

# Roll-A-Ball

Omang Baheti, Rishav Banerjee

January 2, 2025

This tutorial will be done on a live zoom session soon.  
The link for that VOD will be updated here once complete.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Introduction To Unity</b>	<b>2</b>
2.1	Editor Layout . . . . .	2
2.2	Unity Workflow . . . . .	3
<b>3</b>	<b>GameObjects and Components</b>	<b>4</b>
3.1	Sample Scene . . . . .	4
3.2	Creating GameObjects . . . . .	5
3.3	Navigating the Scene View . . . . .	6
3.4	Manipulating GameObjects . . . . .	6
<b>4</b>	<b>Introduction to Unity Physics</b>	<b>9</b>
<b>5</b>	<b>Making Git commits</b>	<b>9</b>
<b>6</b>	<b>Introduction To Scripting</b>	<b>11</b>
6.1	Creating a custom script . . . . .	11
6.2	Registering User Inputs . . . . .	14
6.3	Processing User Inputs . . . . .	15
6.4	Interfacing with other components . . . . .	18
6.5	Good Coding Practices ( <b>Optional</b> ) . . . . .	22
6.5.1	Private Variables and SerializedFields . . . . .	22
6.5.2	Single Responsibility Principle . . . . .	22
<b>7</b>	<b>Polish</b>	<b>25</b>
7.1	Adding Walls . . . . .	25
7.2	Adding Materials . . . . .	26
<b>8</b>	<b>Conclusion</b>	<b>27</b>
8.1	Bonus . . . . .	27

---

# 1. Introduction

Welcome to your very first Unity project! The goal of this tutorial is to make you familiar with Unity's interface and basic scripting. We will be making the Roll-a-Ball project (see [Demo](#) here). **You are expected to follow along and implement everything explained in this document, along with making git commits at relevant points.** Your final submission will be the same project with all the showcased functionality + a bonus feature we'll discuss at the end.

This guide expects you have followed the previous 2 guides on [setting up your unity environment](#) and [creating a new unity project with git](#). If you have not done either of those, please do that first and then revisit this tutorial.

The Roll-a-Ball tutorial (a classic Unity 101 project) will have you create a simple environment where you can roll around a ball in a plane. We will achieve this by making a physics based ball controller using the keys W, A, S and D.

## 2. Introduction To Unity

First, using the Unity Hub, make a new Unity project and call it `RollABall`. Make sure to initialize it with Git, and push the project onto your GitHub account. If you are unsure how to do so, please follow the [previous guide](#).

### 2.1. Editor Layout

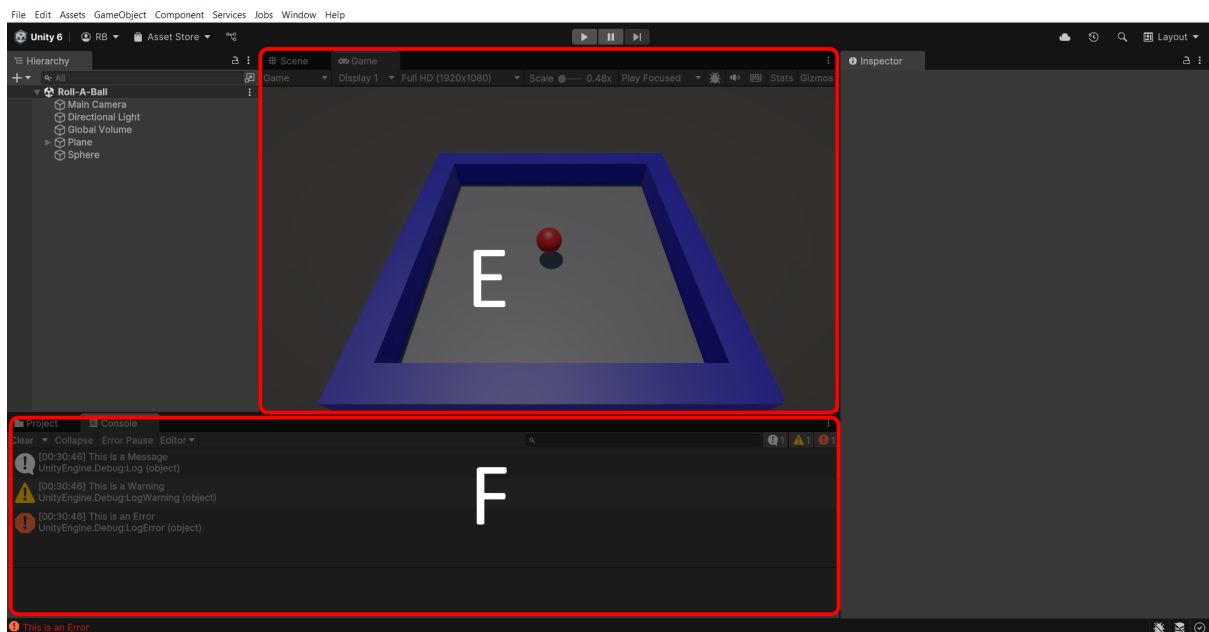
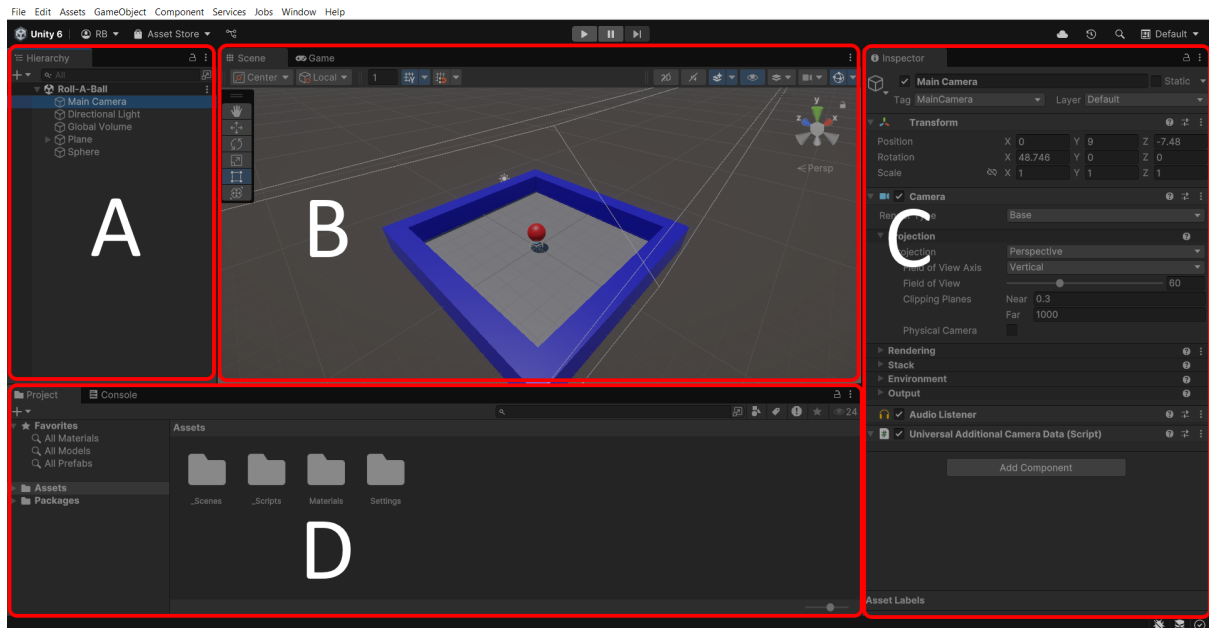
The Unity editor is composed of several windows. These windows can be dragged around, docked to other parts of the screen, or completely separated to its own window. All the possible windows in Unity can be seen in the Window option on the top menu bar.

The view you are currently seeing is the Default Layout. It is selected from the top right dropdown, along with some other presets, and the option to set your own view.

