

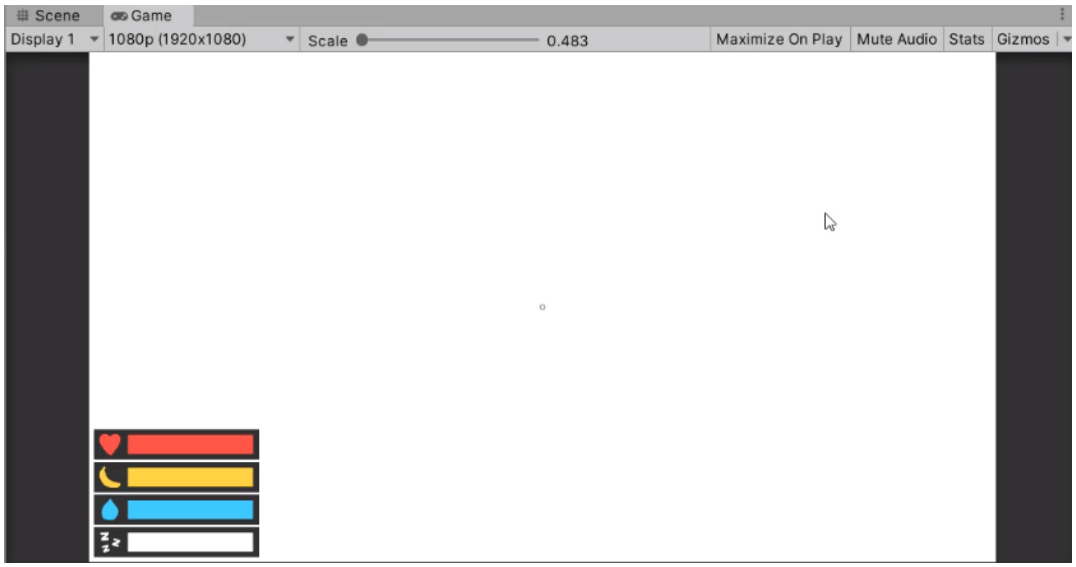
When the player gets hit by something, a flash image UI will show up on the screen to show the player they're taking damage. So we're going to create a new **Image** and rename it to "**DamageIndicator**"–



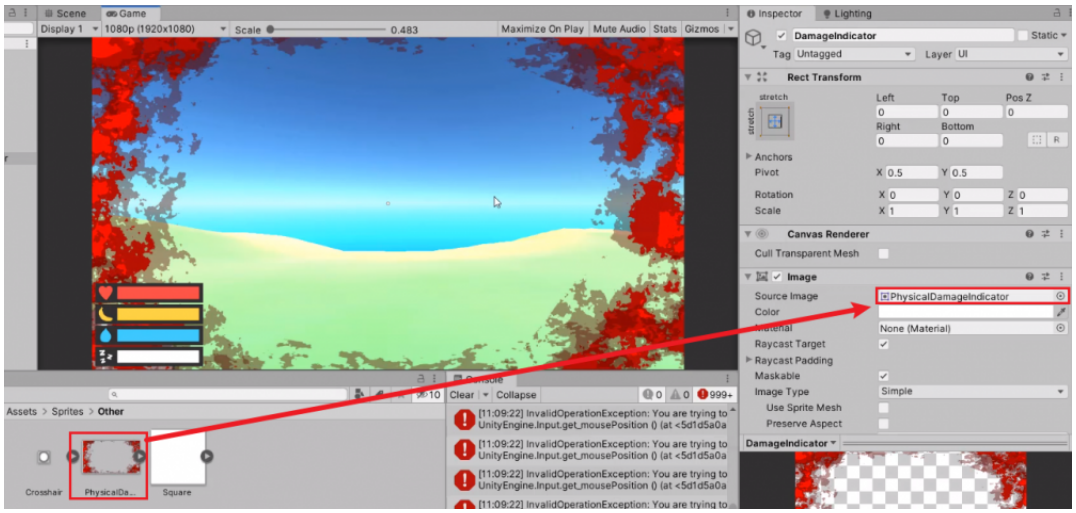
Make sure that it is placed above other UI so it gets rendered behind them.

We then need to scale it so that it is stretched along both x and y. This means that this image will fill up the entire screen, no matter the resolution or aspect ratio.



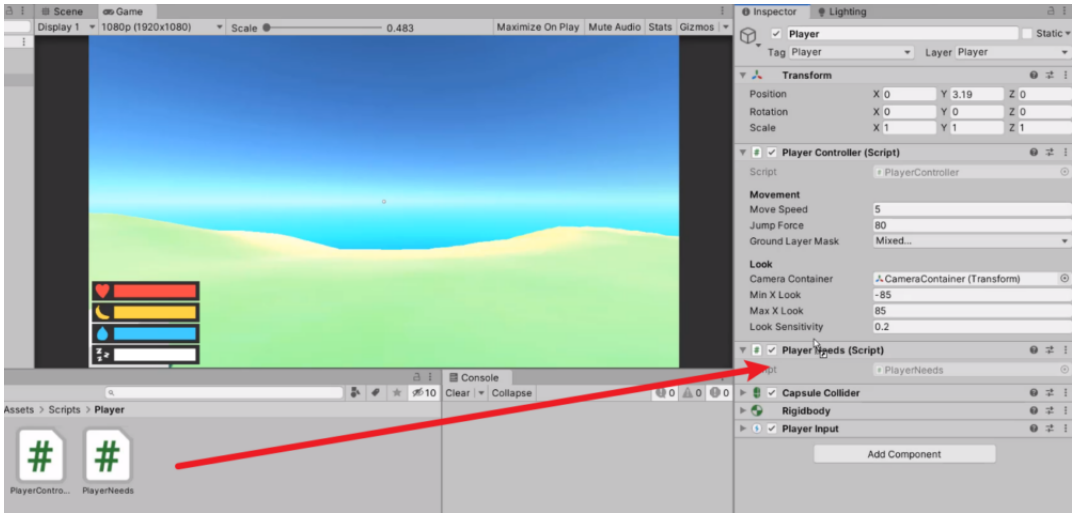


We can then set the **Source Image** to be “PhysicalDamageIndicator”, and **disable** the object to hide it as the default state.



Now that we have our UIs set up, we're going to start implementing the needs system in a new C# script called "**PlayerNeeds**".

Let's attach the script to the **Player** object, and open it up inside of Visual Studio.



The Need Class

First of all, we want to create a new data type called **Need**. In the Need class, we'll create the following variables:

```
public class Need
{
    public float curValue;
    public float maxValue;
    public float startValue;
    public float regenRate;
    public float decayRate;
    public Image uiBar;

}
```

In order to use the **Image** class, we need to declare that we're going to be **using UnityEngine.UI** namespace at the top of the script.

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

We then need to set up a function that adds some amount to the need. To ensure that we don't set the **curValue** to be above the **maxValue**, we will be using **Mathf.Min** and return either the added value or maxValue.

```
[System.Serializable]
public class Need
```



```
{
    [HideInInspector]
    public float curValue;
    public float maxValue;
    public float startValue;
    public float regenRate;
    public float decayRate;
    public Image uiBar;

    // add to the need
    public void Add (float amount)
    {
        curValue = Mathf.Min(curValue + amount, maxValue);
    }
}
```

Similarly, we need to set up a function that subtracts a value from the need. Note that we're using **Mathf.Max** this time as we want the subtracted value to be greater than zero.

```
// subtract from the need
public void Subtract (float amount)
{
    curValue = Mathf.Max(curValue - amount, 0.0f);
}
```

The needs should be displayed in the format of (**curValue/maxValue**), which is between zero and one. When the current value is half of the max value, it is going to return 0.5.

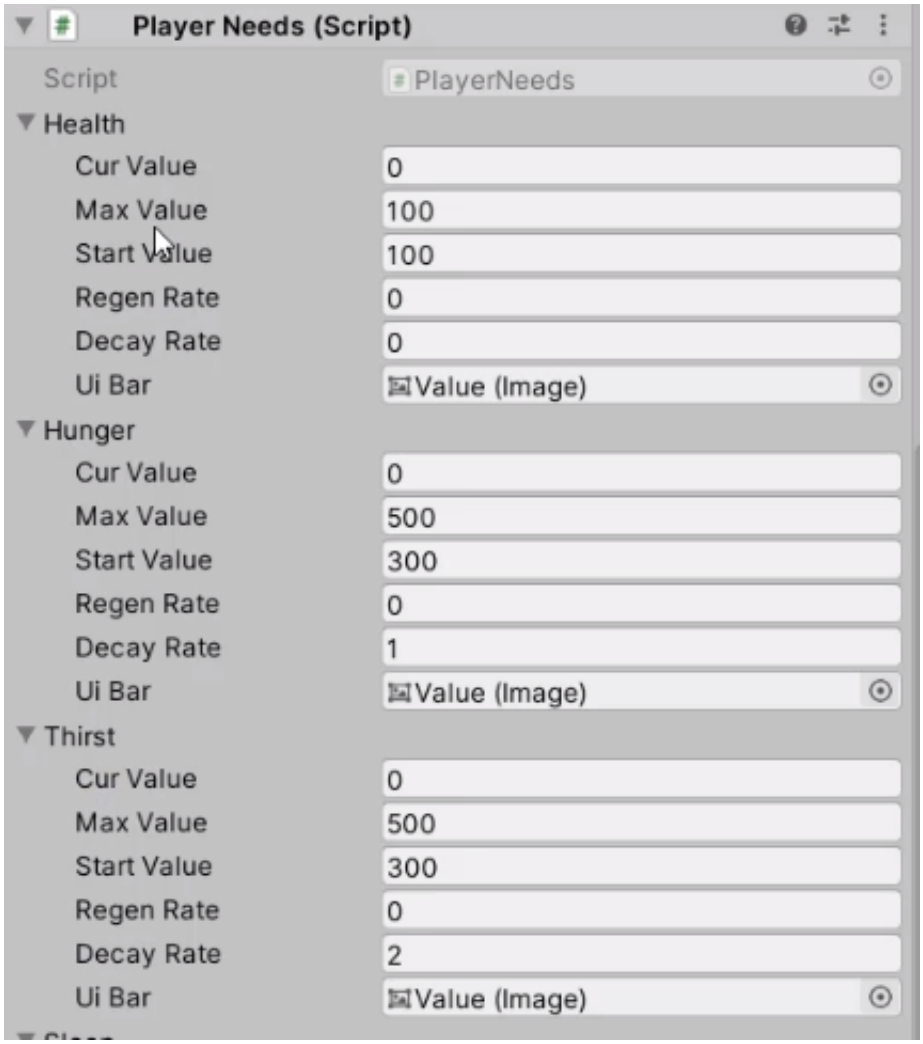
```
// return the percentage value (0.0 - 1.0)
public float GetPercentage ()
{
    return curValue / maxValue;
}
```

Finally, we will make the **Need** class serializable, so we can assign values to the fields in the Inspector.

```
[System.Serializable]
public class Need
{
    [HideInInspector]
    public float curValue;
    public float maxValue;
    public float startValue;
    public float regenRate;
    public float decayRate;
    public Image uiBar;
```

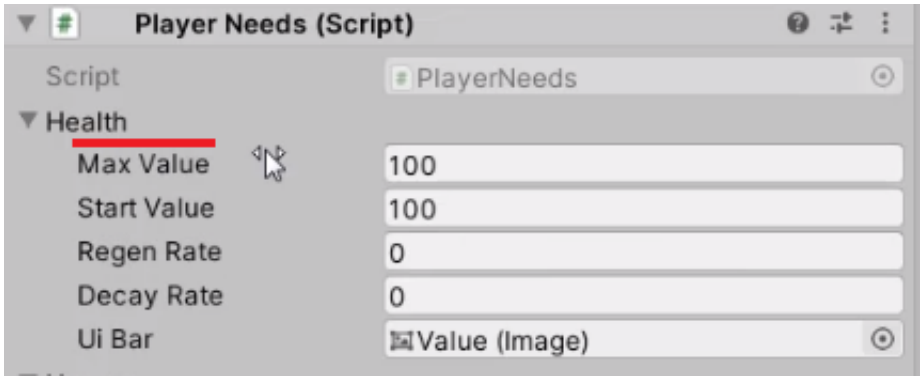


```
    // add to the need
    public void Add (float amount)
    {
        curValue = Mathf.Min(curValue + amount, maxValue);
    }

    // subtract from the need
    public void Subtract (float amount)
    {
        curValue = Mathf.Max(curValue - amount, 0.0f);
    }

    // return the percentage value (0.0 - 1.0)
    public float GetPercentage ()
    {
        return curValue / maxValue;
    }
}
```



Notice how we don't actually need to modify the **curValue** property in the Inspector. We can make it invisible by using the **[HideInInspector]** attribute.


```
[System.Serializable]
public class Need
{
    [HideInInspector]
    public float curValue;
    public float maxValue;
    public float startValue;
    public float regenRate;
    public float decayRate;
    public Image uiBar;
    ...
}
```



Creating Needs

Using the need class that we just created, we can create our four needs: **Health**, **Hunger**, **Thirst,** and **Sleep**.

```
public class PlayerNeeds : MonoBehaviour
{
    public Need health;
    public Need hunger;
    public Need thirst;
    public Need sleep;
    ...
}
```


All these needs should be set with the default **curValues** at the start of the game. So inside the start function, we will add the following lines:

```
void Start ()
{
    // set the start values
    health.curValue = health.startValue;
    hunger.curValue = hunger.startValue;
    thirst.curValue = thirst.startValue;
    sleep.curValue = sleep.startValue;
}
```

Along with that, we want to have variables for indicating how much our needs are going to **decay** if we don’t meet the needs criteria.

```
public class PlayerNeeds : MonoBehaviour, IDamagable
{
    public Need health;
    public Need hunger;
    public Need thirst;
    public Need sleep;

    public float noHungerHealthDecay;
    public float noThirstHealthDecay;
    ...
}
```


Decaying Needs Over Time

We have assigned the start values for our needs, but how about decaying them over time? Let's start with subtracting the hunger in the Update function.

