Bowling with Physics

Rishav, Omang

January 29, 2025

Contents

---

1. Introduction

Welcome to your second unity tutorial! In this guide, we will mainly be covering more physics related concepts and discuss a little about score-keeping with game manager, and UI elements. This guide requires you to have completed the Roll-a-Ball guide first. If you haven't done that, please finish that first. You may need to reference that guide as

well for concepts you may have forgotten, so it is advised to keep that guide handy while following this one.

For this project, we will be making a simple bowling game, which has a player aim and fire a ball at some pins (Demo Video). We will be able to see the total accumulated score and reset the pins and balls. Before you get started, make a new Unity Project following the instructions on New Unity Project Setup guide, and name the project and the git repo appropriately.
Same as last time, you will be expected to replicate this project in its entirety. This one will be significantly longer and more complex, so read through sections multiple times and make sure to scour unity documentation and external online resources (along with asking doubts in the TA hours!)

2. Base Game Setup

We'll start with getting a ball rolling and hitting a collection of pins which fall over.

2.1. Initial Prototype

First, create a sphere and a plane in the current scene. Set up the sphere to be a Rigidbody similar to the initial Roll-A-Ball tutorial (section 4).
We will create two scripts

1. `InputManager.cs` that handles all of our inputs through UnityEvents.
2. `BallController.cs` that is responsible for reacting to input manager and control the ball

Hint

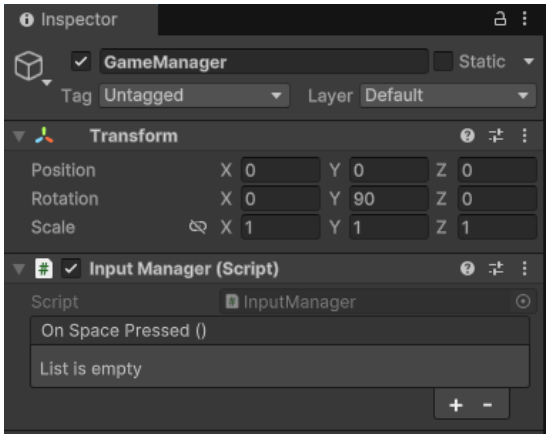
Separating input handling from BallController enables the reuse of the InputManager class in different parts of the project, enhancing modularity. This approach follows the Single Responsibility Principle from the SOLID design principles, ensuring each class has a distinct and focused role.

Now, we'll create `InputManager.cs` that handles all of our inputs through UnityEvents. This script has also been extensively discussed in section 6.5.2 of the Roll-A-Ball tutorial. For now, we just want to listen to the space key in order to launch the ball.

```
1  using System;
2  using UnityEngine;
3  using UnityEngine.Events;
4
5  public class InputManager : MonoBehaviour
6  {
7      public UnityEvent OnSpacePressed = new UnityEvent();
8      void Update()
9      {
10         if (Input.GetKeyDown(KeyCode.Space))
```

```
11          {
12              OnSpacePressed?.Invoke();
13          }
14      }
15  }
16
```

This script listens for the Space key press and triggers the OnSpacePressed event when detected. `Invoke()` calls all functions that have been subscribed to this event,
Make a new empty GameObject called `GameManager` and attach the `InputHandler` script.

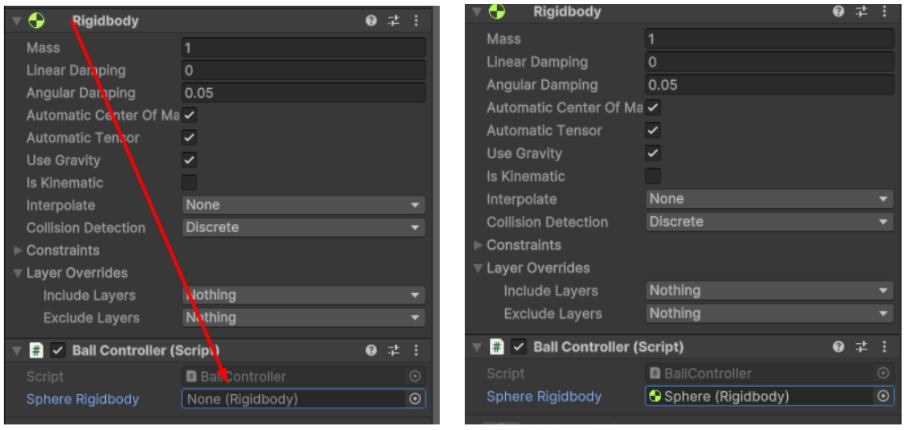


To make the ball respond to input events, we will create a script named `BallController.cs`. This script listens for the `OnSpacePressed` event from the `InputManager` and applies a force to the ball when the event is triggered.