The windows in this default view (some of the most important windows in Unity) are as follows:

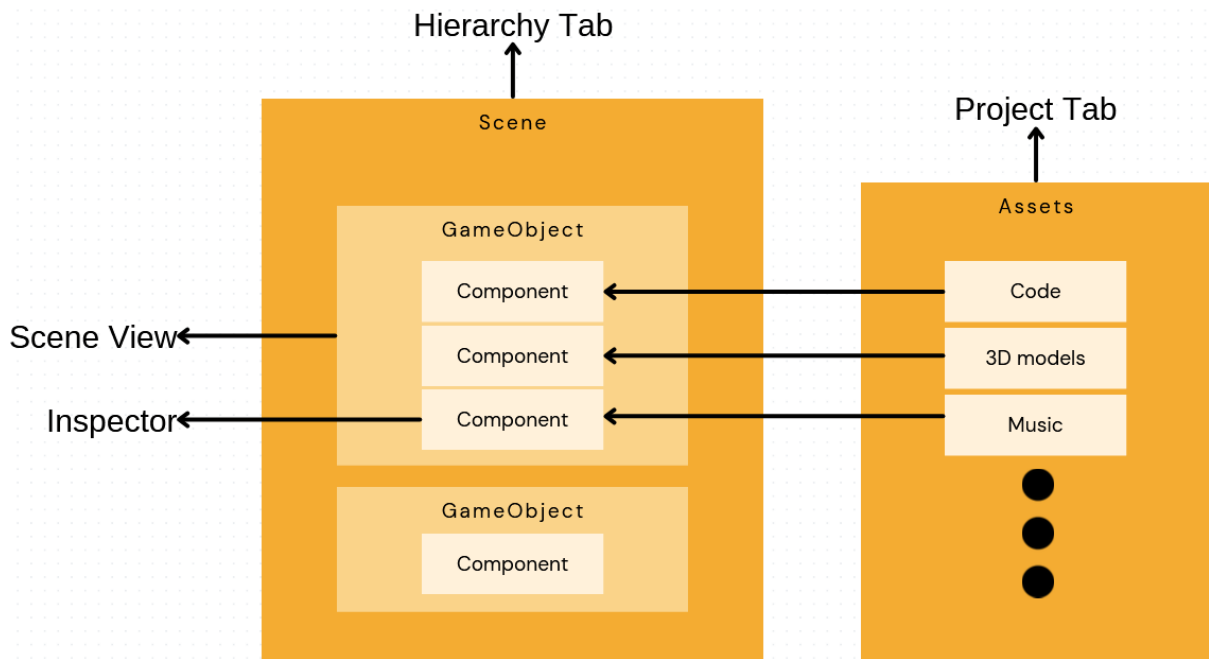
- A. Hierarchy - It consists of all the GameObjects contained in a scene in a list view
- B. Scene View - A 3D workspace where you can move, rotate, and resize GameObjects in your scene.
- C. Inspector - The inspector shows contextual information depending on selected items. If you select a GameObject from the Hierarchy, it shows all the components associated with that GameObject that define its behavior. If you select an asset from the Project Tab, it shows properties associated with the asset.
- D. Project Tab - This tab is a file browser for your unity project. It helps organize all your project assets (images, 3D models, sounds, etc.).
- E. Game View - This windows shows what the game would look like to the player.

**F.** Console - Displays errors, warnings, and messages from your scripts.



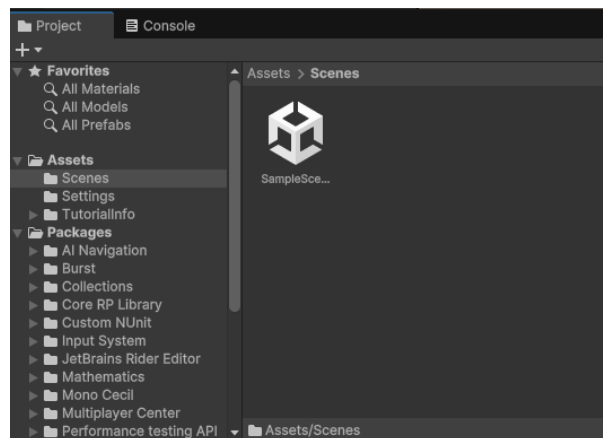
## 2.2. Unity Workflow

A Unity project is usually made up of multiple **Scenes**, each representing a different level or section. A scene contains **GameObjects**, which are made by combining smaller modular units called **Components**. Components are often pre-made scripts provided by Unity or custom scripts written by developers to add specific features to their game. Components also often utilize other assets such as 3D models, images, sound effects etc.

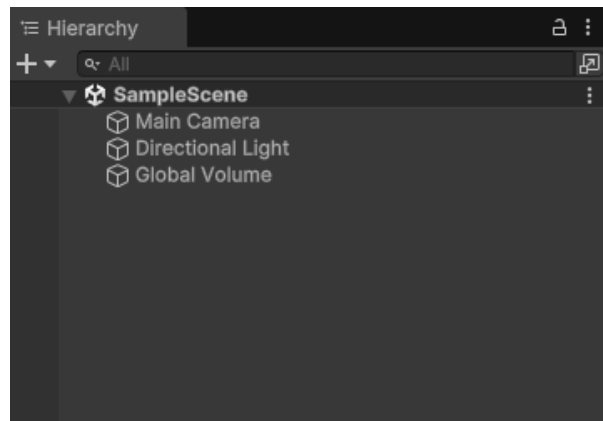


### 3. GameObjects and Components

#### 3.1. Sample Scene



Creating a new project contains a scene called **SampleScene** which opens by default. You can find the scene under **Assets/Scene** in the project window.



A sample scene in Unity includes three GameObjects: the Main Camera, Directional Light, and Global Volume. The Main Camera renders the scene which can be previewed in the Game Window, while the Directional Light handles the lighting and Global Volume manages post-processing effects.

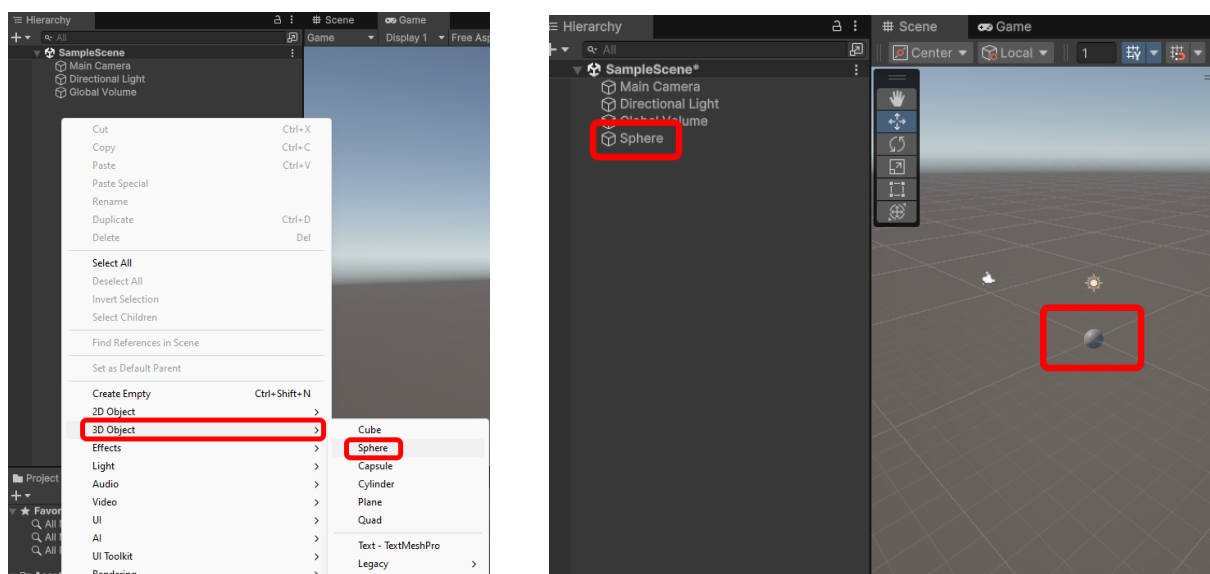
A GameObject can be selected either via the Hierarchy Window or the Scene Window.

#### Note

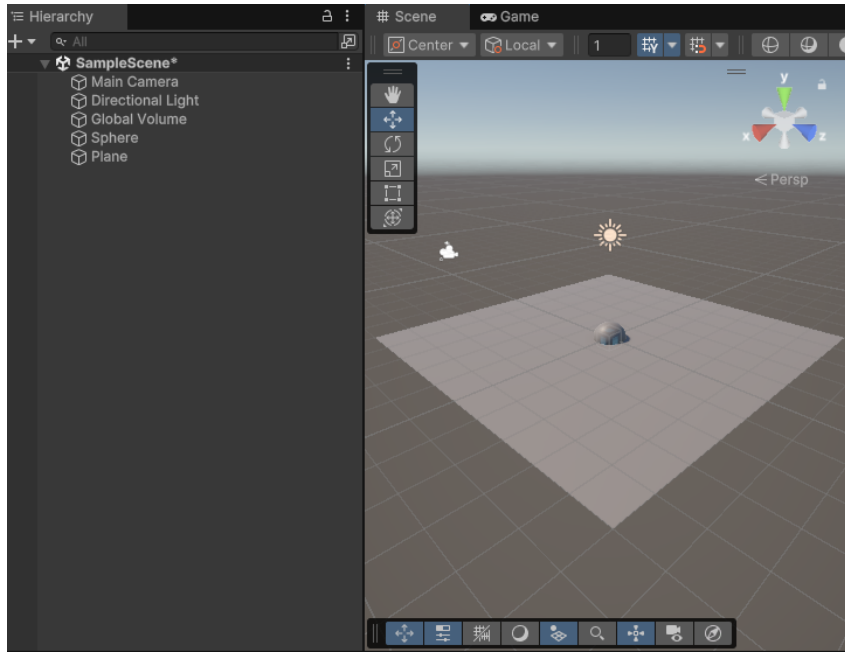
You should have your project connected to Git at this point. If you are unsure of how to do this, please refer to the [previous guide](#).

## 3.2. Creating GameObjects

To get started, we need to add GameObjects to the scene. To start with, we will create a sphere that represents our ball. **Right-Click** in the hierarchy window to open the context Menu. Select **3D Objects > Sphere**



Similarly, try creating a 3D Plane GameObject. When you do so, you will notice that these objects are overlapping.