For each frame, we're going to subtract the hunger and thirst's decay rate, multiplied by **Time.deltaTime**. Time.deltaTime is used here to convert the rate from **per frame** to **per second**.

```
void Update ()
{
    // decay needs over time
    hunger.Subtract(hunger.decayRate * Time.deltaTime);
    thirst.Subtract(thirst.decayRate * Time.deltaTime);
}
```

We also want to regenerate sleep using the same method:

```
void Update ()
{
    // decay needs over time
    hunger.Subtract(hunger.decayRate * Time.deltaTime);
    thirst.Subtract(thirst.decayRate * Time.deltaTime);
    sleep.Add(sleep.regenRate * Time.deltaTime);
}
```

Updating UI bars

The **Fill Amout** property of **UI.Image** is the number that ranges from zero to one, which decides how much of the bar image is being rendered. We're going to be using the **GetPercentage** method to update the full amount for each need.

```
void Update ()
{
    // decay needs over time
    hunger.Subtract(hunger.decayRate * Time.deltaTime);
    thirst.Subtract(thirst.decayRate * Time.deltaTime);
    sleep.Add(sleep.regenRate * Time.deltaTime);

    // update UI bars
    health.uiBar.fillAmount = health.GetPercentage();
    hunger.uiBar.fillAmount = hunger.GetPercentage();
    thirst.uiBar.fillAmount = thirst.GetPercentage();
    sleep.uiBar.fillAmount = sleep.GetPercentage();
}
```

Just under where we decay our needs over time, we also want to implement the ability for our health to subtract over time if our hunger or thirst equals zero.

```
void Update ()
```



```
{
    // decay needs over time
    hunger.Subtract(hunger.decayRate * Time.deltaTime);
    thirst.Subtract(thirst.decayRate * Time.deltaTime);
    sleep.Add(sleep.regenRate * Time.deltaTime);

    // decay health over time if no hunger or thirst
    if(hunger.curValue == 0.0f)
        health.Subtract(noHungerHealthDecay * Time.deltaTime);
    if(thirst.curValue == 0.0f)
        health.Subtract(noThirstHealthDecay * Time.deltaTime);

    // update UI bars
    health.uiBar.fillAmount = health.GetPercentage();
    hunger.uiBar.fillAmount = hunger.GetPercentage();
    thirst.uiBar.fillAmount = thirst.GetPercentage();
    sleep.uiBar.fillAmount = sleep.GetPercentage();
}
```

Player Actions

Now that our player's basic needs UI are set up, we're going to start working on various player actions. These are going to be called **Heal**, **Eat**, **Drink**, and **Sleep**.

Each function takes in a **float** variable for the **amount** to add to or subtract from the player's health, hunger, thirst, or sleep.

```
// adds to the player's HEALTH
public void Heal (float amount)
{
    health.Add(amount);
}

// adds to the player's HUNGER
public void Eat (float amount)
{
    hunger.Add(amount);
}

// adds to the player's THIRST
public void Drink (float amount)
{
    thirst.Add(amount);
}

// subtracts from the player's SLEEP
public void Sleep (float amount)
{
    sleep.Subtract(amount);
}
```


Similarly, we also want two separate functions to be called when the player **Takes Physical Damage** and **Die**. For now, we're just going to display a console log when the player is dead.

```
// called when the player takes physical damage (fire, enemy, etc)
public void TakePhysicalDamage (int amount)
{
    health.Subtract(amount);
}

// called when the player's health reaches 0
public void Die ()
{
    Debug.Log("Player is dead");
}
```

Inside the Update function, we will call the **Die** function if the player's health is zero.

```
void Update ()
{
    // decay needs over time
    hunger.Subtract(hunger.decayRate * Time.deltaTime);
    thirst.Subtract(thirst.decayRate * Time.deltaTime);
    sleep.Add(sleep.regenRate * Time.deltaTime);

    // decay health over time if no hunger or thirst
    if(hunger.curValue == 0.0f)
        health.Subtract(noHungerHealthDecay * Time.deltaTime);
    if(thirst.curValue == 0.0f)
        health.Subtract(noThirstHealthDecay * Time.deltaTime);

    // check if player is dead
    if(health.curValue == 0.0f)
    {
        Die();
    }

    // update UI bars
    health.uiBar.fillAmount = health.GetPercentage();
    hunger.uiBar.fillAmount = hunger.GetPercentage();
    thirst.uiBar.fillAmount = thirst.GetPercentage();
    sleep.uiBar.fillAmount = sleep.GetPercentage();
}
```

One thing to add to the **TakePhysicalDamage** function is the ability to invoke **Unity Event**. So at the top of the script, we're going to add this new line:

```
using UnityEngine.Events;
```

Then we need to create a new **UnityEvent** variable called **OnTakeDamage**.


```
public class PlayerNeeds : MonoBehaviour, IDamagable
{
    public Need health;
    public Need hunger;
    public Need thirst;
    public Need sleep;

    public float noHungerHealthDecay;
    public float noThirstHealthDecay;

    public UnityEvent onTakeDamage;
    ...
}
```

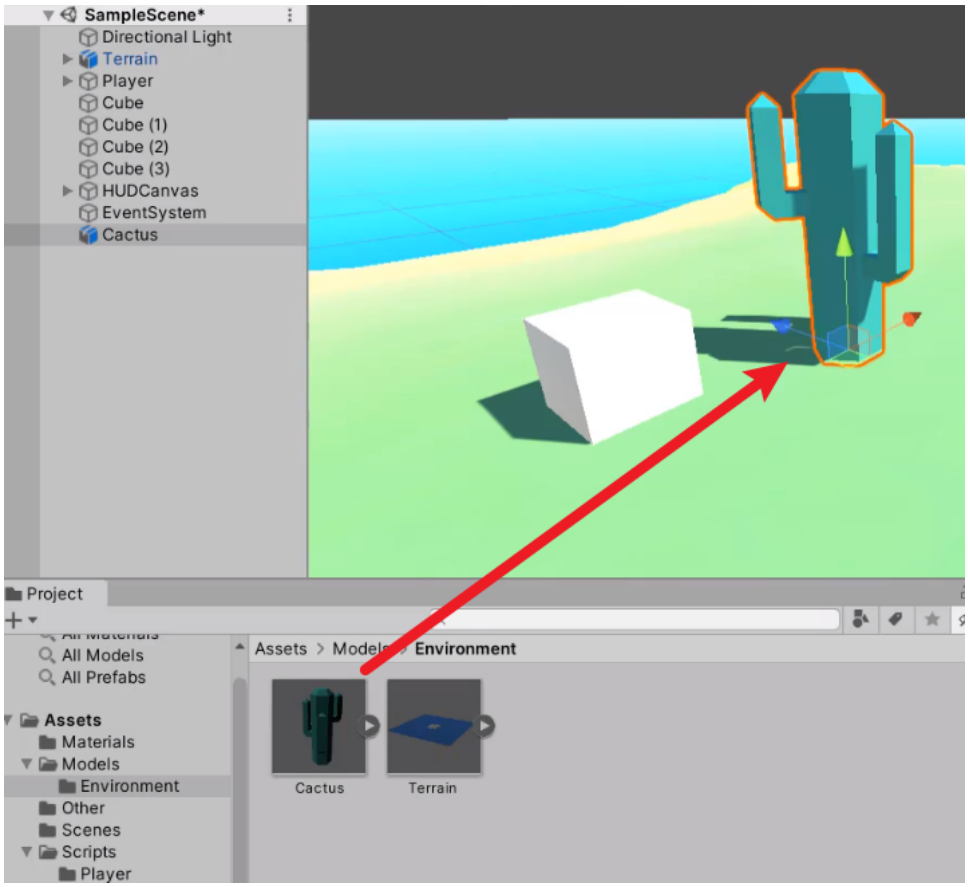
Whenever the player takes physical damage (fire, enemy, etc.), the **TakePhysicalDamage** function is called and subtracts the given amount from health. In addition, we're going to invoke **OnTakeDamage** event here:

```
// called when the player takes physical damage (fire, enemy, etc)
public void TakePhysicalDamage (int amount)
{
    health.Subtract(amount);
    onTakeDamage?.Invoke();
}
```

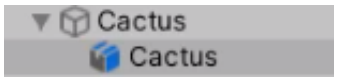
You may be wondering why there is a question mark next to **onTakeDamage**. Normally, if we have nothing subscribed to the **onTakeDamage** event, it will cause an error. So the question mark is placed to check if OnTakeDamage exists first, and invoke the event only if it exists.

Creating A Cactus

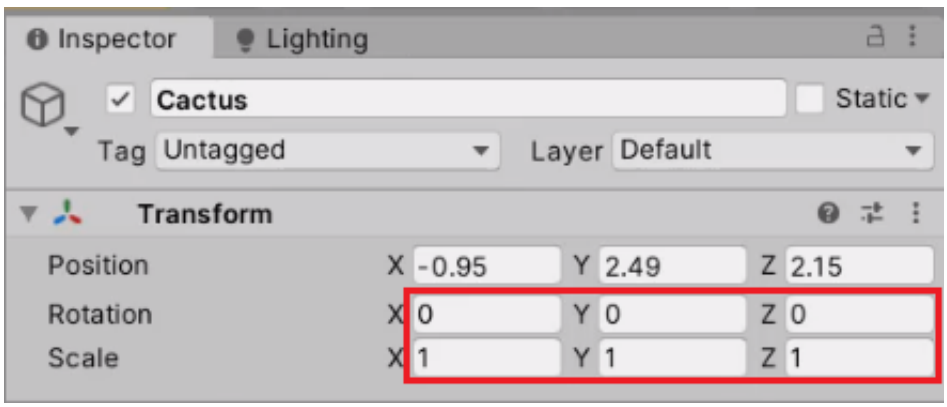
Let's open up **Assets > Models > Environment** and drag the **Cactus** object into the scene. The default size is pretty small, so we scaled it up to (4, 4, 4).



To store this model, let's also create an **empty** GameObject called "**Cactus**" and make it parent to the cactus model.

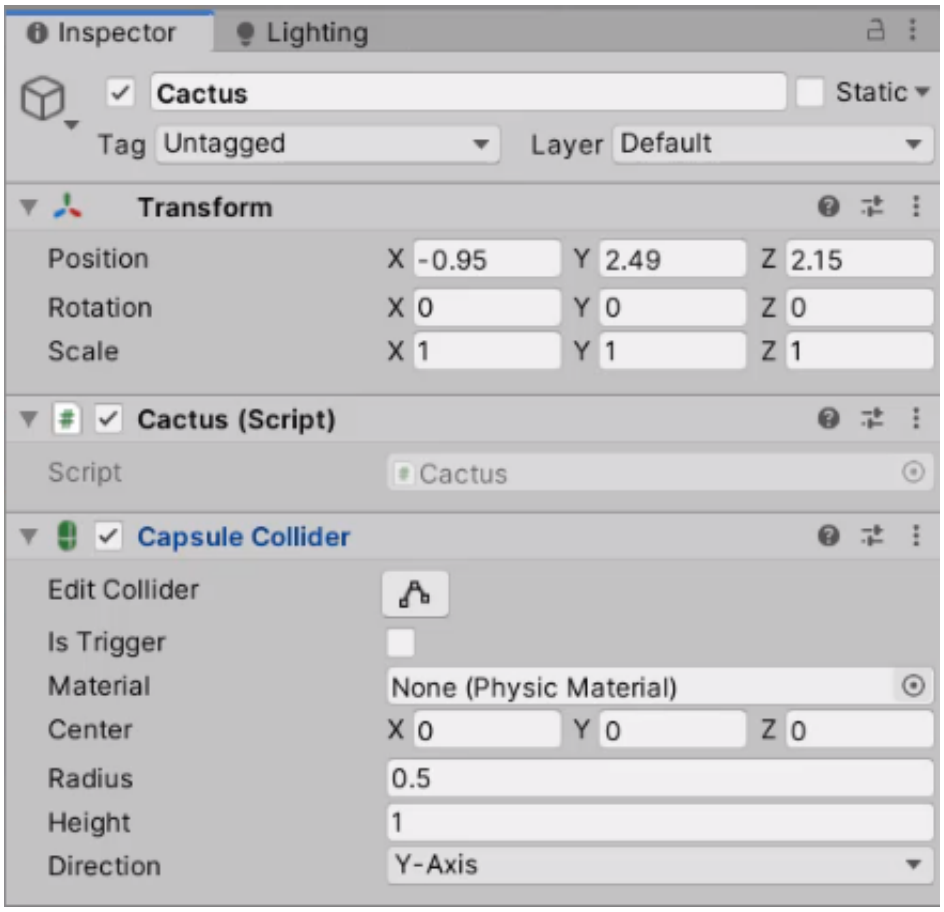


By doing this we can keep the **rotation** and **scale** of the root object consistent, even if we're rotating or scaling the model. This is generally good practice when creating objects in Unity.

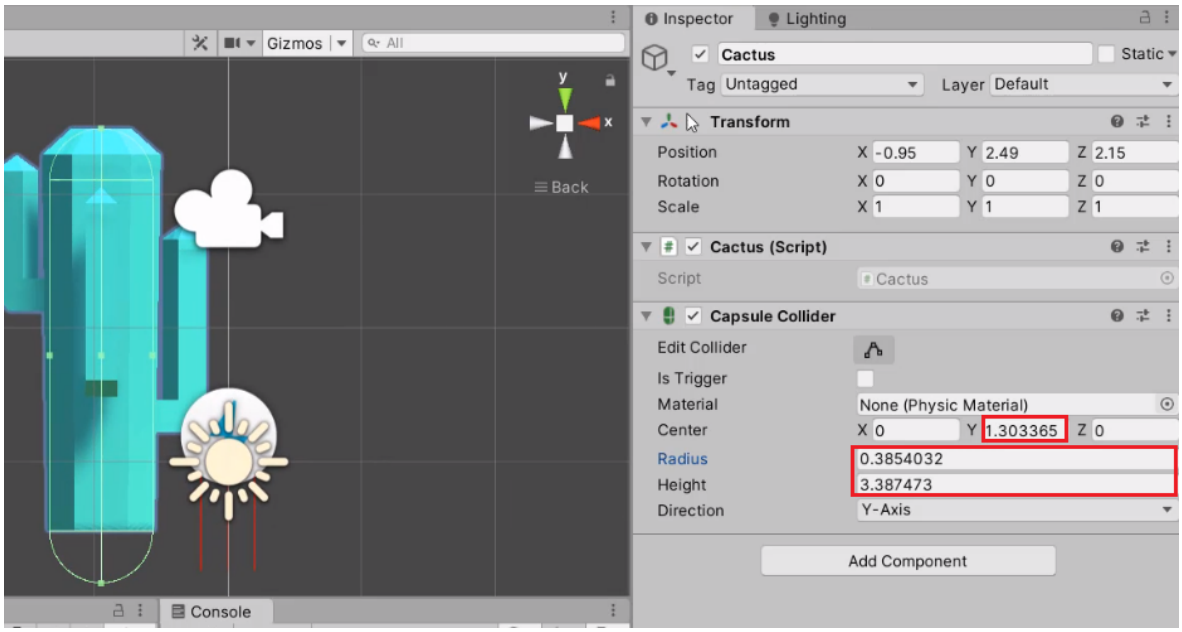


When the player hits this cactus, they're going to start taking damage. To do this, we need to attach

a **Capsule Collider** and a new c# **Script** called “Cactus” to this object.