Note

This style of programming, called the "Observer Pattern," uses events to manage the flow of logic. Normally, we call functions directly, but this can cause issues if the function belongs to an object that doesn't exist or if multiple unrelated functions need to be called, cluttering your script. With the Observer Pattern, an event is "fired", and listeners can choose to "subscribe" to it. Think of it like the code shouting, "Hey, something happened!" Other parts of the code can listen and respond if needed. If a listener is removed, the code still works fine, and multiple listeners can handle the same event. Listeners can also subscribe to events from multiple sources.
There are a lot of other programming patterns that may be useful in designing different systems. You can read about other patterns here

Like in the Roll-A-Ball tutorial, we need to control a RigidBody. Previously, we manually assigned a reference by dragging it in the editor like so:



However, we can script this process using `GetComponent<T>()`, which retrieves components attached on the same GameObject. In our `BallController`, we can call `GetComponent<Rigidbody>()` within the `Start()` function to get the reference automatically.

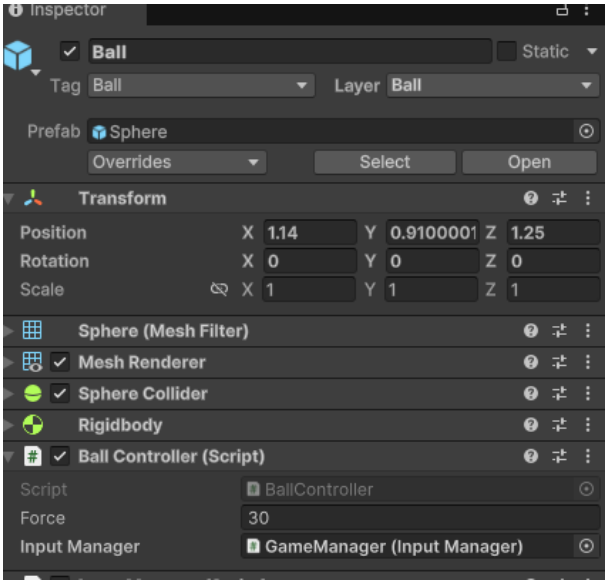
Calling `GetComponent<Rigidbody>()` in the `Start()` function ensures the reference is set before any gameplay logic runs. This approach prevents manual setup errors via the Unity Editor. However, manually referencing via dragging and dropping is also perfectly valid.

```
using UnityEngine;
using UnityEngine.Events;

public class BallController : MonoBehaviour
{
    [SerializeField] private float force = 1f;
    [SerializeField] private InputManager inputManager;

    private Rigidbody ballRB;

    void Start()
    {
        //Grabbing a reference to RigidBody
        ballRB = GetComponent<Rigidbody>();

        // Add a listener to the OnSpacePressed event.
        // When the space key is pressed the
        // LaunchBall method will be called.
        inputManager.OnSpacePressed.AddListener(LaunchBall);
    }


    private void LaunchBall()
    {
        ballRB.AddForce(transform.forward * force, ForceMode.Impulse);
    }
```

```
}

```

AddForce() takes two parameters: the force direction and the force application method. In this tutorial, we'll use ForceMode.Impulse, which applies an instant force change. However, there are three other modes—Force (applied over a frame), Acceleration (force applied independent of mass), and VelocityChange (just updates the velocity). Experiment with different modes to see which feels good to you.