### 3.3. Navigating the Scene View

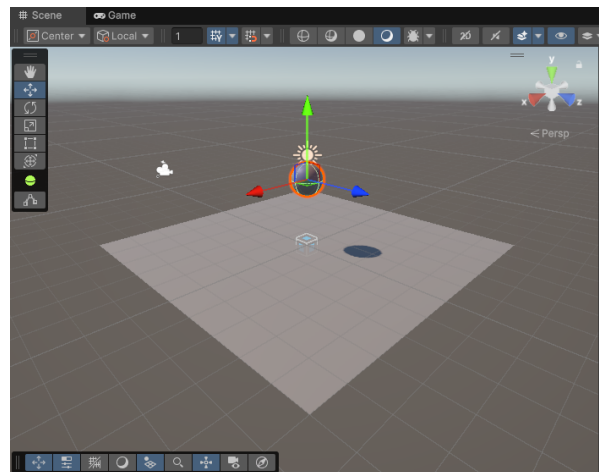
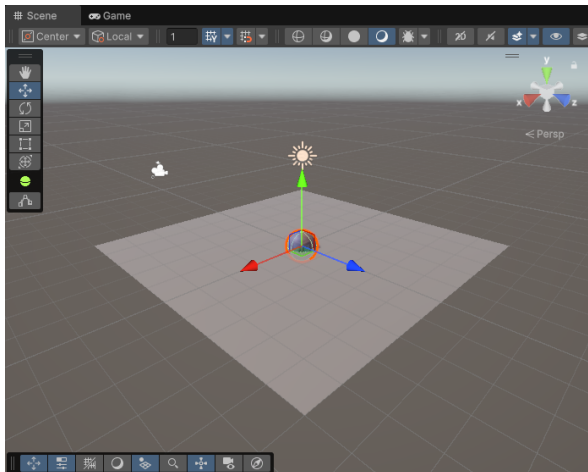
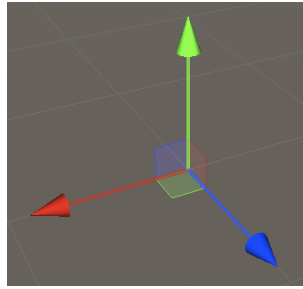
The scene view is a 3d representation of the environment. You have control over a virtual camera which shows a portion of this view. There are several ways to look around in this view:

- Scroll in and out (or pinch in and out on a trackpad) to zoom in and out.
- Press and hold the right mouse button (or two finger press on the trackpad) to look around by changing the camera angle.
- Press and hold the right mouse button (or two finger press on the trackpad) and use **W**, **A**, **S**, **D**, **Q** and **E** keys to move your camera around. W moves it forward in the direction the camera is looking, S backwards, A left and D right. Q moves it upwards and E downwards. You can look around and move around at the same time.
- Press and hold shift while doing the above to move faster.

There are a [lot of different ways](#) to move around in the scene view, all of which you will learn in due time. For now just practice moving the camera around with the above techniques. Try and keep the overlapping sphere and plane in focus.

### 3.4. Manipulating GameObjects

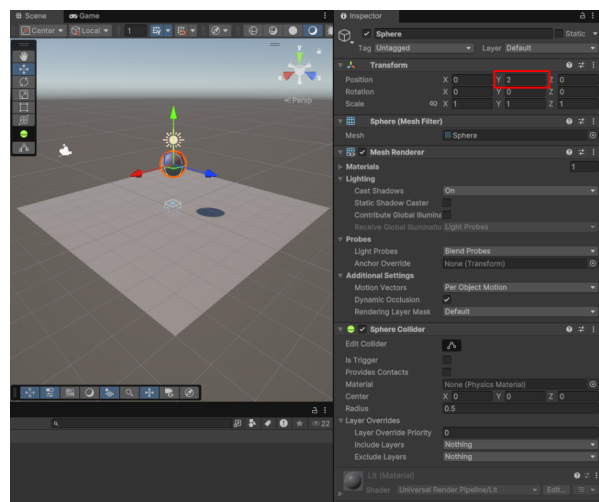
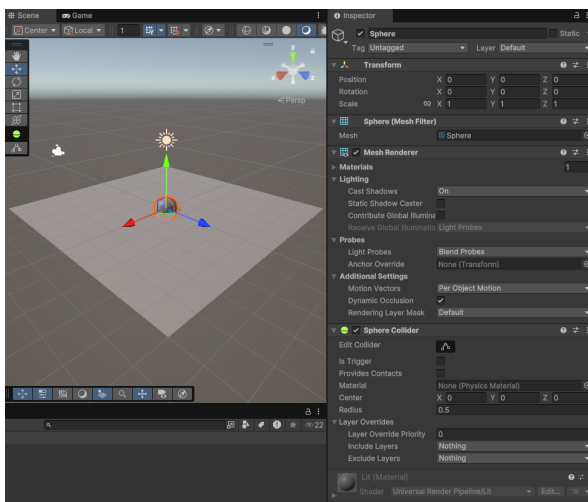
You can click on the desired object in the Scene Window (or from the Hierarchy) and manually drag the objects around by using the Red, Green And Blue Arrows on the Move Tool.



## Tip

You can also focus your scene view camera on the object by selecting it in the hierarchy, and then pressing **F** (for focus). The scene view camera should automatically adjust its view to the selected object.

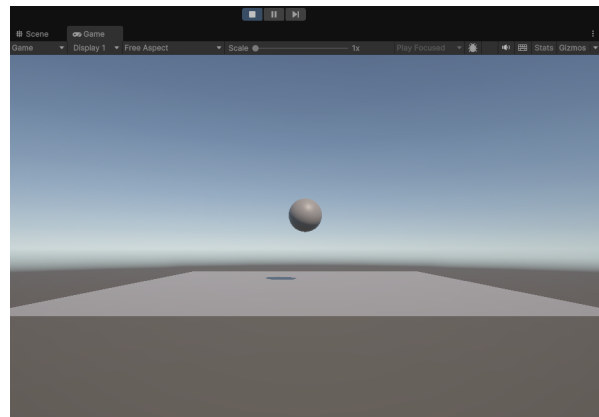
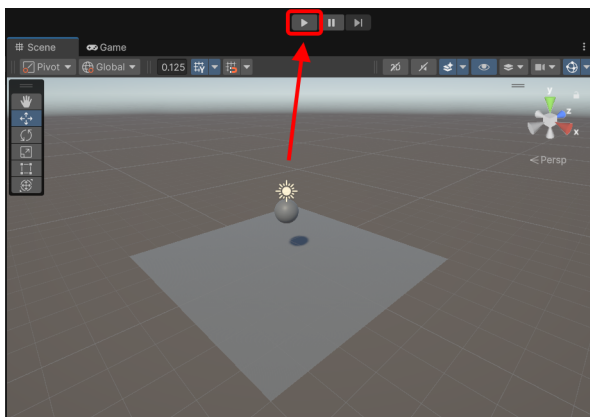
You can also change a GameObject's position using the Inspector window. When you select a GameObject, the inspector window shows all the components contained within it. Every GameObject has a default "Transform" component that manages its position, rotation, and scale in 3D space. Try changing the sphere object's position, rotation and scale using the Transform component.





### Tip

The gizmo tool in the scene view (such as the move tool with the red green and blue arrows) can be easily switched to allow you to position, rotate and scale the currently selected GameObject. When a GameObject is selected, press **W** to activate the move gizmo, **E** for the rotate gizmo and **R** for the scale gizmo. You can do this with the small toolbar on the left side of the scene view as well (second icon to move, third to rotate and fourth to scale).

To see the scene in action, press the play button on the top. Unity will automatically be focused to the game view tab. You can switch between the scene view and game view anytime (both in and out of play mode).

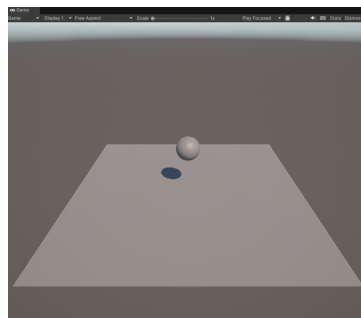


### Tip

You can always press Ctrl+P (for  / Cmd+P (for ) to switch in and out of play mode.

Notice how all objects remain static and do not respond to inputs and the sphere even floats in the air. To make objects react to key inputs and/or physics, you can either write custom logic or use Unity's built-in components.

Now try moving and rotating the camera GameObject so that we have a better game view. **Try to achieve the view below.** You can preview what the camera looks like by switching between the game view and scene view tabs before pressing play.