Make sure to adjust the **Radius, y-Center,** and **Height** of the collider to fit the model correctly.



Creating An Interface

An **interface** contains the definitions of functions or variables that we need in a certain script. Let’s

open up the **PlayerNeeds** script and create a new interface called “IDamagable”.

```
public interface IDamagable
{

}
```

Basically, every object that can be damaged (i.e. Player and Enemy) is going to have this interface. If an object that has the IDamagable interface collides with a cactus, then we will call the **TakePhysicalDamage** function and inflict physical damage.

```
public interface IDamagable
{
    void TakePhysicalDamage(int damageAmount);
}
```

In order for other classes (e.g. Cactus) to access this interface, we need to scroll up to the top of the script and add **IDamagable** next to MonoBehaviour.

```
public class PlayerNeeds : MonoBehaviour, IDamagable
{

...

}
```

Scripting Cactus

Let’s open up the Cactus script and create the following variables:

```
public class Cactus : MonoBehaviour
{
    public int damage;
    public float damageRate;

    private List<IDamagable> thingsToDamage = new List<IDamagable>();
}
```

For every single **damage rate**, we want to deal damage to everything inside the **things to damage** list. So for that, we’re going to create an **IEnumerator** called “DealDamage”.

```
IEnumerator DealDamage ()
{
    // every "damageRate" seconds, damage all thingsToDamage
    while(true)
    {
        yield return new WaitForSeconds(damageRate);
```



```
    }
}
```

IEnumerator is a **Coroutine**, which is a type of function that can be paused for a certain amount of time. For more information, refer to https://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html

At the moment, all it does is pausing for **damageRate** seconds in a **while** loop. There is no exit condition to the loop, so as long as our program is open and this object is active, the while loop is going to continue to run.

Now we want to add a new for loop here, so that every time this while loop runs, we will be looping through each element in **thingsToDamage** list, calling the **TakePhysicalDamage** function.

```
IEnumerator DealDamage ()
{
    // every "damageRate" seconds, damage all thingsToDamage
    while(true)
    {
        for(int i = 0; i < thingsToDamage.Count; i++)
        {
            thingsToDamage[i].TakePhysicalDamage(damage);
        }

        yield return new WaitForSeconds(damageRate);
    }
}
```


In the **DealDamage** IEnumerator, we are damaging everything inside of the **thingsToDamage** list every **damageRate** seconds.

IDamagable Collision Detection

Now what we need to do is adding elements to the list whenever any objects with the IDamagable interface touch the cactus. So when the player or the enemy is touching the cactus, they're going to take damage every single **damageRate** seconds. This can be done inside the **OnCollisionEnter** function:

```
// called when an object collides with the cactus
private void OnCollisionEnter (Collision collision)
{
    // if it's an IDamagable, add it to the list
    if(collision.gameObject.GetComponent<IDamagable>() != null)
    {
        thingsToDamage.Add(collision.gameObject.GetComponent<IDamagable>());
    }
}
```

And similar to adding to the list, we can remove elements from the list when the objects leave the cactus, using **OnCollisionExit** function:

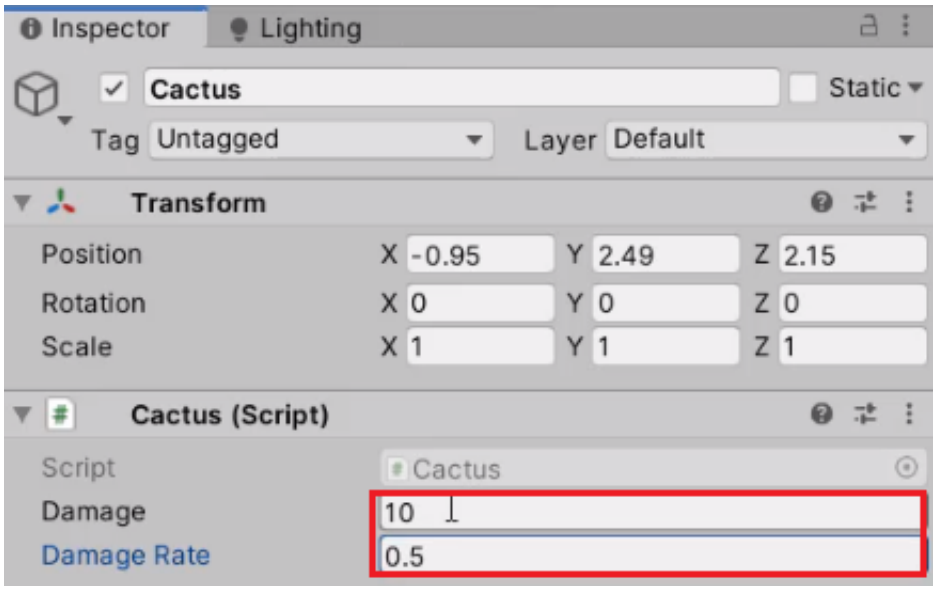
```
// called when an object stops colliding with the cactus
private void OnCollisionExit (Collision collision)
{
    // if it's an IDamagable, remove it from the list
    if(collision.gameObject.GetComponent<IDamagable>() != null)
    {
        thingsToDamage.Remove(collision.gameObject.GetComponent<IDamagable>());
    }
}
```

Finally, we can start the **DealDamage** coroutine inside the **Start** function.

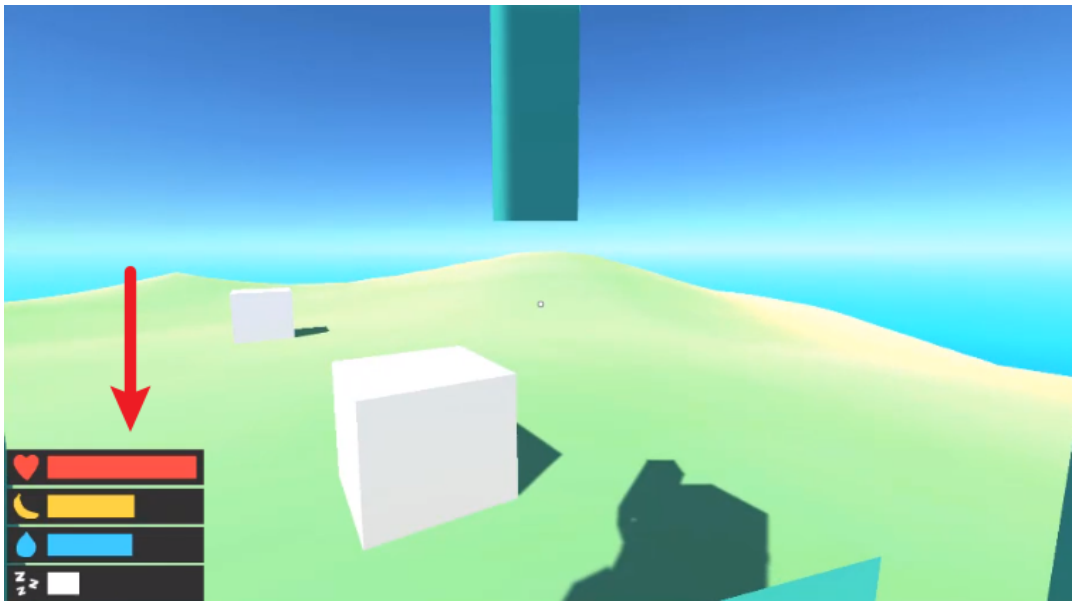
```
void Start ()
{
    StartCoroutine(DealDamage());
}
```

Testing It Out

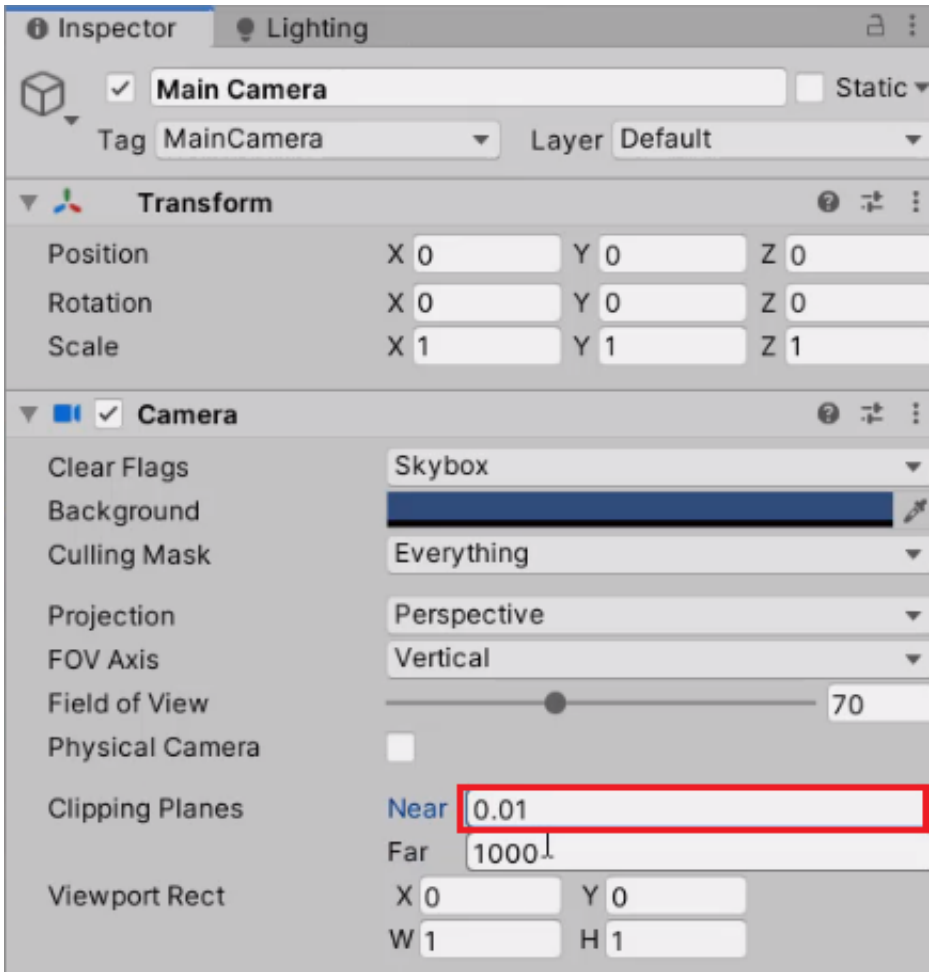
Let's save the script and assign values to the public variables of Cactus.



Now you should be able to run into the cactus and see the health bar getting reduced.



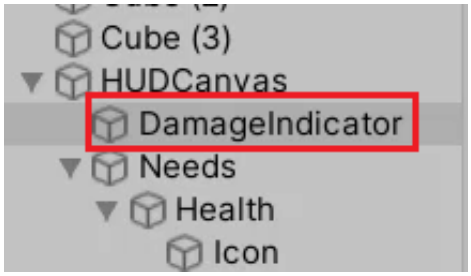
You may notice that some part of the cactus gets clipped when we get close to it. This is because the default distance of the **near-clipping plane** is 0.3. This means the camera cannot see geometry that is closer than the distance.



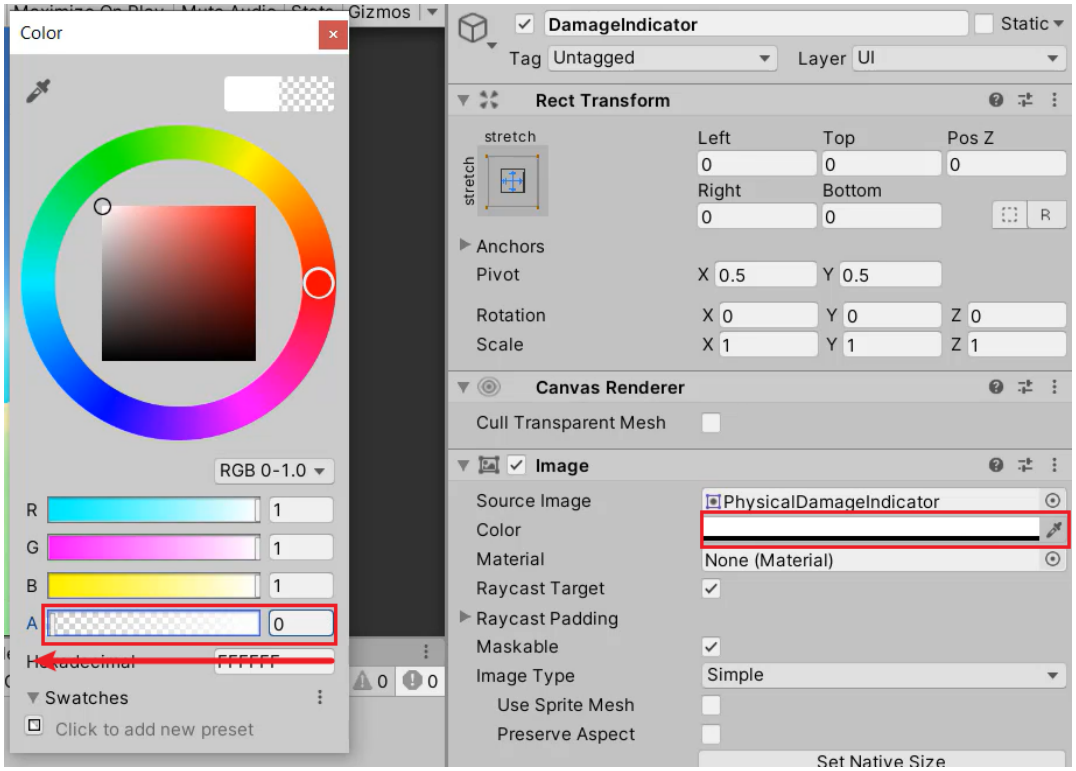
To fix this, we can change the **Near Clipping Plane** of our Main Camera to a very small number, like 0.01.