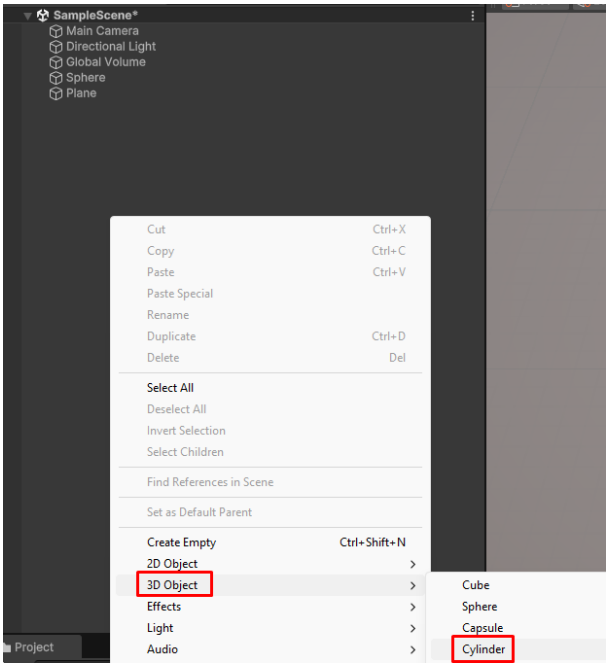
Attach the `BallController` to a Sphere GameObject similar to how we created our ball in Roll-A-Ball. Make sure to assign the InputManager Reference.



Press play to try it out.

> **Tip**
>
> This is the same as creating a new Unity Event in the Unity Editor via the Inspector, and linking the relevant script's function as shown towards the end of Section 6.5.2 of the Roll-A-Ball assignment.

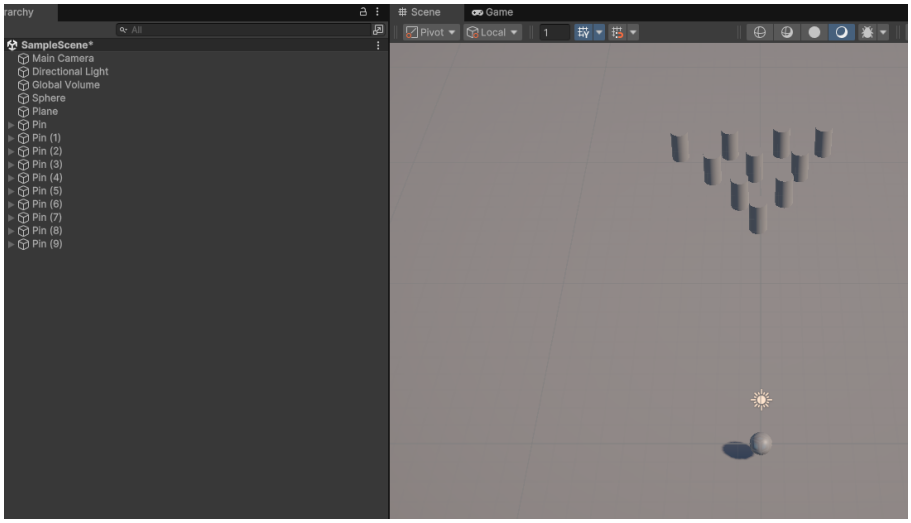
Now we'll add a cylinder GameObject to represent our pin. You can add cylinders by going to `Hierarchy > Right Click > 3D object > Cylinder`. **Remember to manually add a RigidBody to this GameObject.**
Create 10 duplicates of the Cylinder GameObject and position them in a triangular formation, resembling bowling pins in a standard bowling setup.



Tip

You can click on a GameObject and press Ctrl + D (for) or Cmd + D (for) to duplicate it quickly.

You can arrange your pins to resemble how bowling pins are laid out in a bowling match.



Make sure the forward orientation of the ball (the blue line in the move gizmo) is towards the pins you just set up using the rotation fields in the transform component.

Press play to try it out.