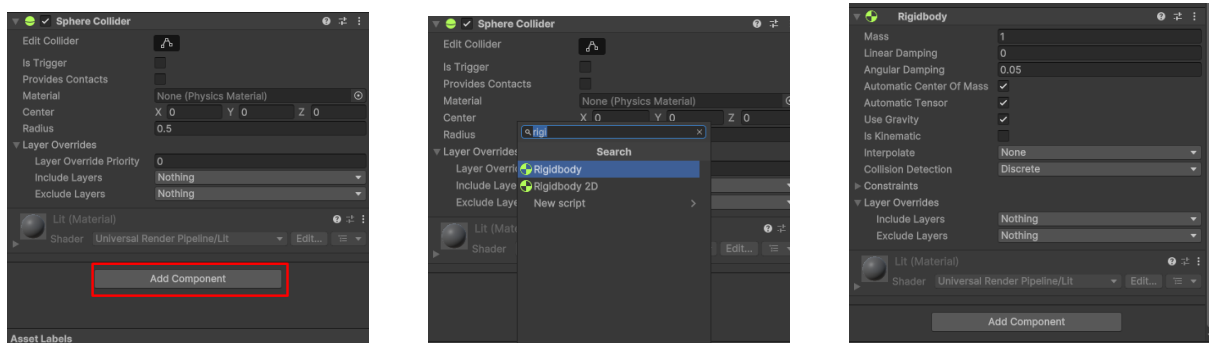


## Hint

You can select the Camera GameObject from the Hierarchy or from the gizmo icon of the camera in the scene view. Additionally, the different windows in Unity can be dragged around and placed as per your preference. You can drag out the camera window and place it somewhere else in the editor as well so that you can see the camera preview as you are editing the position in the scene view. To return to the base layout, click on the dropdown in the top right of Unity, and select the Default layout.

## 4. Introduction to Unity Physics

Dynamic objects, like our ball that need to react to things like gravity, collisions and user input, need a **Rigidbody** component. The Rigidbody lets Unity's physics engine handle movement and apply forces. These objects also need a Collider, which defines their physics shape and helps detect collisions. The physics shape can be (and often is) different from the visual mesh, for optimization, game design and/or simplicity reasons. Our sphere GameObject already contains a sphere collider. So, all we need to do is add a Rigidbody to add physics functionality



Static objects, like the ground or walls, don't move, so they only need a Collider. The Collider lets them interact with dynamic objects (eg: RigidBody) without extra calculations. This setup makes interactions look realistic while keeping the game running smoothly.

**Press the play button to see the ball react to the gravity and fall onto the plane**

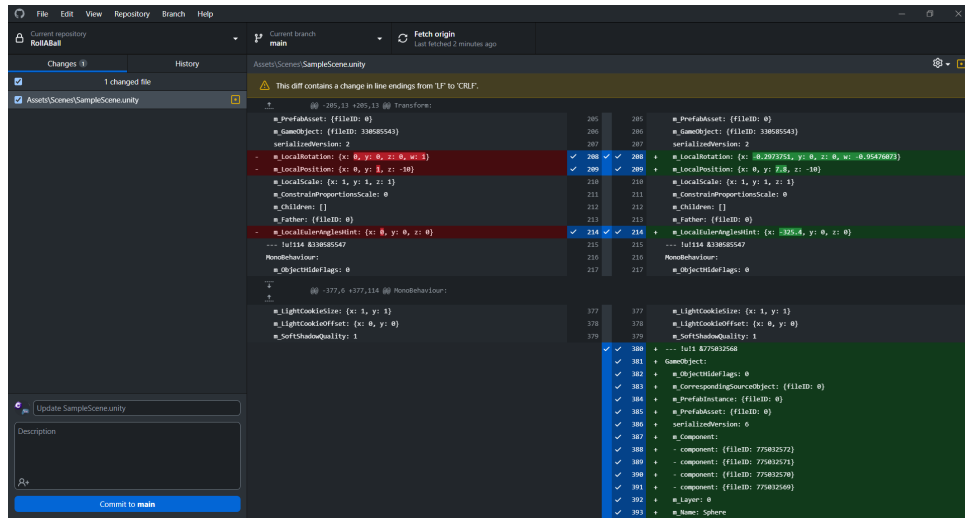
## Hint

If you don't see your ball fall, verify that the "Use gravity" option is active on the Rigidbody component. If you see the ball fall through the plane, verify the plane object has a Mesh Collider component attached to it, else add it.

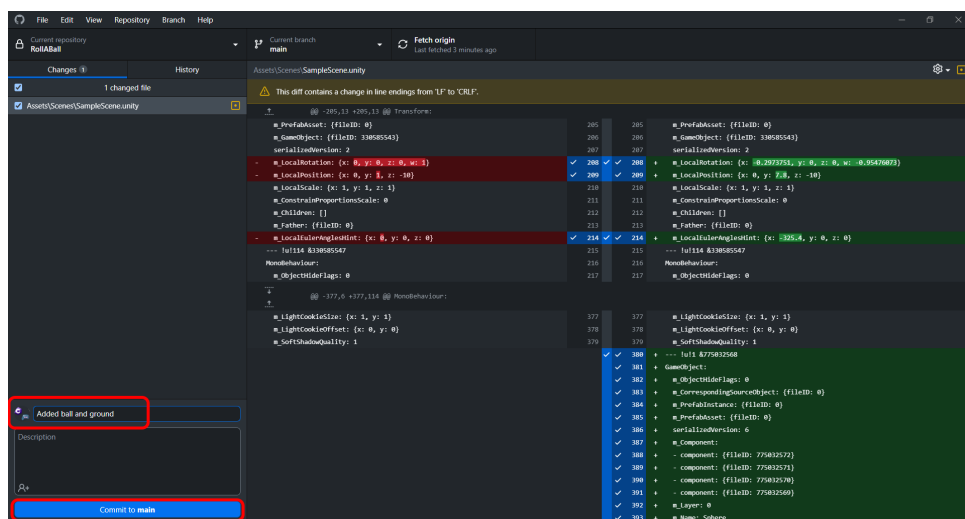
## 5. Making Git commits

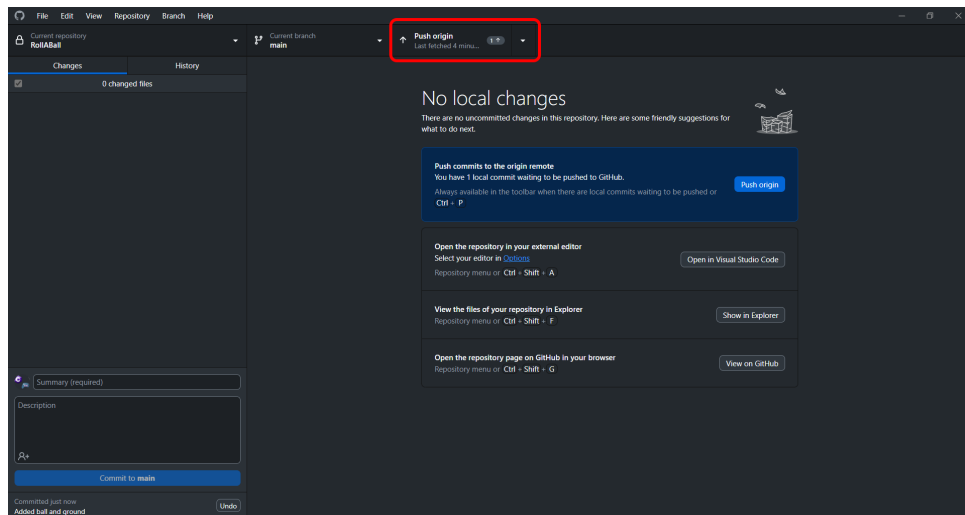
When developing, it's always good to make frequent commits as you develop features, fix bugs, or refactor code. As discussed in the earlier guide, Git commits can be treated like

save points, which create snapshots of your project. It's good to maintain these snapshots, so that you can easily recover your project if you make some destructive changes (like deleting the whole project). Now would be a good time to make a commit. Open up Github Desktop. If you have properly set up the project, then you should see the following view.



GitHub can track that you have made changes to your scene, as noted by the yellow color next to the `SampleScene.unity` file. Add a relevant commit message in the bottom left. A possible entry could be "Added ball and ground". Then click on the **Commit to main** option. This will create the save point, but only locally. Next, on the top bar, click on the **Push Changes** option. This pushes your changes to GitHub. This allows others to see your changes, and quickly fetch them, and you can clone this state of the project onto another machine with relative ease.





And that's it for making and pushing the commit.

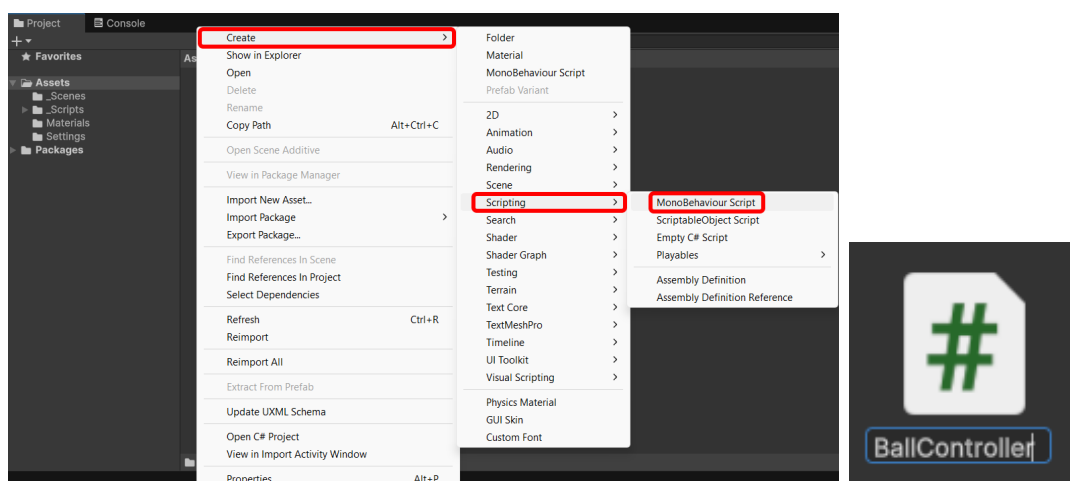
### Tip

Commit often! Even having incomplete commits is better than the fear of losing a lot of progress from the lack of commits.