In this lesson, we're going to continue on setting up our **Damage Indicator** to flash on screen when the player takes damage.

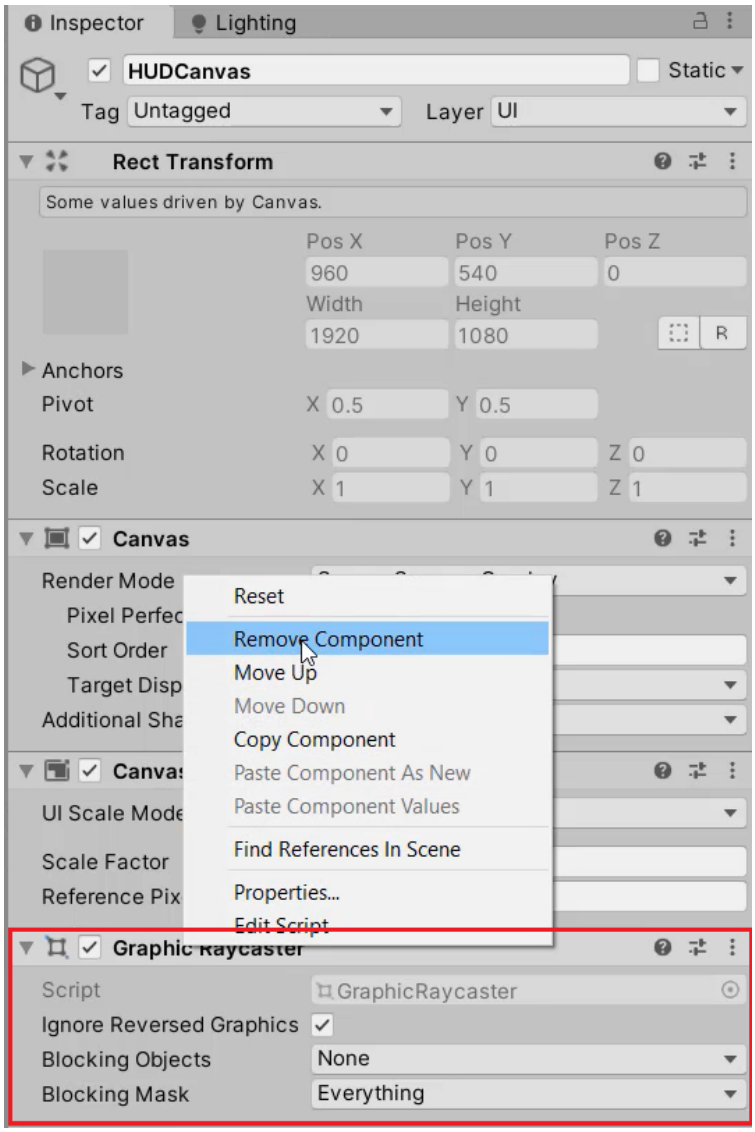
To begin, let's enable the **DamageIndicator** gameObject.



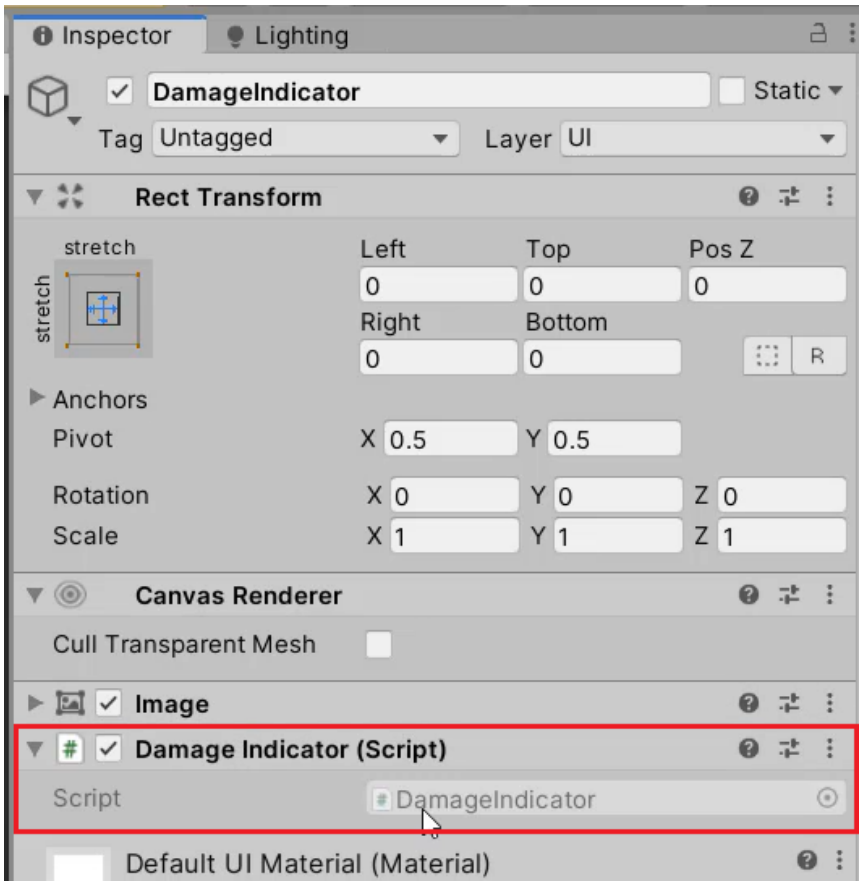
Then, select the **Color** property and change the **Alpha** down to zero to make it invisible for now.



Next, we're going to remove the **Graphic Raycaster** component on **HUDCanvas** to disable the ability to detect mouse clicks and improve performance.



We now need to create a new C# script and attach it to the damage indicator object.



Creating Variables

Inside the script, we need to create the following variables:

```
public class DamageIndicator : MonoBehaviour
{
    public Image image;
    public float flashSpeed;

    private Coroutine fadeAway;
}
```

We also need to import **Unity.UI** library to access the **Image** component.

```
using UnityEngine.UI;
```

Flashing Damage Indicator

To get the Damage Indicator flash, we'll create a new function called **Flash**. In here, we will check to see if we're currently fading the image away.

```
 // called when the player takes damage - listening to the OnTakeDamage event
public void Flash ()
```



```
{
    // stop all currently running FadeAway coroutines
    if(fadeAway != null)
        StopCoroutine(fadeAway);
}
```

After this, we can enable the image, and set the image color to **Color.White**. Note that this doesn't mean the image is going to appear white, rather it will set the tint to white, increasing the alpha value up to 1.

```
// called when the player takes damage - listening to the OnTakeDamage event
public void Flash ()
{
    // stop all currently running FadeAway coroutines
    if(fadeAway != null)
        StopCoroutine(fadeAway);

    // reset the image
    image.enabled = true;
    image.color = Color.white;
}
```

Let's create an **IEnumerator** called **FadeAway**. The role of this enumerator is to decrease the alpha over time and disable the image when we're not needing it.

```
// fades the image away over time
IEnumerator FadeAway ()
{
    float a = 1.0f;

    while(a > 0.0f)
    {
        a -= (1.0f / flashSpeed) * Time.deltaTime;
        image.color = new Color(1.0f, 1.0f, 1.0f, a);
        yield return null;
    }

    image.enabled = false;
}
```

Finally, back up inside of our **Flash** function, we will start the **FadeAway** coroutine after resetting the image.

```
// called when the player takes damage - listening to the OnTakeDamage event
public void Flash ()
{
    // stop all currently running FadeAway coroutines
    if(fadeAway != null)
        StopCoroutine(fadeAway);
```

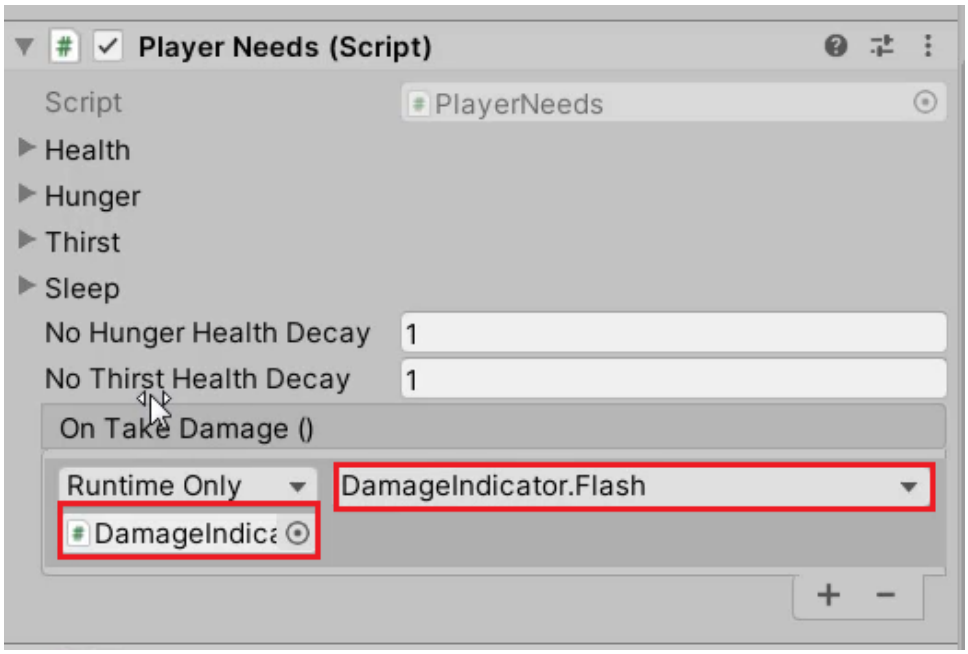


```
    // reset the image and fade it away
    image.enabled = true;
    image.color = Color.white;
    fadeAway = StartCoroutine(FadeAway());
}
```

Linking Script To Event

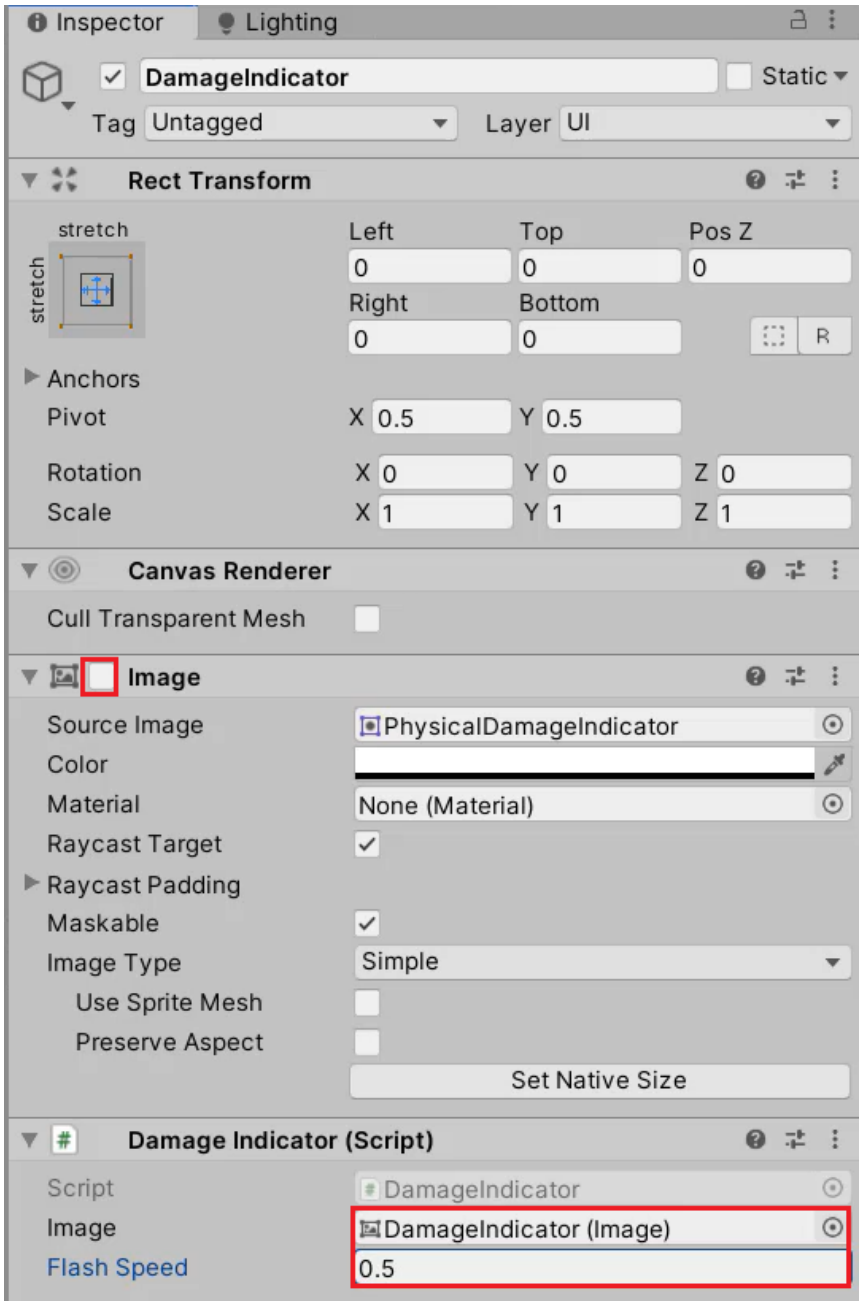
Make sure that the script is saved, and let's go back to the Editor to connect the **Flash** function to the **PlayerNeeds** script.

Select the **Player** object, and drag the **DamageIndicator** object into the PlayerNeeds.**OnTakeDamage** event listener. Then, select the DamageIndicator.**Flash** function.



This will allow DamageIndicator.**Flash** to be called whenever the player takes damage.

We can now disable the **Image** component and set up the public variables of the DamageIndiactor.