This implementation works well enough, but there is a glaring issue in our code. The user can press the space key multiple times to trigger the launch function multiple times. To prevent this from happening we will add a `isBallLaunched` check to our code

```
1 using UnityEngine;
2 using UnityEngine.Events;
```

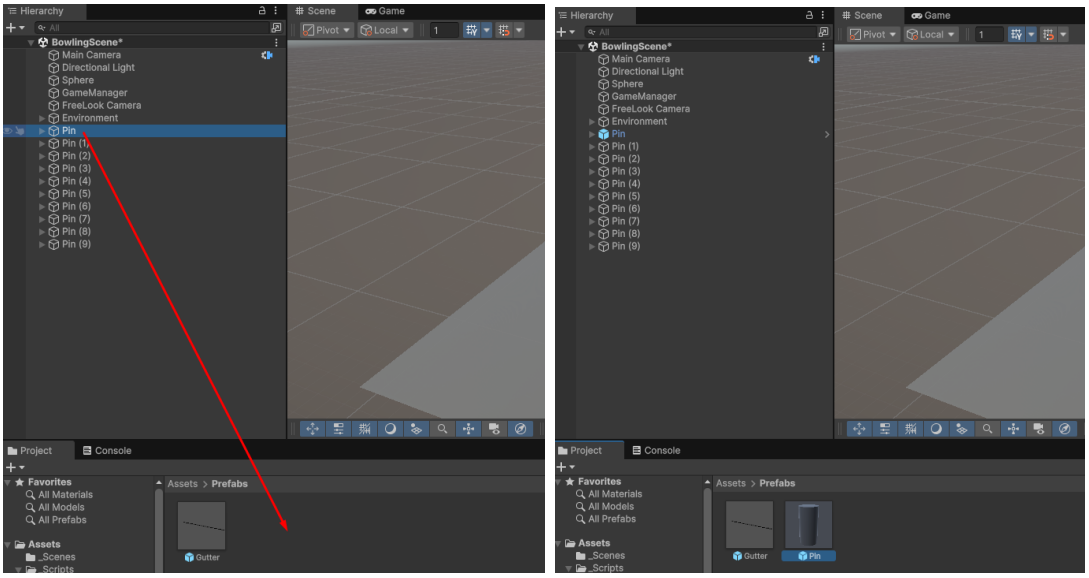
```
3
4  public class BallController : MonoBehaviour
5  {
6      [SerializeField] private float force = 1f;
7
8      // name booleans like a question
9      private bool isBallLaunched;
10     private Rigidbody ballRB;
11     private InputManager inputManager;
12
13     void Start()
14     {
15         ballRB = GetComponent<Rigidbody>();
16         inputManager.OnSpacePressed.AddListener(LaunchBall);
17     }
18
19
20     private void LaunchBall()
21     {
22         // now your if check can be framed like a sentence
23         // "if ball is launched, then simply return and do nothing"
24         if (isBallLaunched) return;
25         // "now that the ball is not launched, set it to true and launch the ball"
26         isBallLaunched = true;
27         ballRB.AddForce(transform.forward * force, ForceMode.Impulse);
28     }
29 }
```

2.2. Prefabs

Since, we have 10 pins in our scene, and all of them need to behave in a consistent way, manually adjusting their properties one by one will become a tedious task. To streamline this process, Unity has a feature to create a reusable template of a GameObject called **Prefabs**.

This template can then be instantiated multiple times in a scene or during runtime. Prefabs are especially valuable for creating and managing replicated objects, such as enemies, obstacles, items, or, in this case, bowling pins.