## 6. Introduction To Scripting

### 6.1. Creating a custom script

Now, to make our ball reactive to user inputs, we need to create our own component that can take in user inputs. To do so, go to the project window and **right-click** to open context menu and then **Create > Scripting > MonoBehaviour Script**. Then, give the script an appropriate name like **BallController**



**Double-click** on this script to open it in your code editor.

#### Note

If you are using some other code editor like Rider, VS Code or Vim, you will have to configure Unity to launch it with that Editor. This is done from **Edit > Preferences > External Tools > External Script Editor**.

```
1 using UnityEngine;
2
3 public class BallController : MonoBehaviour
4 {
5     // Start is called once before the first execution of Update
6     // after the MonoBehaviour is created
7     void Start()
8     {
9
10    }
11
12    // Update is called once per frame
13    void Update()
14    {
15
16    }
17 }
```

Scripts that are used to control the behavior of GameObjects and typically derive from the **MonoBehaviour** class, which provides essential functionality like handling game events. Two core methods in a Unity script are the **Start** and **Update** functions.

The **Start** method is called once when the script is first initialized, making it ideal for setup tasks like initializing variables or preparing the scene. On the other hand, the **Update** method runs every frame like an infinite loop, allowing you to implement dynamic behaviors such as movement or real-time changes.

#### Tip

If you are familiar with reflection based method invocation from other languages such as Java, Javascript or Python, this is effectively what's happening under the hood for the **MonoBehavior** class with methods like **Start** and **Update**.

We will use the **Debug.Log** method to print out a console message when the **Start** and **Update** function is called. **Start** will print it once in the very beginning, and then **Update** will print it once every frame.

Modify the script as shown below in order to see these core methods in action

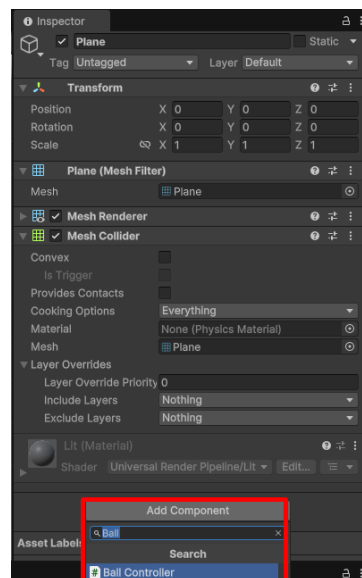
```
1 using UnityEngine;
2
3 public class BallController : MonoBehaviour
4 {
5     // Start is called once before the first execution of Update
```

```

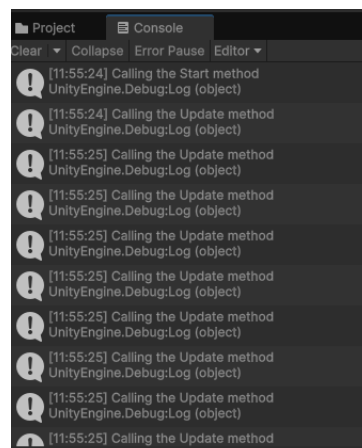
6      // after the MonoBehaviour is created
7      void Start()
8      {
9          Debug.Log("Calling the Start method");
10     }
11
12     // Update is called once per frame
13     void Update()
14     {
15         Debug.Log("Calling the Update method");
16     }
17 }

```

In order to execute the script, make sure to add the script as a component on any GameObject. For now, we will add it to the sphere from the Inspector > Add Component > BallController.



Upon pressing the play button, we can see the outputs in the console window. We can see that "Calling the Start method" is printed once while "Calling the Update method" is called every frame and is printed continuously in the console



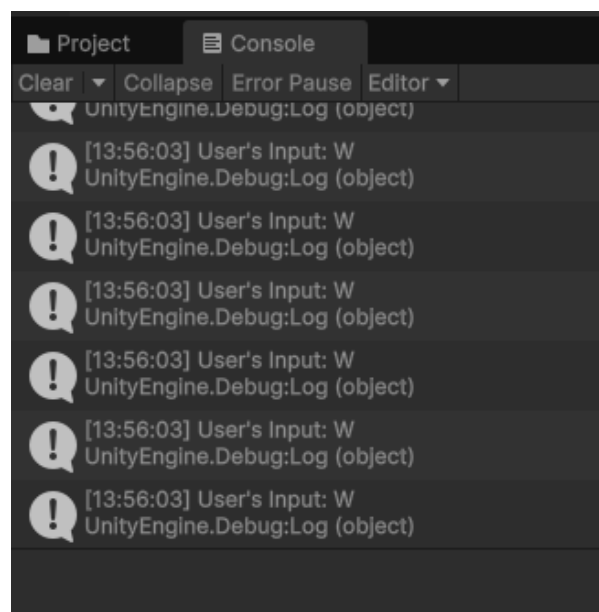
## 6.2. Registering User Inputs

Now, to take in user inputs, we will use Unity's [Input class](#).

### Note

A quick reminder, this is the old input system. Ideally for larger projects you would learn and implement the new input system. However, with the scope of this course, the older input system is easier to work with and setup. You are free to learn the new input system, but support from our side will be limited.

```
1  using UnityEngine;
2
3  public class BallController : MonoBehaviour
4  {
5      void Start()
6      {
7
8      }
9
10     // Update is called once per frame
11     void Update()
12     {
13         if (Input.GetKey(KeyCode.W))
14         {
15             Debug.Log("User's Input: W");
16         }
17     }
18 }
```



Try logging the inputs for A, S and D keys by modifying the above sample code and test it out in play mode.

### Tip

`Debug.Log` is a very useful function. Similar to other languages and their print statements, it is very often used for debugging purposes. If you're ever stuck with logic, or wondering why certain behaviors aren't being executed the way you expect it to, don't forget to make ample usage of `Debug.Log`!

## 6.3. Processing User Inputs

Now, we need to map W, A, S, D keys to Vector values to represent the corresponding directions. In most games, W is mapped to Up direction, A is mapped to Left direction, S is mapped to Down direction and D is mapped to the Right direction

We can map these keys to corresponding 2D vectors.

Key	X Axis Value	Y Axis Value	Direction
W	0	1	Up
A	-1	0	Left
S	0	-1	Down
D	1	0	Right

In Unity, we will use the [Vector2 class](#) to represent these values in Unity Scripts. Vector2 class already has a few static members that represent the required directions. We can now modify the code to log each of the directional values when the corresponding key is pressed.

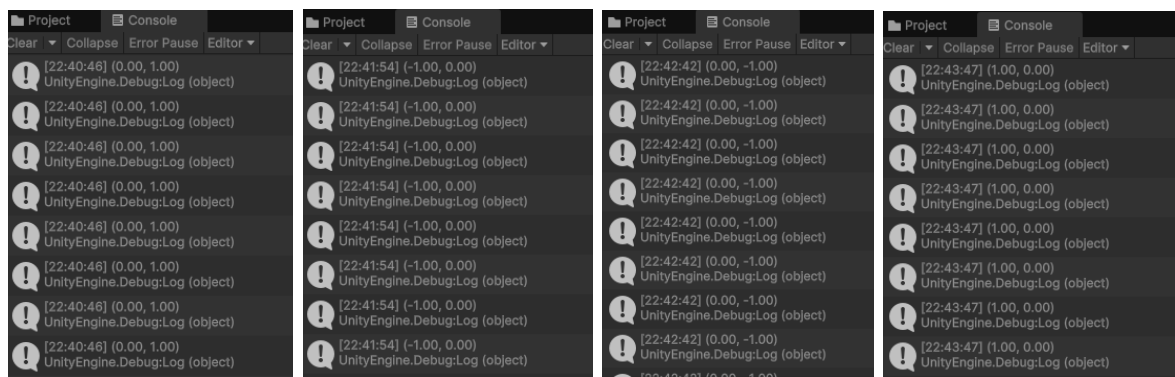
```
1 using UnityEngine;
2
3 public class BallController : MonoBehaviour
4 {
5     void Start()
6     {
7
8     }
9
10    void Update()
11    {
12        if (Input.GetKey(KeyCode.W))
13        {
14            Debug.Log(Vector2.up);
15        }
16
17        if (Input.GetKey(KeyCode.S))
18        {
19            Debug.Log(Vector2.down);
20        }
21
22        if (Input.GetKey(KeyCode.D))
23        {
```

```

24         Debug.Log(Vector2.right);
25     }
26
27     if (Input.GetKey(KeyCode.A))
28     {
29         Debug.Log(Vector2.left);
30     }
31 }
32

```

This code snippet will give the following outputs in the console when the corresponding keys are pressed.