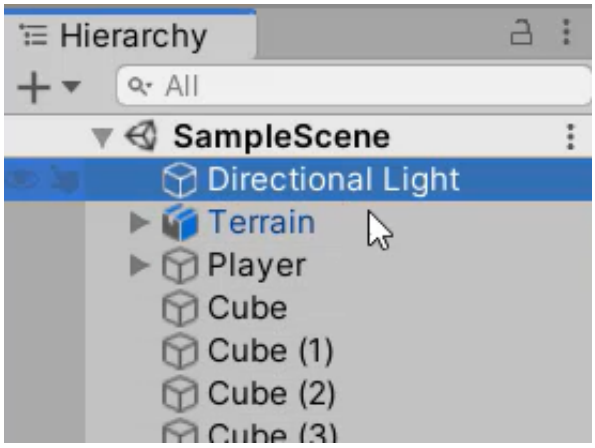


If you press Play now, you should be able to see that the damage indicator fades away every time you take damage.

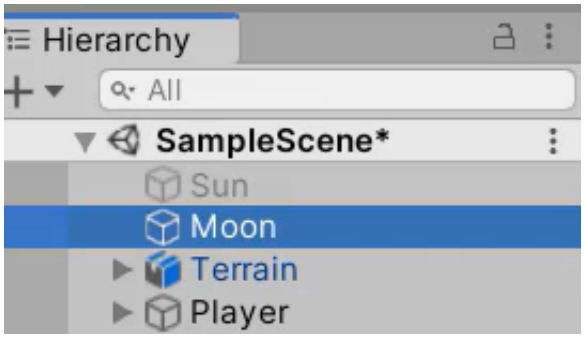
The **Day & Night Cycle** system that we're going to be working on is not specific for this survival game. You can just copy over the script in order to have this working in any game you want. Let's begin by creating the **Sun** and the **Moon**.

Setting Up The Sun

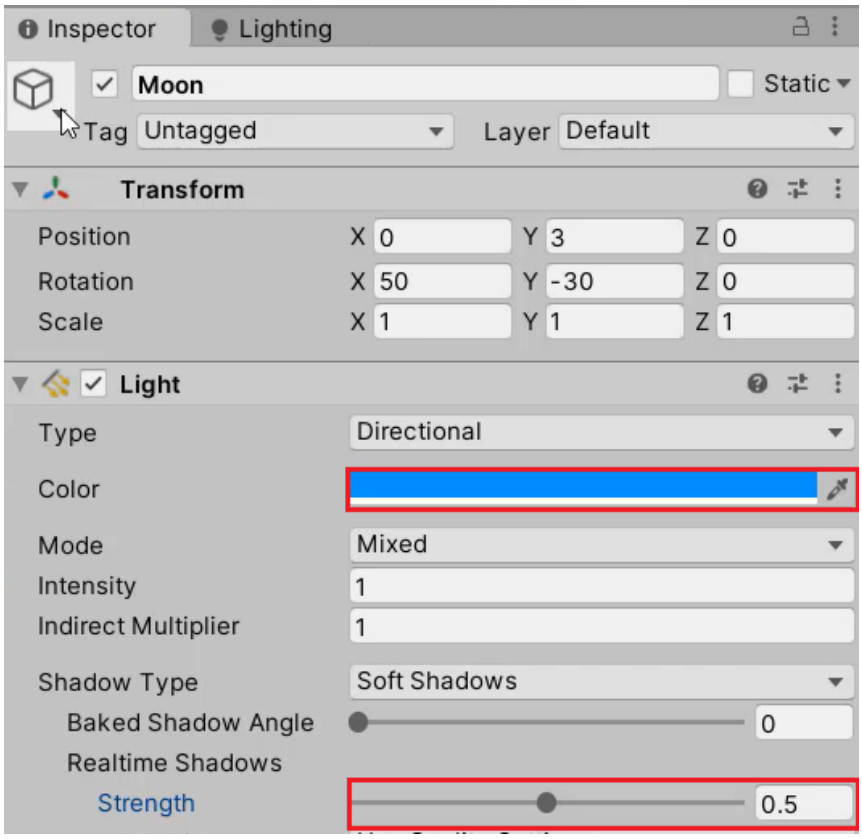
First of all, we're going to rename the **Directional Light** to "**Sun**".



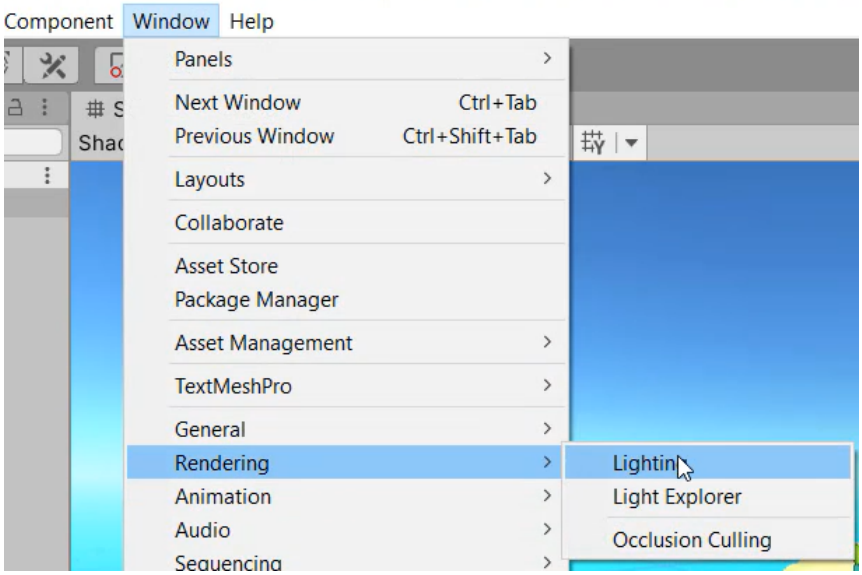
And then we can **duplicate** the light and rename it to "**Moon**". Disable the sun object for now, just so that we can work on the moon.



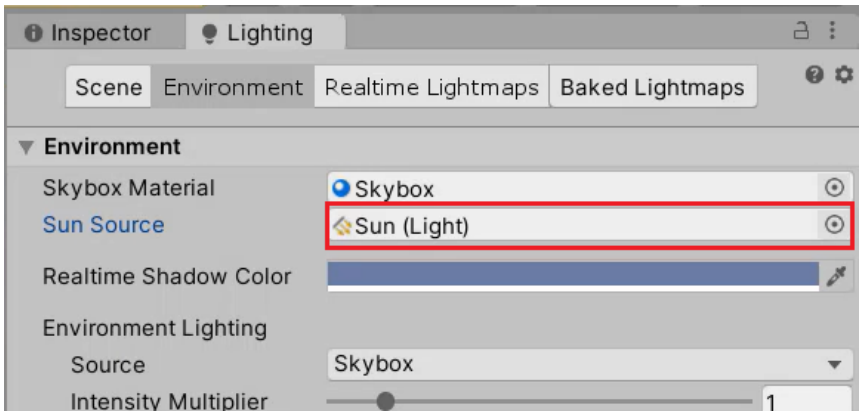
With moonlight at night time, the lighting has a blue tint. Let's change the **Color** to **blue**, and decrease the **Shadow Strength** to **0.5**.

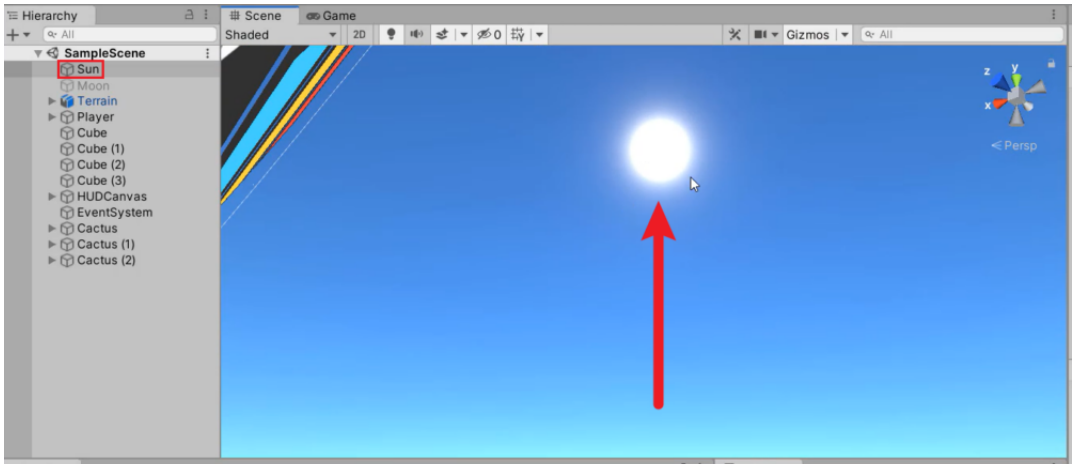


We can now disable the moon and enable the sun. Go to **Window > Rendering > Lighting** and open up the **Environment** tab.



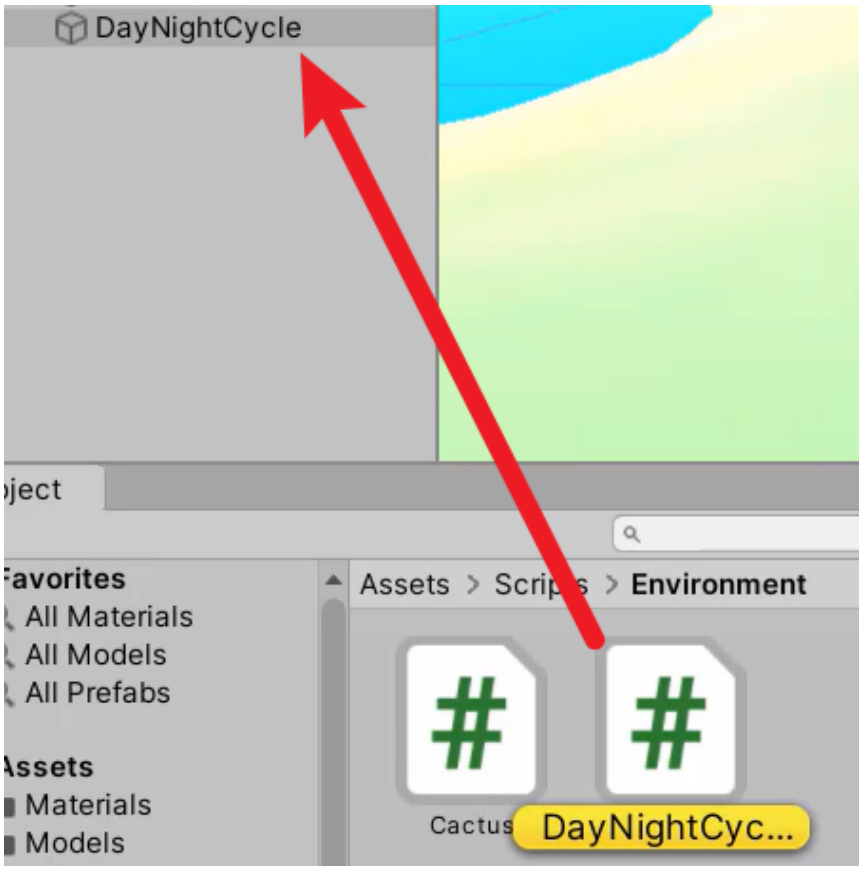
Make sure that the **Sun Source** component is filled with the directional light that we renamed to “Sun”. This is very important as the sun in the skybox is what is going to be moving up and down, modifying the color of the skybox based on the sun’s rotation.





Scripting Day Night Cycle

Now we need to create a C# script that manages the Day Night cycle. Let’s then attach it to a new empty GameObject called “DayNightCycle”.



The first variable to create inside the script is a float indicating **time**. The time will be a number ranging from zero to one– zero being 12 a.m. and one being 11:59 p.m. We can then give it a **Range** attribute to test this out in the inspector.

```
public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
    public float time;
}
```

Next, we need to know how long is a **full day** going to be in terms of seconds, which can also be represented as a float.

```
public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
    public float time;
    public float fullDayLength;
}
```

We also need to know what time will it be at the start of the game (i.e. **start time**), and how much to increment the time forward every frame (i.e. **time rate**).


```
public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
    public float time;
    public float fullDayLength;
    public float startTime = 0.4f;
    private float timeRate;
}
```

When it comes to rotating the sun and the moon around, we're going to rotate it based on the rotation when it is noon. This is just going to be a **Vector3** variable that we can assign in the Inspector.

```
public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
    public float time;
    public float fullDayLength;
    public float startTime = 0.4f;
    private float timeRate;
    public Vector3 noon;
}
```

Along with this, we also need settings specific to our sun and our moon. So let's go ahead and set up the **Headers**, as well as the **Light** component to modify their color, rotation, etc.

```
public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
    public float time;
    public float fullDayLength;
    public float startTime = 0.4f;
    private float timeRate;
    public Vector3 noon;

    [Header("Sun")]
    public Light sun;

    [Header("Moon")]
    public Light moon;
}
```

In Unity, we can use **Animation Curve** to sample a specific time to get a value. Because a **Gradient** in Unity has a range between 0.0 and 1.0, we can also use the combination of gradient and animation curve to get a specific color.