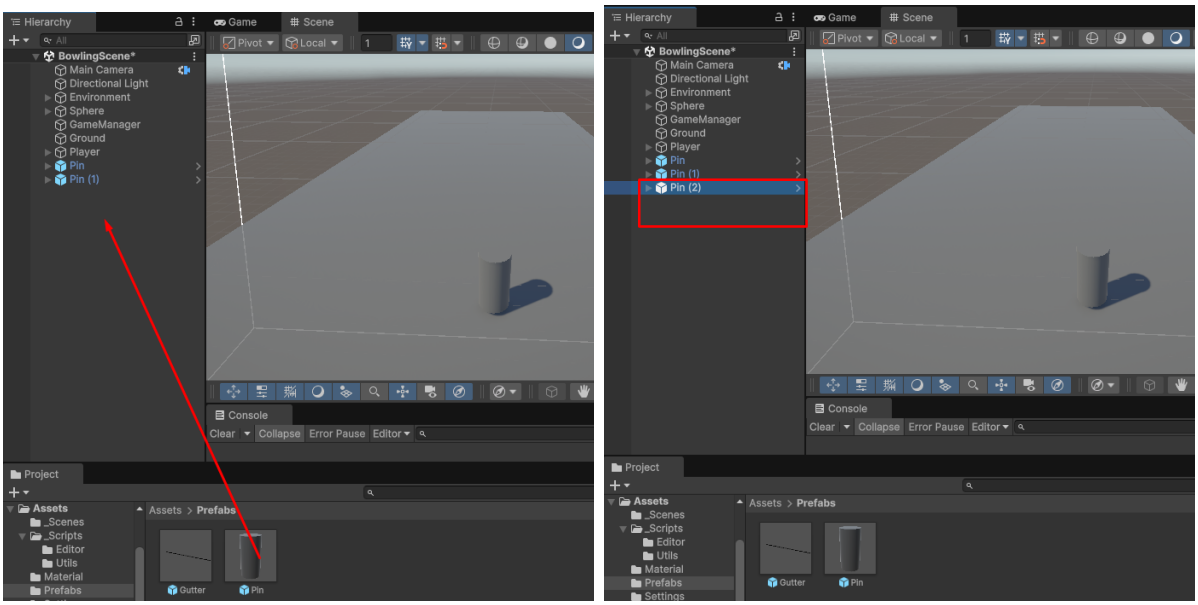
To create a prefab, simply drag one of the pin objects from the Hierarchy Window into the project window.



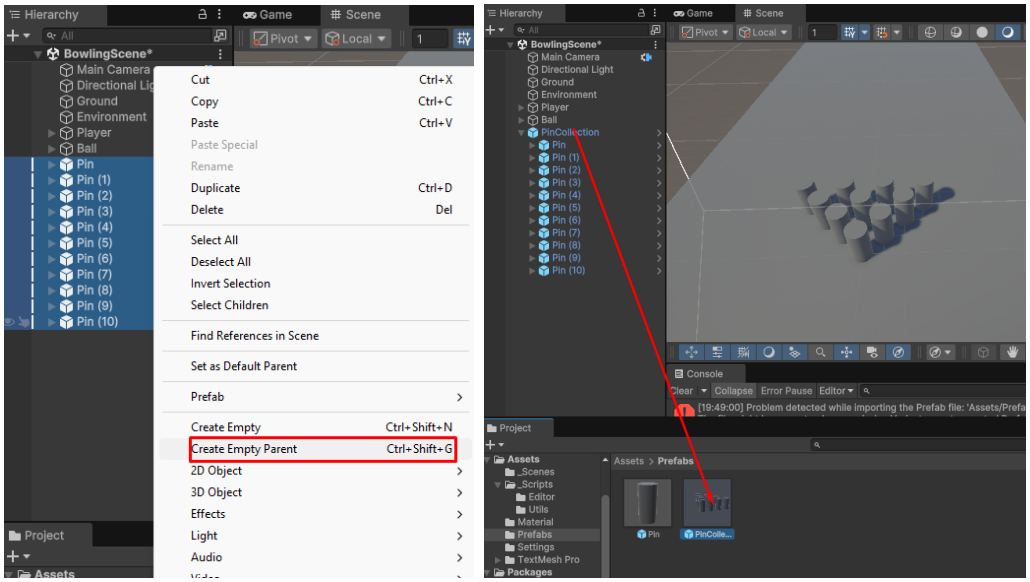
Replace all Pin GameObjects with the Pin prefab by dragging it from the Project tab into the Hierarchy window ten times to generate ten identical clones. Once added, arrange them to replicate the layout of bowling pins.

Hint

Prefab stands for prefabricated object. It's a blueprint of sorts for Unity GameObjects. You only need one prefab per object.



Unity also allows nesting prefabs, meaning you can create a prefab that contains other prefabs. To do this, group your pins under a single parent prefab. Start by selecting all the pins in the Hierarchy, then right-click and choose Create Empty Parent. Once the empty parent object is created, simply drag it into the Project tab, just like any other GameObject, to create a "Prefab of Prefabs." Prefabs can be nested infinitely!