W

A

S

D

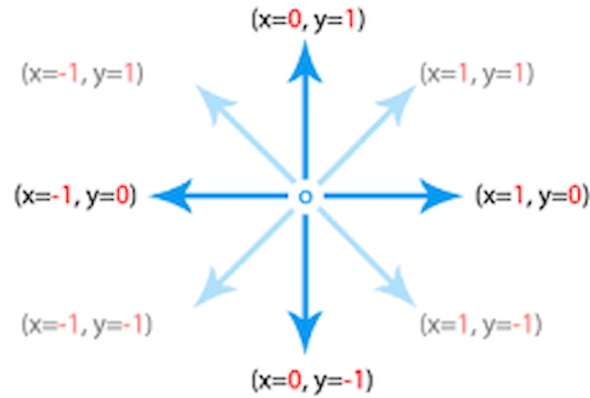
To calculate the resultant vector for movement in Unity when multiple keys are pressed, you can combine their directional inputs. The W and S keys typically control vertical movement, with W adding a positive value (up) and S adding a negative value (down). Similarly, the A and D keys control horizontal movement, with A subtracting from the horizontal axis (left) and D adding to it (right).

By summing these inputs, the resultant vector reflects the combined direction. For example:

- Pressing W (+1) and S (-1) cancels out to a vertical value of 0.
- Pressing A (-1) and D (+1) cancels out to a horizontal value of 0.
- Combining diagonal inputs like W (+1 vertical) and D (+1 horizontal) results in a vector of (1, 1), enabling movement toward the top-right.

This approach supports 8-directional movement, allowing for smoother and more precise control when multiple keys are pressed simultaneously. To implement this functionality we can create a Vector2 variable within the Update() function. The idea is to check for key presses and add the respective vector components (x and y) to the variable.





```

1  using UnityEngine;
2
3  public class BallController : MonoBehaviour
4  {
5      void Start()
6      {
7
8      }
9
10     void Update()
11     {
12         Vector2 inputVector = Vector2.zero; // initialize our input vector
13         if (Input.GetKey(KeyCode.W))
14         {
15             inputVector += Vector2.up; // "a += b" <=> "a = a + b"
16         }
17
18         if (Input.GetKey(KeyCode.S))
19         {
20             inputVector += Vector2.down;
21         }
22
23         if (Input.GetKey(KeyCode.D))
24         {
25             inputVector += Vector2.right;
26         }
27
28         if (Input.GetKey(KeyCode.A))
29         {
30             inputVector += Vector2.left;
31         }
32
33         Debug.Log("Resultant Vector: " + inputVector);
34     }
35 }

```

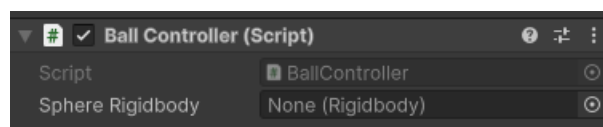
Press the play button and test it out with different combinations of W, A, S and D.

## 6.4. Interfacing with other components

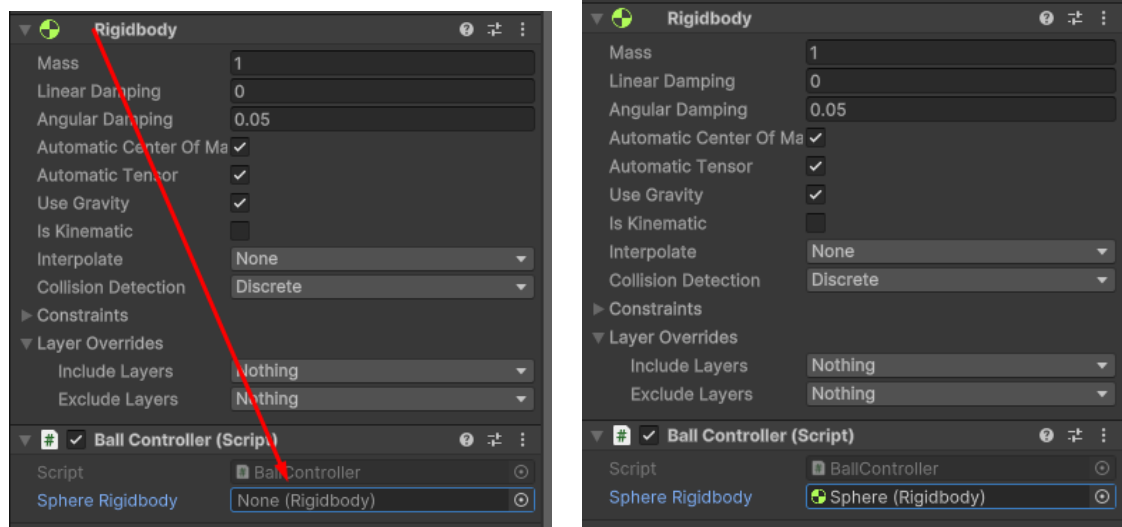
Now, in order to control the ball using these inputs, we need to add a force in the corresponding direction. We can do that by using the [Rigidbody.Addforces](#) function. To use this, we first need to access the Rigidbody component attached to the sphere GameObject. Start by declaring a public variable of type Rigidbody, like this:

```
1  using UnityEngine;
2
3  public class BallController : MonoBehaviour
4  {
5      public Rigidbody sphereRigidbody;
6
7      void Start()
8      {
9
10     }
11
12     void Update()
13     {
14         .
15         .
16         .
17
18         Debug.Log("Resultant Vector: " + inputVector);
19     }
20 }
```

Now if we look in the inspector for the sphere GameObject, the BallController component has an exposed field with the name "Sphere Rigidbody".

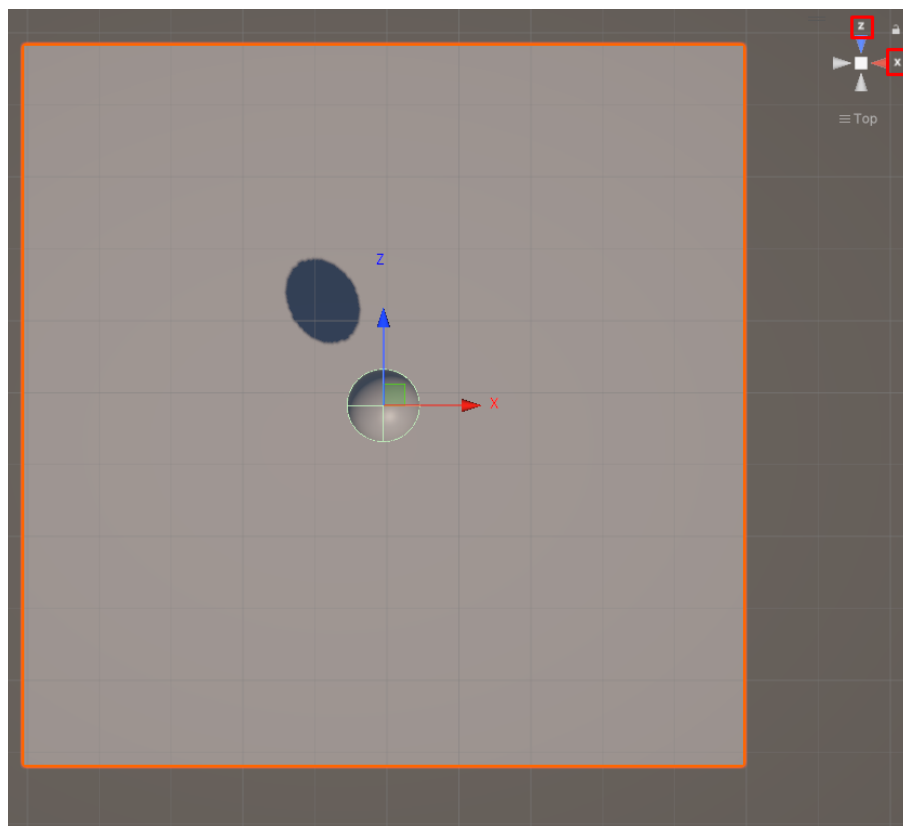


We can drag and drop our Rigidbody component that is attached on this GameObject.



This lets our `BallController` script know which instance of `RigidBody` to control. In order to apply a force, we can simply call the built-in `AddForce` function on the `sphereRigidBody`. The function takes in the target vector to produce the force accordingly.

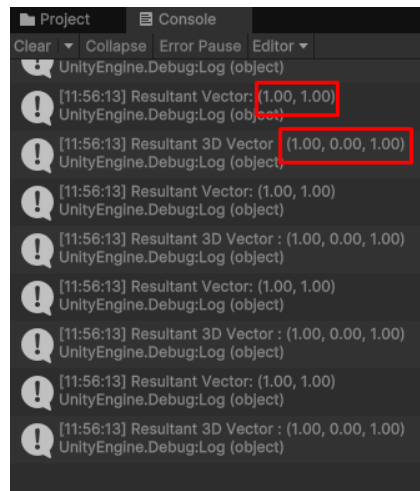
Our input is a 2D vector, but movement in the scene occurs in 3D space. Since we want the movement to happen on the X and Z plane, we need to convert the 2D input vector (XY plane) into a 3D vector.