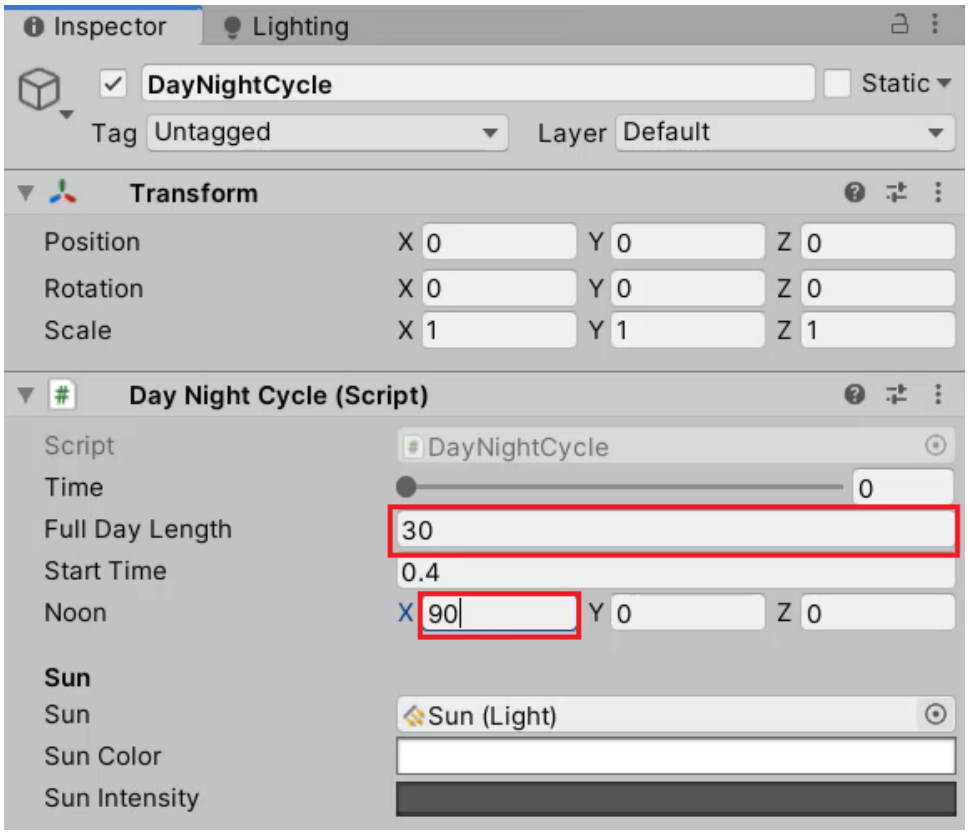
```
public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
```



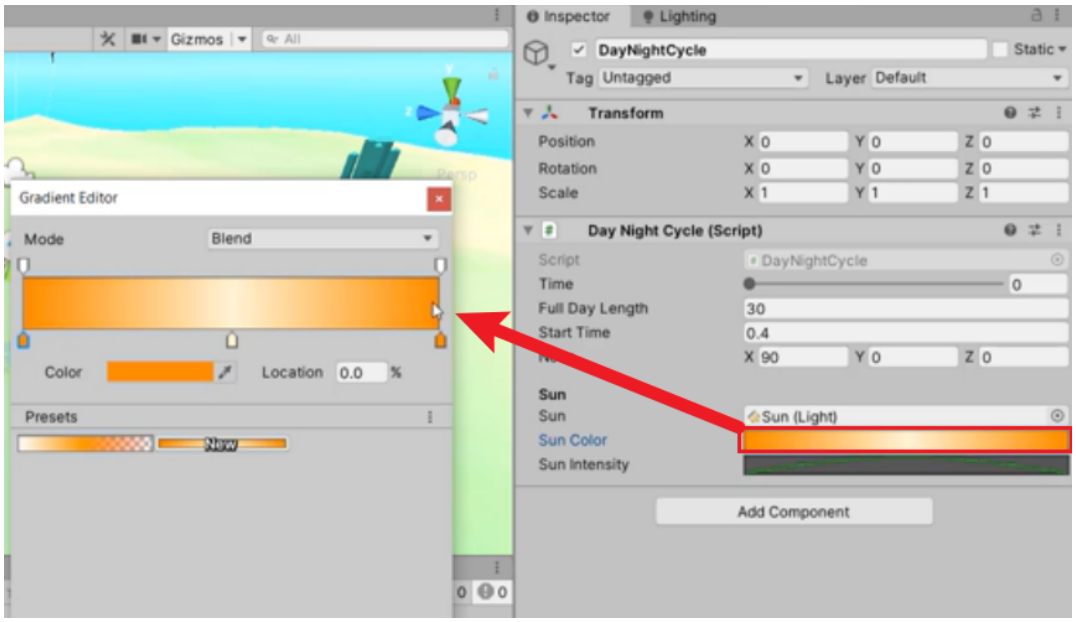
```
    public float time;
    public float fullDayLength;
    public float startTime = 0.4f;
    private float timeRate;
    public Vector3 noon;

    [Header("Sun")]
    public Light sun;
    public Gradient sunColor;
    public AnimationCurve sunIntensity;

    [Header("Moon")]
    public Light moon;
    public Gradient moonColor;
    public AnimationCurve moonIntensity;

}
```

Setting Up Sun Property Values

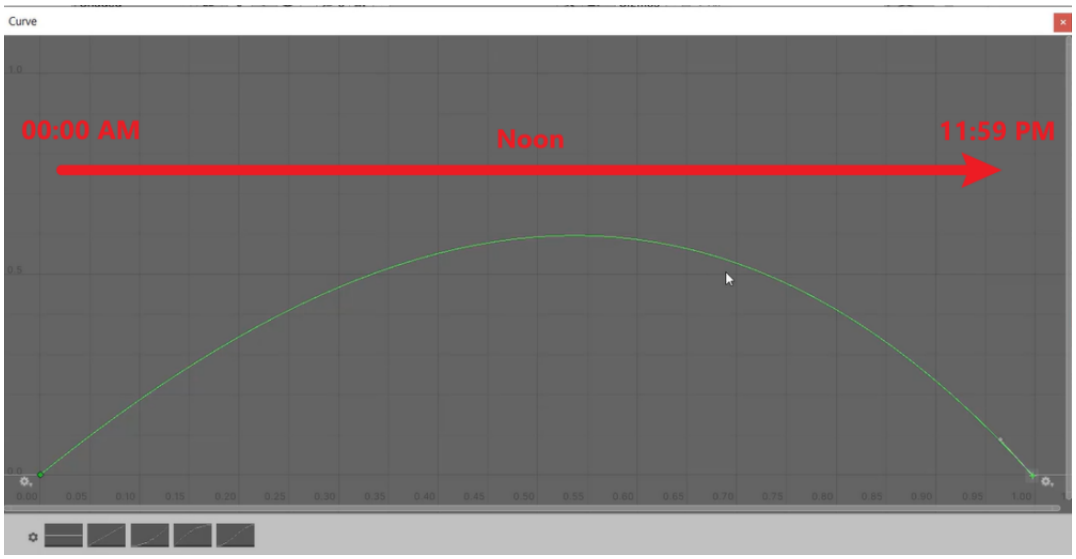
Let's save the script and go back to the Editor. We're going to set the **Full Day Length** to 30, and set the **Noon** property to (90, 0, 0).



Then set the **Sun Color** to be a custom gradient. Since our time is based on a zero (00:00 am) to one (11:59 pm) scale, the gradient on the left-hand side would be shown during the morning.



This also applies to the **Sun Intensity** property, where we can modify a curve to define sun intensity at specific times of the day.



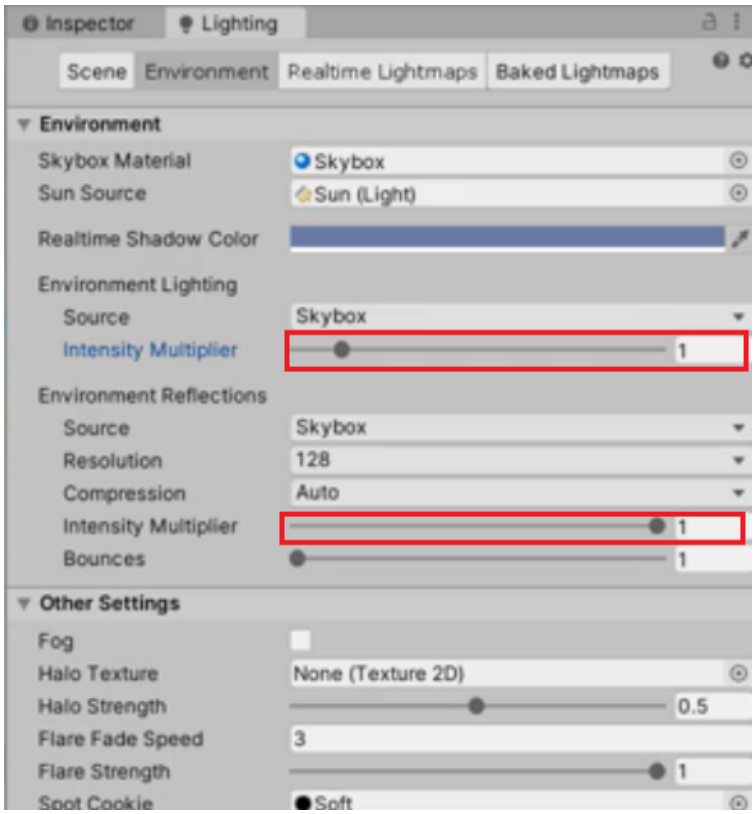
By modifying the curve as arch-shaped (**∩**), we can make it so that the intensity is low during the day, but as it goes towards midday it gets brighter, and then it goes back down during the night.

Moon Settings

Along with the light, color and intensity, our moon is going to have more specific settings, such as the **lighting intensity** and **reflections intensity**. Let's open up the **DayNightCycle** script and add the two variables under the "**Other Lighting**" heading:

```
public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
    public float time;
    public float fullDayLength;
    public float startTime = 0.4f;
    private float timeRate;
    public Vector3 noon;

    [Header("Sun")]
    public Light sun;
    public Gradient sunColor;
    public AnimationCurve sunIntensity;

    [Header("Moon")]
    public Light moon;
    public Gradient moonColor;
    public AnimationCurve moonIntensity;

    [Header("Other Lighting")]
    public AnimationCurve lightingIntensityMultiplier;
    public AnimationCurve reflectionsIntensityMultipler;
}
```

The **lighting intensity multiplier** and the **reflections intensity multiplier** decide how much lighting/reflections every object will take on from the skybox. We can also access these properties in **Window > Rendering > Lighting,** under the **Environment** tab.



Scripting Day Night Cycle

Now let's go back to our script and set up the **time** of the day and **time rate** at the start of the game. The time rate is basically how much time do we add to our time every 'frame' (which will be converted to every 'second' in the next step).

```
void Start ()
{
    timeRate = 1.0f / fullDayLength;
    time = startTime;
}
```

The **Update** function is where we're going to be updating all the positions, intensities and colors. First, we're adding **timeRate** to **time**, and we're multiplying the timeRate by **Time.deltaTime** to convert it from per frame to per second.

```
void Update ()
{
    // increment time
    time += timeRate * Time.deltaTime;
}
```

We then want to **loop** the day when the **time** is greater than or equal to one.

```
void Update ()
{
```



```
    // increment time
    time += timeRate * Time.deltaTime;

    if(time >= 1.0f)
        time = 0.0f;
}
```

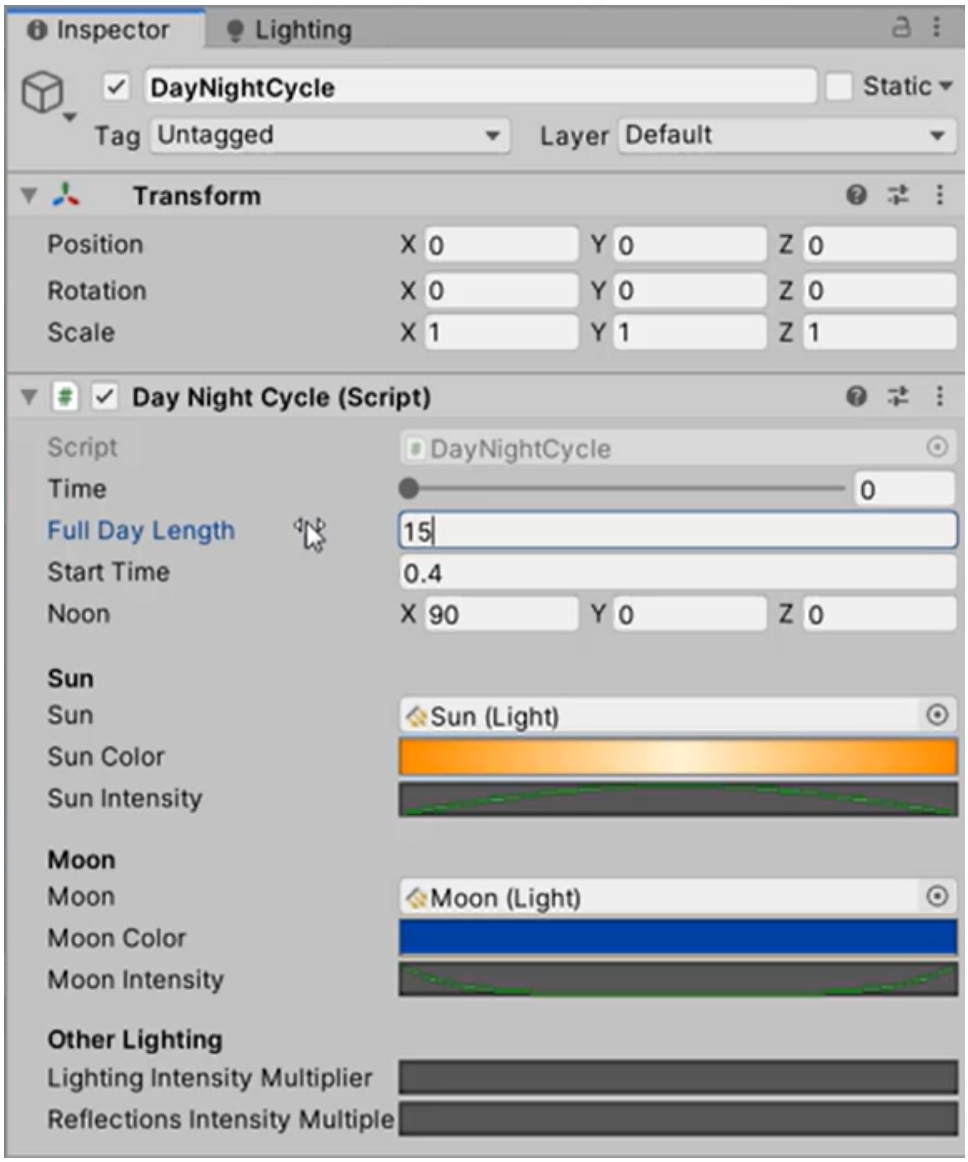
Next, we want to rotate our sun and our moon, based on the time of the day. The formula we're going to use here is going to make it so that, as our sun rotates around, the moon also rotates around in the opposite direction. When our sun is at the top of our world, the moon is at the bottom.

```
void Update ()
{
    // increment time
    time += timeRate * Time.deltaTime;

    if(time >= 1.0f)
        time = 0.0f;

    // light rotation
    sun.transform.eulerAngles = (time - 0.25f) * noon * 4.0f;
    moon.transform.eulerAngles = (time - 0.75f) * noon * 4.0f;
}
```

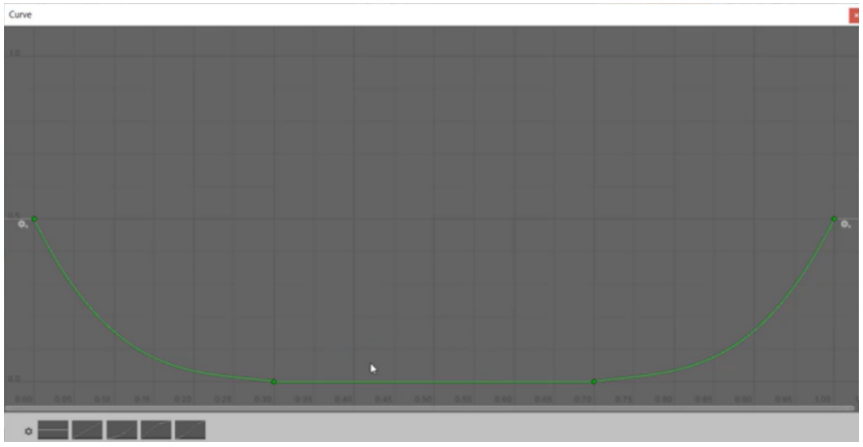
The next step is setting up our sun and moon **intensity**. To get the values from AnimationCurve, we need to be calling the **Evaluate** function. As a parameter to the function, we need to give it our **time** variable.

```
void Update ()
{
    // increment time
    time += timeRate * Time.deltaTime;

    if(time >= 1.0f)
        time = 0.0f;

    // light rotation
    sun.transform.eulerAngles = (time - 0.25f) * noon * 4.0f;
    moon.transform.eulerAngles = (time - 0.75f) * noon * 4.0f;

    // light intensity
    sun.intensity = sunIntensity.Evaluate(time);
    moon.intensity = moonIntensity.Evaluate(time);
}
```

Let's go back to the inspector and fill in the rest of our variables.