Tip

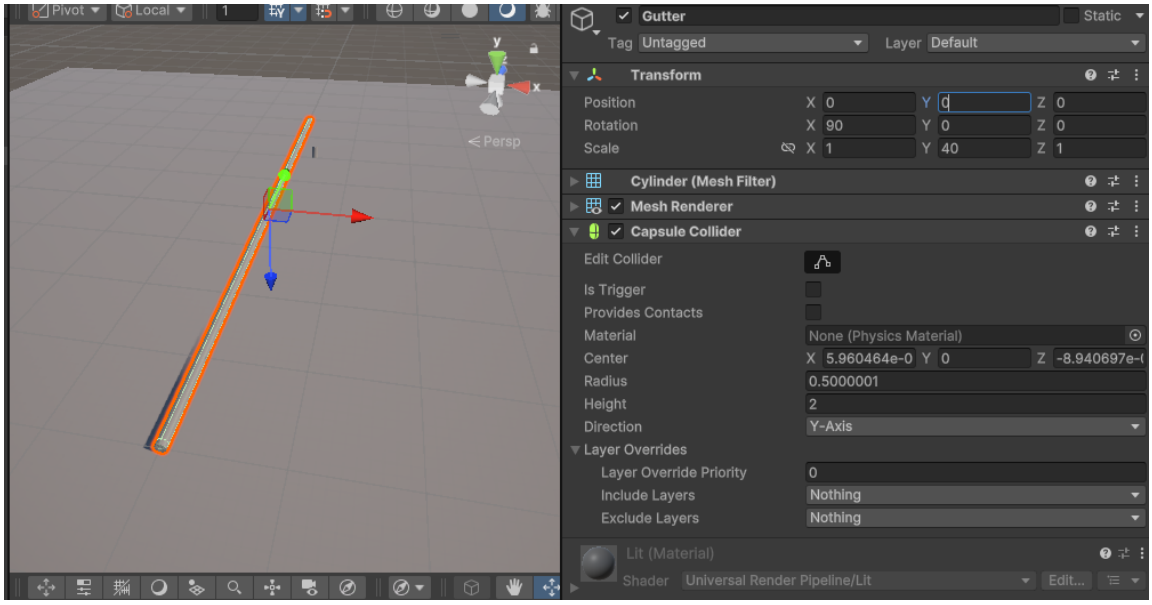
Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Basic bowling and pins setup" would suffice.

Press play to see if our prefab is behaving as expected

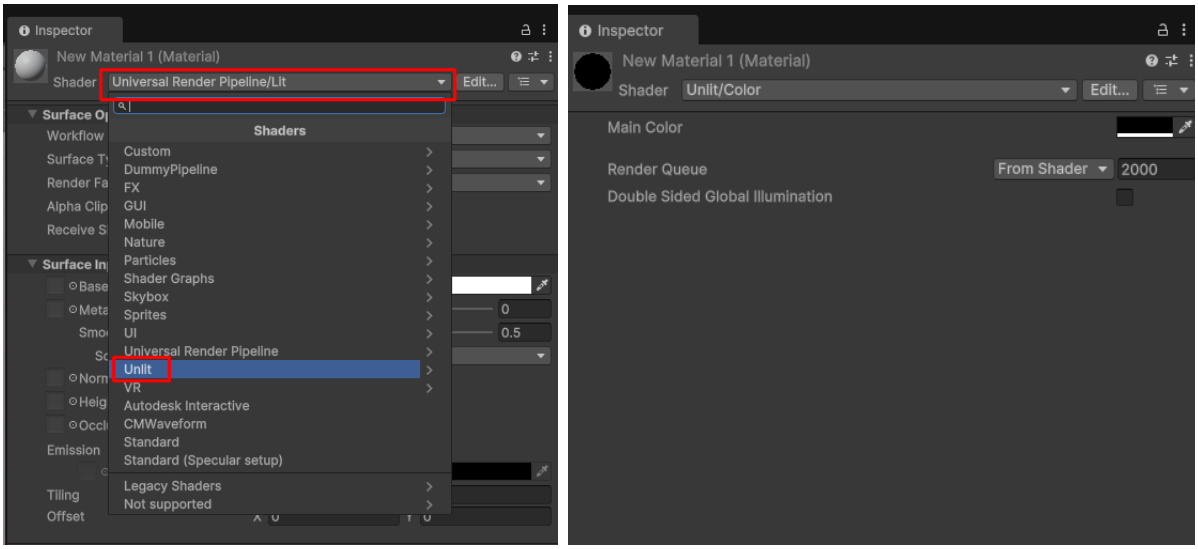
2.3. Trigger Colliders - Gutter

The next step would involve the implementation of a gutter mechanic.