To do this, we can create a `Vector3` variable inside the `Update` loop and assign the input's X value to the X axis and the Y value to the Z axis of the new vector. This allows the

input to control movement on the desired plane.

```
1  using UnityEngine;
2
3  public class BallController : MonoBehaviour
4  {
5      public Rigidbody sphereRigidbody;
6
7      void Start()
8      {
9
10     }
11
12     void Update()
13     {
14         .
15         .
16         Vector3 inputXZPlane = new Vector3(inputVector.x, 0, inputVector.y);
17         Debug.Log("Resultant Vector: " + inputVector);
18         Debug.Log("Resultant 3D Vector: " + inputVector);
19     }
20 }
```



Now all we need to do is use the **AddForce** function on the **sphereRigidbody**.

```
1  using UnityEngine;
2
3  public class BallController : MonoBehaviour
4  {
5      public Rigidbody sphereRigidbody;
6      void Start()
7      {
8
```

```

9      }
10
11      void Update()
12      {
13          .
14          .
15          .
16          Vector3 inputXZPlane = new(inputVector.x, 0, inputVector.y);
17          sphereRigidbody.AddForce(inputXZPlane);
18      }
19  }

```

Press the play button and try controlling the sphere.

#### Hint

Keep an eye on the console. If you get a `NullReferenceException` about not being able to find the sphere `Rigidbody`, you may not have attached the component reference. See the start of 6.4

Ideally, we would also want how fast or slow the ball moves on plane. We can simply add float variable named `ballSpeed`. We can then multiply our `inputXZPlane` with the float to increase the magnitude of force applied.

```

1  using UnityEngine;
2
3  public class BallController : MonoBehaviour
4  {
5      public Rigidbody sphereRigidbody;
6      public float ballSpeed = 2f;
7      void Start()
8      {
9
10     }
11
12     void Update()
13     {
14         .
15         .
16         .
17         Vector3 inputXZPlane = new(inputVector.x, 0, inputVector.y);
18         sphereRigidbody.AddForce(inputXZPlane * ballSpeed);
19     }
20 }

```

#### Tip

This would be a great time to create and push a commit. Something along the lines of "Added ball movement" should suffice.

## 6.5. Good Coding Practices (Optional)

This is an **optional section** of the tutorial introduces good coding practices that are not only relevant to Unity development but can also be applied across various tech stacks. Following these practices can help improve code readability, maintainability, and efficiency, making your projects more robust and easier to manage in the long term.

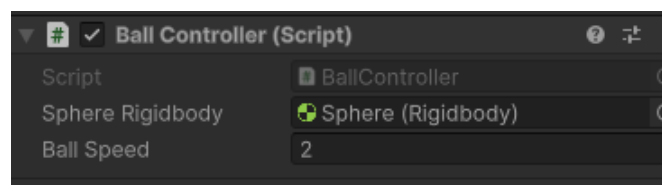
If this section seems too advanced or complex, you can skip to section 7.

### 6.5.1 Private Variables and SerializedFields

It is a good coding practice to keep variables private and control their access using getters and setters. This prevents unintended access or modification of the data, protecting the integrity of the object. However, if we make a variable private in Unity, they are no longer exposed in the inspector. **Try to make the sphereRigidbody variable private and check the inspector.**

Unity allows for private variables to be exposed in the inspector by simply adding a `[SerializeField]` attribute

```
1  using UnityEngine;
2
3  public class BallController : MonoBehaviour
4  {
5      [SerializeField] private Rigidbody sphereRigidbody;
6      [SerializeField] private float ballSpeed = 2f;
7
8      .
9      .
10     .
11 }
```



### 6.5.2 Single Responsibility Principle

The Single Responsibility Principle (SRP) is one of the five principles of object-oriented design commonly referred to as SOLID principles. It states that a class should have only one reason to change, meaning it should have only one responsibility or purpose. This principle promotes a modular and maintainable code structure.

Currently, our `BallController` script handles both, inputs and adding force. However, if we want to use the inputs to control another entity, such as a humanoid character, we would have to process the inputs differently. To address this, we will separate the input handling into a new script called `InputManager`. Using [UnityEvents](#), we will link the `InputManager` with the `BallController` script, allowing it to respond to input events

efficiently.

Follow the steps in section 6.1 to create an InputManager script. In this script add the namespace `UnityEngine.Events`.

```
1 using UnityEngine;
2 using UnityEngine.Events;
3
4 public class InputManager : MonoBehaviour
5 {
6     .
7     .
8 }
```

Next, we're going to add a variable of type `UnityEvent<>`. The `<>` brackets signifies that its of a generic type. This allows us to pass arguments through the event, and those arguments can be of any type. Since our input will be a `Vector2`, we will define a `UnityEvent<Vector2>` specifically for this type.

Simultaneously, we will also copy the logic from the BallController script to process the inputs from the update loop

```
1 using UnityEngine;
2 using UnityEngine.Events;
3
4 public class InputManager : MonoBehaviour
5 {
6     public UnityEvent<Vector2> OnMove = new UnityEvent<Vector2>();
7
8     void Update()
9     {
10         Vector2 inputVector = Vector2.zero;
11         if (Input.GetKey(KeyCode.W))
12         {
13             inputVector += Vector2.up;
14         }
15         if (Input.GetKey(KeyCode.S))
16         {
17             inputVector += Vector2.down;
18         }
19         if (Input.GetKey(KeyCode.A))
20         {
21             inputVector += Vector2.left;
22         }
23         if (Input.GetKey(KeyCode.D))
24         {
25             inputVector += Vector2.right;
26         }
27     }
28 }
```

Now we are going to send the data using this UnityEvent every frame using the Invoke function.

```
1 using UnityEngine;
2 using UnityEngine.Events;
3
4 public class InputManager : MonoBehaviour
5 {
6     public UnityEvent<Vector2> OnMove = new UnityEvent<Vector2>();
7
8     void Update()
9     {
10         Vector2 inputVector = Vector2.zero;
11         if (Input.GetKey(KeyCode.W))
12         {
13             inputVector += Vector2.up;
14         }
15         .
16         .
17
18         OnMove?.Invoke(inputVector);
19     }
20 }
```

#### Tip

The ? in `OnMove?.Invoke` implies that if there are no listeners for this event, then the compiler will safely handle it. Without the question mark, if there are no listeners, you will get a null reference exception.

Our InputManager script is now ready. Attach this script to our sphere object as shown in 6.1.

We now need to add our BallController script as a listener to this class. Remove the existing update loop from the ball controller script and create a new function called "MoveBall" with a parameter of type Vector2. We are going to transpose the 2D vector to a 3D vector and apply the force on rigidbody.

**Make sure that MoveBall function is public as it needs to be added as a listener to our event**

```
1 using UnityEngine;
2
3 public class BallController : MonoBehaviour
4 {
5     [SerializeField] private Rigidbody sphereRigidbody;
6     [SerializeField] float ballSpeed;
7
8     public void MoveBall(Vector2 input)
```

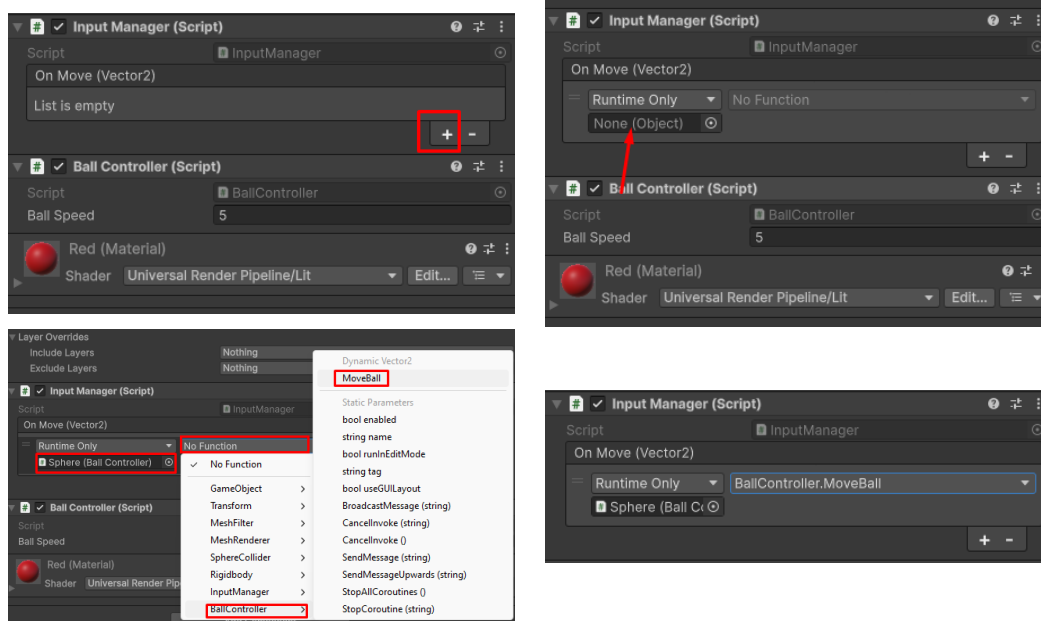


```

9      {
10         Vector3 inputXZPlane = new(input.x, 0, input.y);
11         sphereRigidbody.AddForce(inputXZPlane * ballSpeed);
12     }
13 }