Note that the **moon intensity** curve is going to take the opposite shape (U-shape) to the sun intensity curve (∩). However, its maximum value should be 0.5 instead of 1 since the moonlight isn't supposed to be as bright as the sun.



Now we have our sun and moon rotating around in our world. Along with this, we want the colors to be changing as the days go on.

Changing Sky Color

So let's continue editing the **DayNightCycle** class. In the **Update** function, we will use the **Evaluate(time)** function again to get the specific color for the specific time of the day.

```
void Update ()
{
    // increment time
    time += timeRate * Time.deltaTime;

    if(time >= 1.0f)
        time = 0.0f;

    // light rotation
    sun.transform.eulerAngles = (time - 0.25f) * noon * 4.0f;
    moon.transform.eulerAngles = (time - 0.75f) * noon * 4.0f;

    // light intensity
    sun.intensity = sunIntensity.Evaluate(time);
    moon.intensity = moonIntensity.Evaluate(time);

    // change colors
    sun.color = sunColor.Evaluate(time);
    moon.color = moonColor.Evaluate(time);

}
```

Enabling/Disabling The Sun

We will now enable or disable the sun based on the time of the day. This is because if the sun is active at midnight, there will be some lighting artifacts that can still shine through at the bottom of objects, so we want to get rid of them.

For this, we need to check to see if the sun is still active in the scene using **gameObject.activeInHierarchy**, and also check if the sun's **intensity** is zero. If both conditions are met, we can disable the sun.

Otherwise, if the sun's intensity is greater than zero and the sun is disabled, then we will enable the sun to make it rise again.

```
void Update ()
{
    // increment time
    time += timeRate * Time.deltaTime;

    if(time >= 1.0f)
        time = 0.0f;

    // light rotation
    sun.transform.eulerAngles = (time - 0.25f) * noon * 4.0f;
```



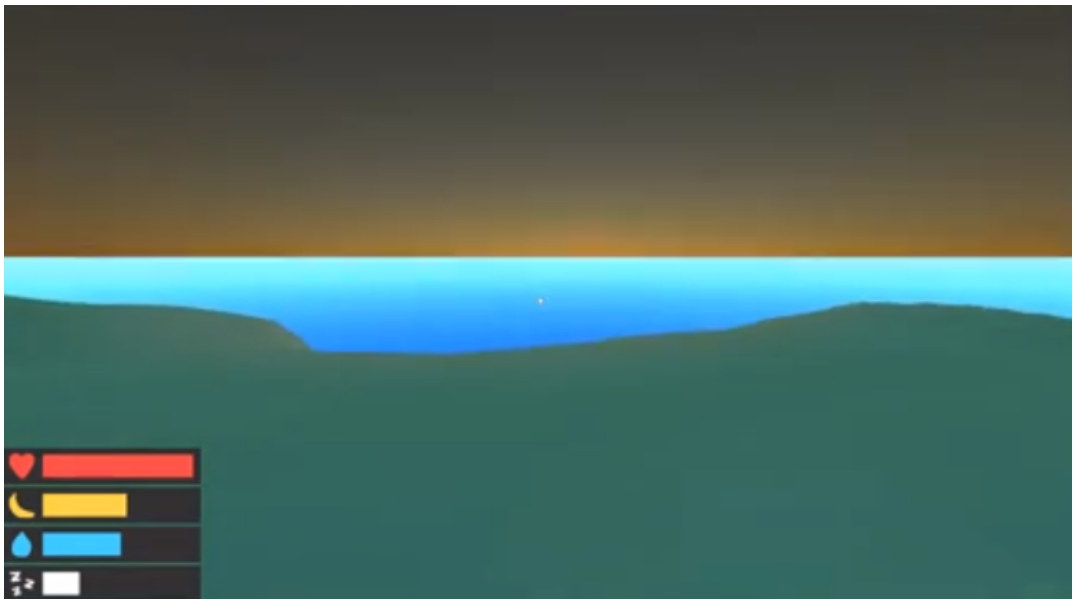
```
    moon.transform.eulerAngles = (time - 0.75f) * noon * 4.0f;

    // light intensity
    sun.intensity = sunIntensity.Evaluate(time);
    moon.intensity = moonIntensity.Evaluate(time);

    // change colors
    sun.color = sunColor.Evaluate(time);
    moon.color = moonColor.Evaluate(time);

    // enable / disable the sun
    if(sun.intensity == 0 && sun.gameObject.activeInHierarchy)
        sun.gameObject.SetActive(false);
    else if(sun.intensity > 0 && !sun.gameObject.activeInHierarchy)
        sun.gameObject.SetActive(true);
}
```

We can apply the exact same code block to the moon as well.

```
void Update ()
{
    // increment time
    time += timeRate * Time.deltaTime;

    if(time >= 1.0f)
        time = 0.0f;

    // light rotation
    sun.transform.eulerAngles = (time - 0.25f) * noon * 4.0f;
    moon.transform.eulerAngles = (time - 0.75f) * noon * 4.0f;

    // light intensity
    sun.intensity = sunIntensity.Evaluate(time);
    moon.intensity = moonIntensity.Evaluate(time);

    // change colors
    sun.color = sunColor.Evaluate(time);
    moon.color = moonColor.Evaluate(time);

    // enable / disable the sun
    if(sun.intensity == 0 && sun.gameObject.activeInHierarchy)
        sun.gameObject.SetActive(false);
    else if(sun.intensity > 0 && !sun.gameObject.activeInHierarchy)
        sun.gameObject.SetActive(true);

    // enable / disable the moon
    if(moon.intensity == 0 && moon.gameObject.activeInHierarchy)
        moon.gameObject.SetActive(false);
    else if(moon.intensity > 0 && !moon.gameObject.activeInHierarchy)
        moon.gameObject.SetActive(true);
}
```


Lighting Intensity

If you now press Play, our sun and moon get enabled and disabled based on the time of the day.



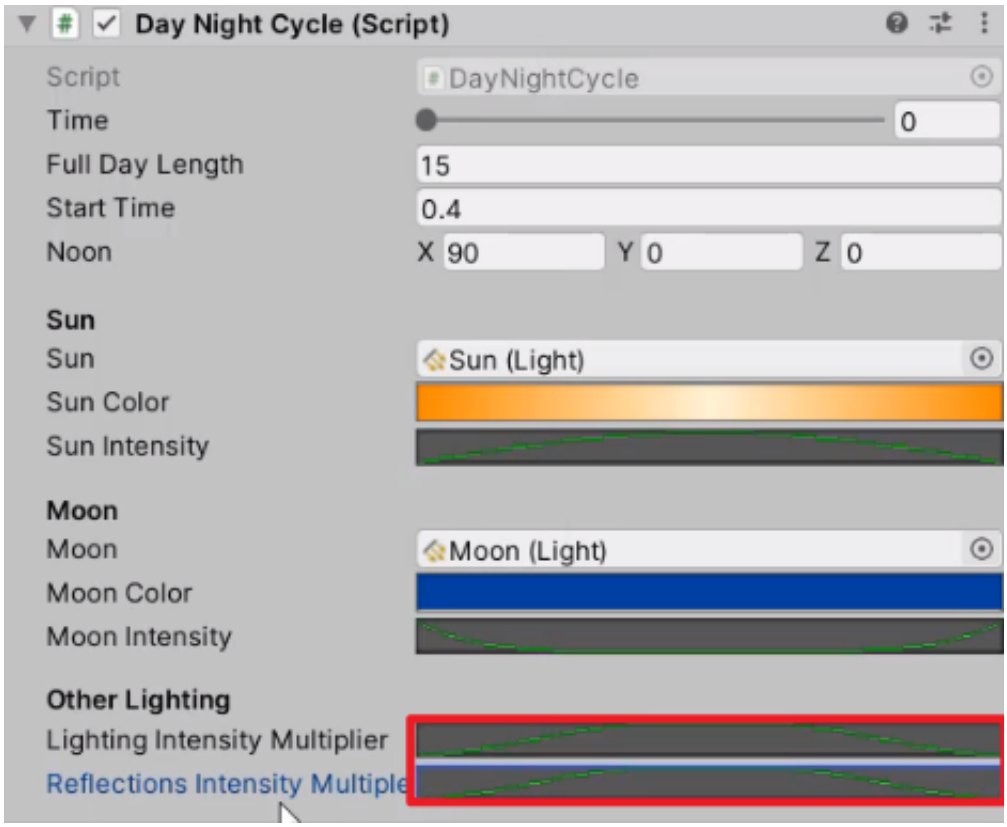
You'll also see that when it turns nighttime, we can still see the ocean clear as blue, and the objects are brightly lit.

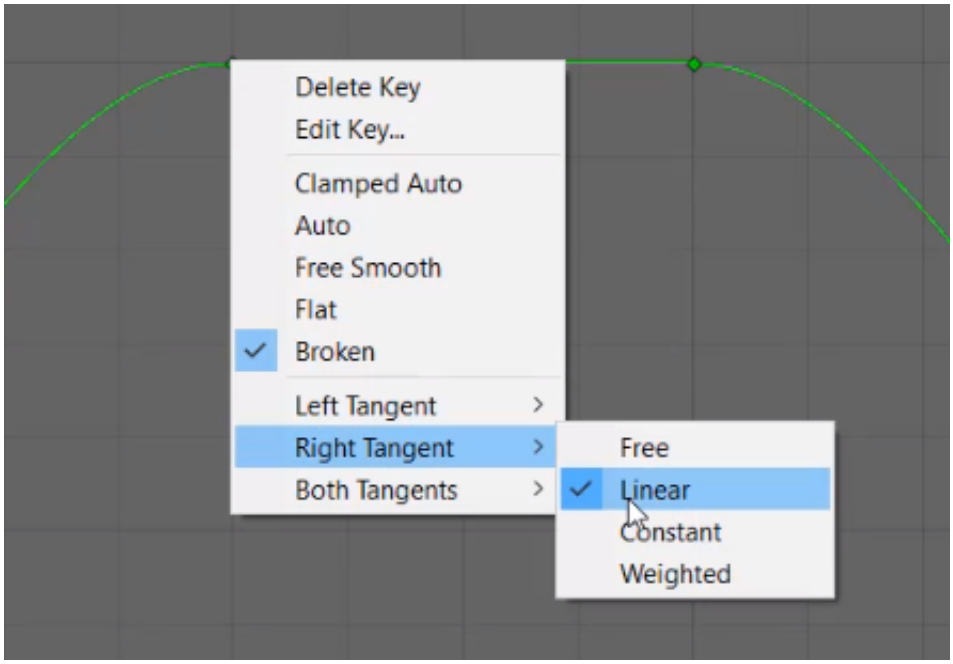
To fix this problem, we need to access the render settings in the script and adjust the **ambient intensity** and the **reflection intensity** so that they change based on the current time of the day.

```
void Update ()
{
    // increment time
    time += timeRate * Time.deltaTime;

    if(time >= 1.0f)
        time = 0.0f;
```

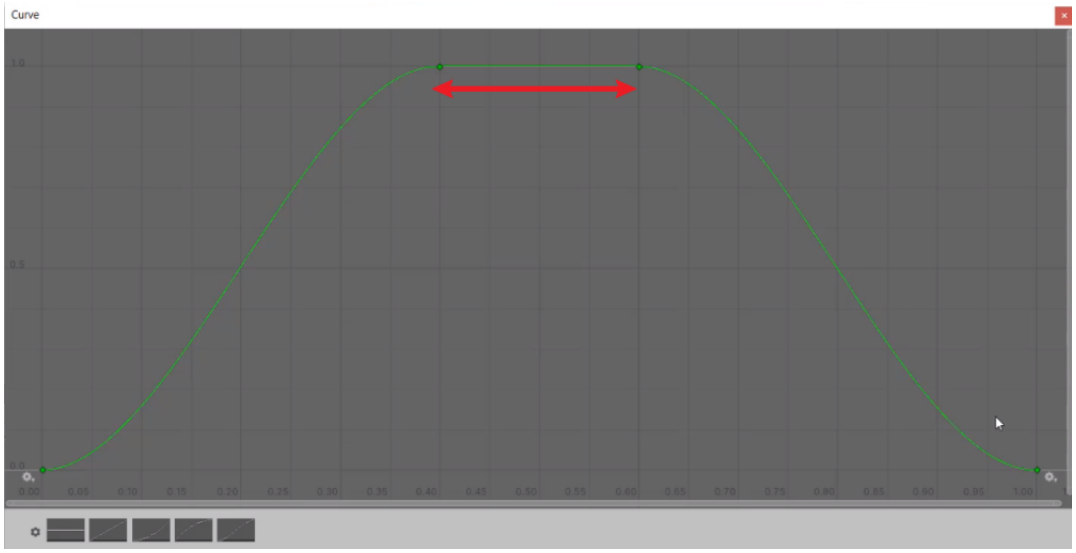
```
    // light rotation
    sun.transform.eulerAngles = (time - 0.25f) * noon * 4.0f;
    moon.transform.eulerAngles = (time - 0.75f) * noon * 4.0f;

    // light intensity
    sun.intensity = sunIntensity.Evaluate(time);
    moon.intensity = moonIntensity.Evaluate(time);

    // change colors
    sun.color = sunColor.Evaluate(time);
    moon.color = moonColor.Evaluate(time);

    // enable / disable the sun
    if(sun.intensity == 0 && sun.gameObject.activeInHierarchy)
        sun.gameObject.SetActive(false);
    else if(sun.intensity > 0 && !sun.gameObject.activeInHierarchy)
        sun.gameObject.SetActive(true);

    // enable / disable the moon
    if(moon.intensity == 0 && moon.gameObject.activeInHierarchy)
        moon.gameObject.SetActive(false);
    else if(moon.intensity > 0 && !moon.gameObject.activeInHierarchy)
        moon.gameObject.SetActive(true);

    // lighting and reflections intensity
    RenderSettings.ambientIntensity = lightingIntensityMultiplier.Evaluate(time);
    RenderSettings.reflectionIntensity = reflectionsIntensityMultipler.Evaluate(time)
;
}
```

To make it more realistic, we could open up the **Intensity Multiplier** settings in the Inspector, and **Right-click > Right Tangent > Linear**.





This will make the middle part of the curve flat so that the intensity multiplier is at 1 during the day, and it’s going to fall to zero at nighttime.



In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

PlayerNeeds.cs

Found in Project/Assets/Scripts/Player

This script manages the needs (health, hunger, thirst, sleep) of a game's player character. It determines how these needs change over time and under certain conditions such as lack of food or water. It also handles player healing, eating, drinking, sleeping, taking damage, dying, and displays user interface bars corresponding to each need.

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

public class PlayerNeeds : MonoBehaviour, IDamagable
{
    public Need health;
    public Need hunger;
    public Need thirst;
    public Need sleep;

    public float noHungerHealthDecay;
    public float noThirstHealthDecay;

    public UnityEvent onTakeDamage;

    void Start ()
    {
        // set the start values
        health.curValue = health.startValue;
        hunger.curValue = hunger.startValue;
        thirst.curValue = thirst.startValue;
        sleep.curValue = sleep.startValue;
    }

    void Update ()
    {
        // decay needs over time
        hunger.Subtract(hunger.decayRate * Time.deltaTime);
        thirst.Subtract(thirst.decayRate * Time.deltaTime);
        sleep.Add(sleep.regenRate * Time.deltaTime);

        // decay health over time if no hunger or thirst
        if(hunger.curValue == 0.0f)
            health.Subtract(noHungerHealthDecay * Time.deltaTime);
        if(thirst.curValue == 0.0f)
            health.Subtract(noThirstHealthDecay * Time.deltaTime);
```



```
        // check if player is dead
        if(health.curValue == 0.0f)
        {
            Die();
        }

        // update UI bars
        health.uiBar.fillAmount = health.GetPercentage();
        hunger.uiBar.fillAmount = hunger.GetPercentage();
        thirst.uiBar.fillAmount = thirst.GetPercentage();
        sleep.uiBar.fillAmount = sleep.GetPercentage();
    }

    // adds to the player's HEALTH
    public void Heal (float amount)
    {
        health.Add(amount);
    }

    // adds to the player's HUNGER
    public void Eat (float amount)
    {
        hunger.Add(amount);
    }

    // adds to the player's THIRST
    public void Drink (float amount)
    {
        thirst.Add(amount);
    }

    // subtracts from the player's SLEEP
    public void Sleep (float amount)
    {
        sleep.Subtract(amount);
    }

    // called when the player takes physical damage (fire, enemy, etc)
    public void TakePhysicalDamage (int amount)
    {
        health.Subtract(amount);
        onTakeDamage?.Invoke();
    }

    // called when the player's health reaches 0
    public void Die ()
    {
        Debug.Log("Player is dead");
    }
}

[System.Serializable]
public class Need
{
```

```
    [HideInInspector]
    public float curValue;
    public float maxValue;
    public float startValue;
    public float regenRate;
    public float decayRate;
    public Image uiBar;

    // add to the need
    public void Add (float amount)
    {
        curValue = Mathf.Min(curValue + amount, maxValue);
    }

    // subtract from the need
    public void Subtract (float amount)
    {
        curValue = Mathf.Max(curValue - amount, 0.0f);
    }

    // return the percentage value (0.0 - 1.0)
    public float GetPercentage ()
    {
        return curValue / maxValue;
    }
}

public interface IDamagable
{
    void TakePhysicalDamage(int damageAmount);
}
```

PlayerController.cs

Found in Project/Assets/Scripts/Player

This script controls player movement, adding functionality for forward/backward and right/left movement, jumping, and responding to mouse input for camera rotation. It includes checks to determine if the player is grounded before jumping and utilizes Unity's Input System for fluid control interactions. Player movement is also visualized using Gizmos for debugging purposes.

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

public class PlayerController : MonoBehaviour
{
    [Header("Movement")]
    public float moveSpeed;
    private Vector2 curMovementInput;
    public float jumpForce;
    public LayerMask groundLayerMask;
```



```
    [Header("Look")]
    public Transform cameraContainer;
    public float minXLook;
    public float maxXLook;
    private float camCurXRot;
    public float lookSensitivity;

    private Vector2 mouseDelta;

    // components
    private Rigidbody rig;

    void Awake ()
    {
        // get our components
        rig = GetComponent<Rigidbody>();
    }

    void Start ()
    {
        // lock the cursor at the start of the game
        Cursor.lockState = CursorLockMode.Locked;
    }

    void FixedUpdate ()
    {
        Move();
    }

    void LateUpdate ()
    {
        CameraLook();
    }

    void Move ()
    {
        // calculate the move direction relative to where we're facing.
        Vector3 dir = transform.forward * curMovementInput.y + transform.right * curM
ovementInput.x;
        dir *= moveSpeed;
        dir.y = rig.linearVelocity.y;

        // assign our Rigidbody velocity
        rig.linearVelocity = dir;
    }

    void CameraLook ()
    {
        // rotate the camera container up and down
        camCurXRot += mouseDelta.y * lookSensitivity;
        camCurXRot = Mathf.Clamp(camCurXRot, minXLook, maxXLook);
        cameraContainer.localEulerAngles = new Vector3(-camCurXRot, 0, 0);

        // rotate the player left and right
```

```
        transform.eulerAngles += new Vector3(0, mouseDelta.x * lookSensitivity, 0);
    }

    // called when we move our mouse - managed by the Input System
    public void OnLookInput (InputAction.CallbackContext context)
    {
        mouseDelta = context.ReadValue<Vector2>();
    }

    // called when we press WASD - managed by the Input System
    public void OnMoveInput (InputAction.CallbackContext context)
    {
        // are we holding down a movement button?
        if(context.phase == InputActionPhase.Performed)
        {
            curMovementInput = context.ReadValue<Vector2>();
        }
        // have we let go of a movement button?
        else if(context.phase == InputActionPhase.Canceled)
        {
            curMovementInput = Vector2.zero;
        }
    }

    // called when we press down on the spacebar - managed by the Input System
    public void OnJumpInput (InputAction.CallbackContext context)
    {
        // is this the first frame we're pressing the button?
        if(context.phase == InputActionPhase.Started)
        {
            // are we standing on the ground?
            if(IsGrounded())
            {
                // add force updwards
                rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
            }
        }
    }

    bool IsGrounded ()
    {
        Ray[] rays = new Ray[4]
        {
            new Ray(transform.position + (transform.forward * 0.2f) + (Vector3.up * 0
.01f), Vector3.down),
            new Ray(transform.position + (-transform.forward * 0.2f) + (Vector3.up *
0.01f), Vector3.down),
            new Ray(transform.position + (transform.right * 0.2f) + (Vector3.up * 0.0
1f), Vector3.down),
            new Ray(transform.position + (-transform.right * 0.2f) + (Vector3.up * 0.
01f), Vector3.down)
        };

        for(int i = 0; i < rays.Length; i++)
        {
```

```
            if(Physics.Raycast(rays[i], 0.1f, groundLayerMask))
            {
                return true;
            }
        }

        return false;
    }

    private void OnDrawGizmos ()
    {
        Gizmos.color = Color.red;

        Gizmos.DrawRay(transform.position + (transform.forward * 0.2f), Vector3.down)
;
        Gizmos.DrawRay(transform.position + (-transform.forward * 0.2f), Vector3.down
);
        Gizmos.DrawRay(transform.position + (transform.right * 0.2f), Vector3.down);
        Gizmos.DrawRay(transform.position + (-transform.right * 0.2f), Vector3.down);
    }
}
```

Cactus.cs

Found in Project/Assets/Scripts/Environment

This script is for a ‘Cactus’ object in the game that inflicts damage to any ‘IDamagable’ game objects that are in contact with it. It continually deals a set amount of damage at a given rate to ‘IDamagable’ objects as long as they are in contact with the ‘Cactus’.

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

public class Cactus : MonoBehaviour
{
    public int damage;
    public float damageRate;

    private List<IDamagable> thingsToDamage = new List<IDamagable>();

    void Start ()
    {
        StartCoroutine(DealDamage());
    }

    IEnumerator DealDamage ()
    {
        // every "damageRate" seconds, damage all thingsToDamage
        while(true)
        {
            for(int i = 0; i < thingsToDamage.Count; i++)
            {
```



```
                thingsToDamage[i].TakePhysicalDamage(damage);
            }

            yield return new WaitForSeconds(damageRate);
        }
    }

    // called when an object collides with the cactus
    private void OnCollisionEnter (Collision collision)
    {
        // if it's an IDamagable, add it to the list
        if(collision.gameObject.GetComponent<IDamagable>() != null)
        {
            thingsToDamage.Add(collision.gameObject.GetComponent<IDamagable>());
        }
    }

    // called when an object stops colliding with the cactus
    private void OnCollisionExit (Collision collision)
    {
        // if it's an IDamagable, remove it from the list
        if(collision.gameObject.GetComponent<IDamagable>() != null)
        {
            thingsToDamage.Remove(collision.gameObject.GetComponent<IDamagable>());
        }
    }
}
```

DayNightCycle.cs

Found in Project/Assets/Scripts/Environment

The script enables a dynamic day-night cycle in a game. It controls the rotation of the sun and moon, adjusts their intensity and colors based on the time of day, and also modifies the ambient and reflection lighting accordingly. The sun and moon lights are activated and deactivated dependent on their intensity values.

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

public class DayNightCycle : MonoBehaviour
{
    [Range(0.0f, 1.0f)]
    public float time;
    public float fullDayLength;
    public float startTime = 0.4f;
    private float timeRate;
    public Vector3 noon;

    [Header("Sun")]
    public Light sun;
    public Gradient sunColor;
```



```
    public AnimationCurve sunIntensity;

    [Header("Moon")]
    public Light moon;
    public Gradient moonColor;
    public AnimationCurve moonIntensity;

    [Header("Other Lighting")]
    public AnimationCurve lightingIntensityMultiplier;
    public AnimationCurve reflectionsIntensityMultipler;

    void Start ()
    {
        timeRate = 1.0f / fullDayLength;
        time = startTime;
    }

    void Update ()
    {
        // increment time
        time += timeRate * Time.deltaTime;

        if(time >= 1.0f)
            time = 0.0f;

        // light rotation
        sun.transform.eulerAngles = (time - 0.25f) * noon * 4.0f;
        moon.transform.eulerAngles = (time - 0.75f) * noon * 4.0f;

        // light intensity
        sun.intensity = sunIntensity.Evaluate(time);
        moon.intensity = moonIntensity.Evaluate(time);

        // change colors
        sun.color = sunColor.Evaluate(time);
        moon.color = moonColor.Evaluate(time);

        // enable / disable the sun
        if(sun.intensity == 0 && sun.gameObject.activeInHierarchy)
            sun.gameObject.SetActive(false);
        else if(sun.intensity > 0 && !sun.gameObject.activeInHierarchy)
            sun.gameObject.SetActive(true);

        // enable / disable the moon
        if(moon.intensity == 0 && moon.gameObject.activeInHierarchy)
            moon.gameObject.SetActive(false);
        else if(moon.intensity > 0 && !moon.gameObject.activeInHierarchy)
            moon.gameObject.SetActive(true);

        // lighting and reflections intensity
        RenderSettings.ambientIntensity = lightingIntensityMultiplier.Evaluate(time);
        RenderSettings.reflectionIntensity = reflectionsIntensityMultipler.Evaluate(t
ime);
    }
}
```


DamageIndicator.cs

Found in Project/Assets/Scripts/UI

This script manages a damage indicator for a game. When the player takes damage, the image flashes on the screen and then fades away. If the player takes damage again before the previous flash fully fades, the image resets and starts fading anew.

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

public class DamageIndicator : MonoBehaviour
{
    public Image image;
    public float flashSpeed;

    private Coroutine fadeAway;

    // called when the player takes damage - listening to the OnTakeDamage event
    public void Flash ()
    {
        // stop all currently running FadeAway coroutines
        if(fadeAway != null)
            StopCoroutine(fadeAway);

        // reset the image and fade it away
        image.enabled = true;
        image.color = Color.white;
        fadeAway = StartCoroutine(FadeAway());
    }

    // fades the image away over time
    IEnumerator FadeAway ()
    {
        float a = 1.0f;

        while(a > 0.0f)
        {
            a -= (1.0f / flashSpeed) * Time.deltaTime;
            image.color = new Color(1.0f, 1.0f, 1.0f, a);
            yield return null;
        }

        image.enabled = false;
    }
}
```