```

To receive the input data, we can connect the InputManager through the Inspector on the sphere GameObject



Press the play button and test it out

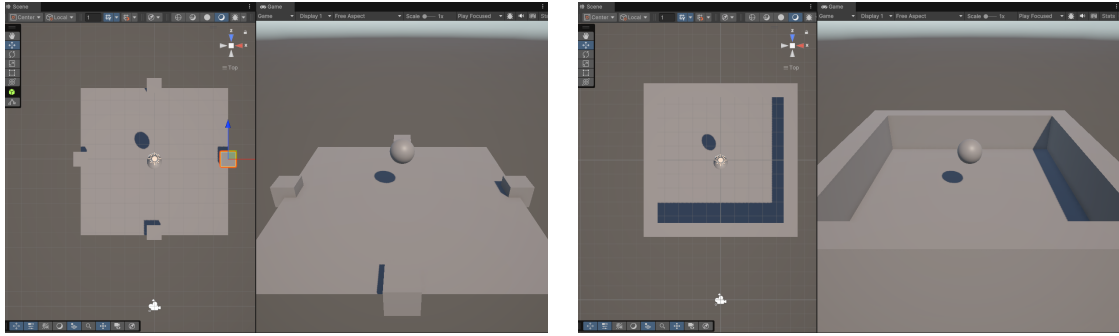
### Tip

This would be a great time to create and push a commit. Something along the lines of "Refactored input handling" should suffice.

## 7. Polish

### 7.1. Adding Walls

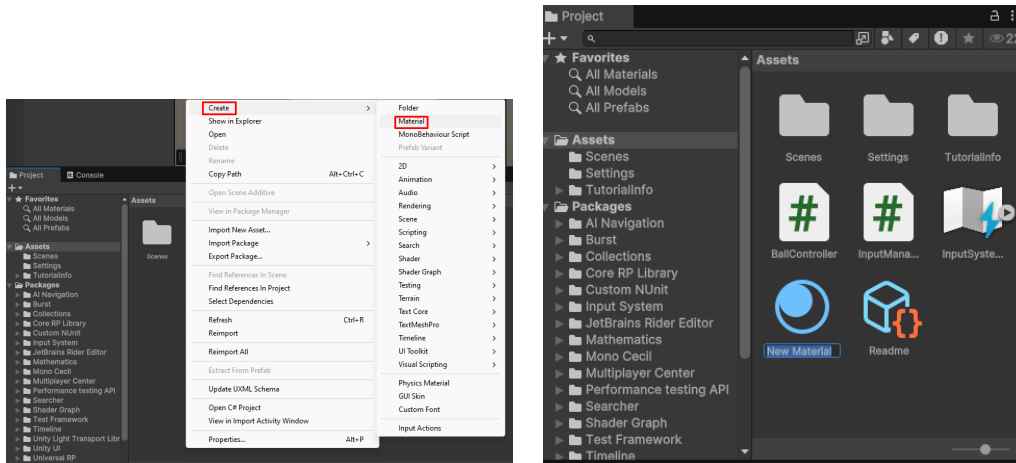
Add 4 cubes to our scene by following the steps similar to the Section 3.2. Now move all the cubes such that they are sitting at each edge of our plane. Then adjust their scales in X, Y and Z from the inspector so that they cover the entire edge of the plane and do not allow the ball to jump over.



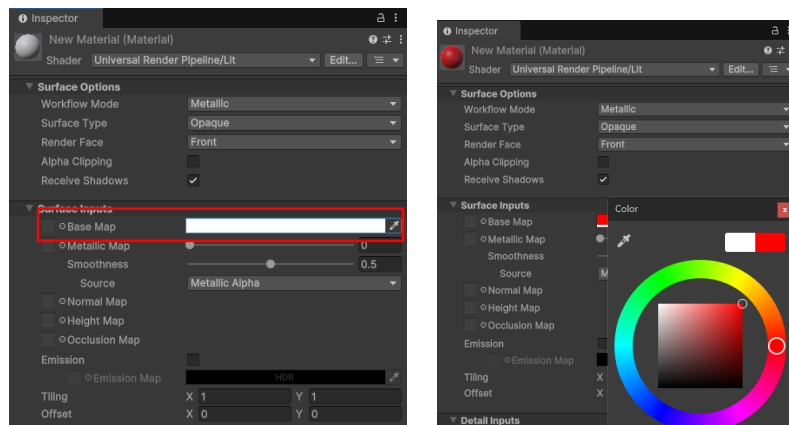
## 7.2. Adding Materials

Currently, our roll a ball looks very plain and boring. To make it look better we can add colors to our ball and the walls.

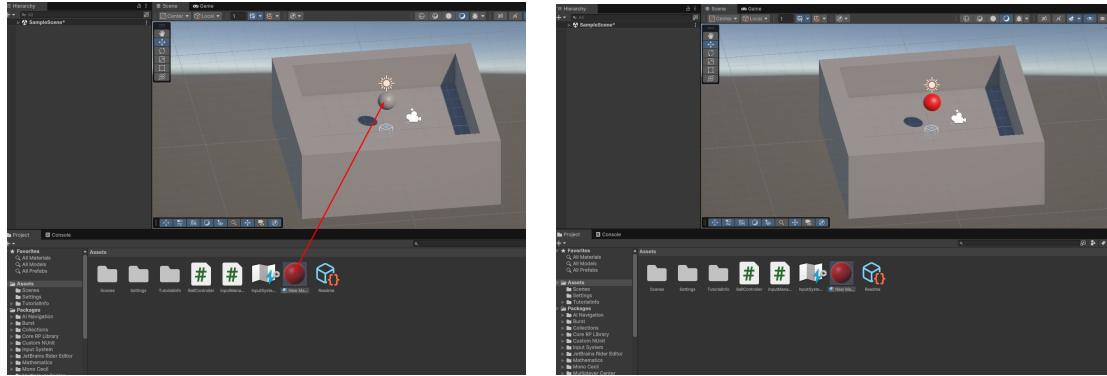
To do so, we will create a "Material". A material allows us to apply textures and colors to 3D objects. Go to Project Tab > Right Click > Context menu > Create > Material.



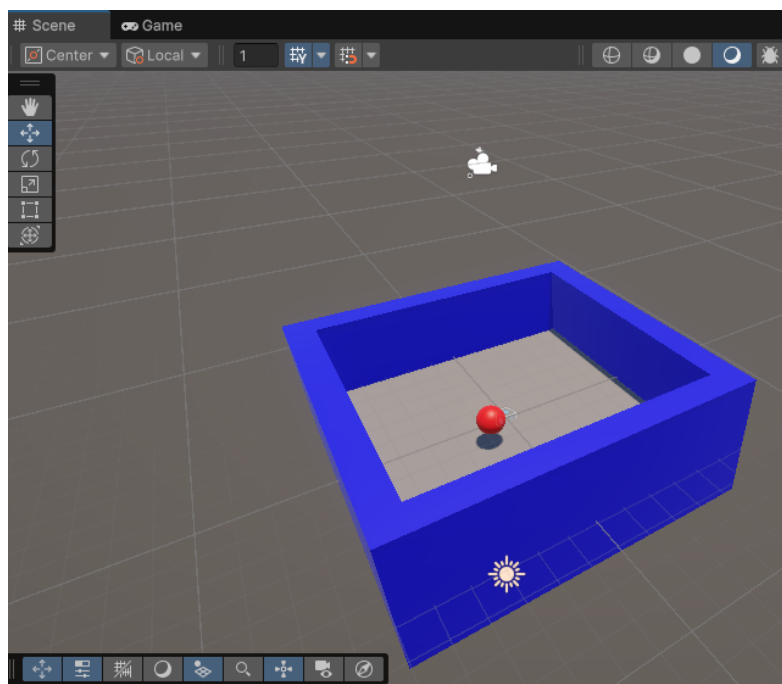
Now we can change the color of the material through the inspector window. Click on the material to reveal its inspector. Then click on the BaseMap property to assign a color to it using a color wheel.



Now Simply drag and drop the material on the desired object in the Scene View.



Repeat the steps for each of the walls and plane by creating different materials.



### Tip

This would be a great time to create and push a commit. Something along the lines of "Set up environment" should suffice.

## 8. Conclusion

This concludes our tutorial on RollABall. You should now have a decent high level overview of the Unity game engine. For the assignment submission, your deliverable will be the GitHub link to this starter project. Future assignments will showcase similar guides, and then expect you to reapply those skills to a related but different task.

### 8.1. Bonus

If you've finished this tutorial, and are itching to learn more early, here's a bonus assignment: **Make the ball jump**. Now if you have followed this guide properly, this might

seem trivial at first glance. You can detect a button press (such as spacebar), and then add a force in the y-axis. But the issue is the player will be able to press space multiple times, launching the ball upward constantly. You will need some way to be able to *detect if the ball is touching the ground...*

#### Hint

To implement this feature before the next assignment, you will have to search up unity guides on your own. There are two possible ways you can implement this feature: detecting collisions between the ball and the ground, or making use of raycasts. Both are viable options, and should be easy to search and find out about. Good luck!