For that functionality we will create another cylinder and rotate it by 90 degrees in the X axis. Additionally, we will set its scale in Y axis to 40 to make it resemble a gutter.



We will also create a material for the gutter so that it looks hollow and not reflective to lighting. Using a lit material will make the gutters look out of place as we have used a cylinder for the gutter. To do so, go `Project Tab > Right Click > Create > Material`. Click on the Drop-down menu in the inspector of the material and then Select `Unlit > Color`. Apply this material to the cylinder by dragging and dropping it from the project window onto the cylinder

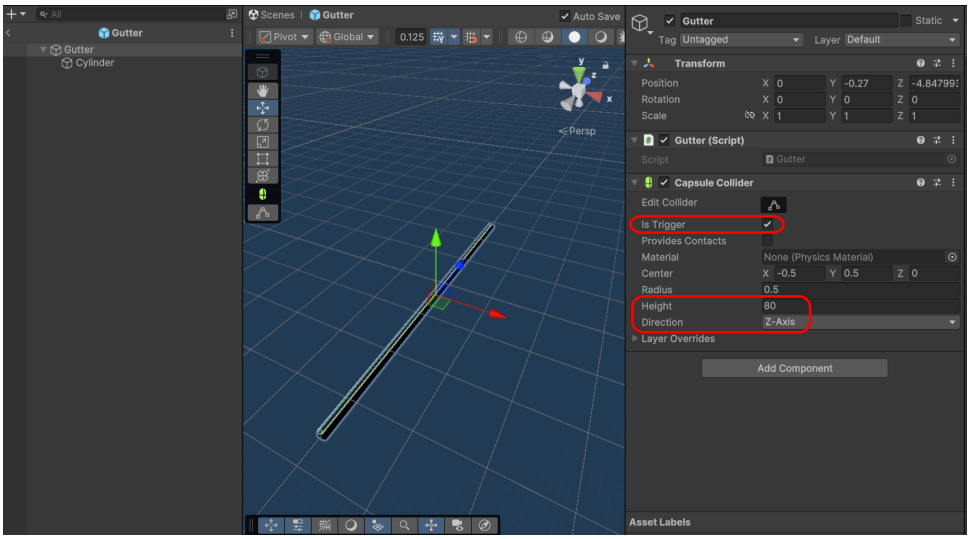


Hint

Since we'll have two gutters on either side of the pins, It would be nice for the gutter to be a prefab at this point

Double click to open the gutter prefab. Right-click it and create a cylinder child. Stretch this cylinder by increasing the y scale so that it's significantly longer (for example, y = 40). Next, on the parent object, add a capsule collider component, and set it to trigger. Set the height of this collider to match the length of the cylinder child you modified (this tends to be about double that scale value, so the image below shows height = 80). Make

sure that the direction of the capsule collider is in Z-axis so that the collider aligns with the forward direction of our transform (i.e. Blue Arrow).

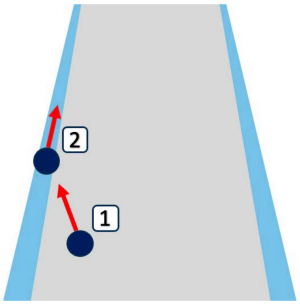


Since we only want to track when the ball enters the gutter without triggering any physical collision response, we need to enable the `"Is Trigger"` property on the gutter's collider. Enabling this property ensures that the collider behaves as a trigger, meaning it no longer interacts with other objects physically but can still detect when other objects pass through it.

To detect when the ball enters the gutter, we will create a script called `Gutter.cs`. To retrieve information about which Rigidbody has entered the trigger, Unity provides the `OnTriggerEnter` method, which is part of any script that derives from MonoBehaviour. When a trigger collider is attached to a GameObject with a MonoBehaviour script, the `OnTriggerEnter` method is automatically called whenever another Rigidbody enters the trigger's bounds. This makes it an efficient way to detect interactions without impacting gameplay physics. The `OnTriggerEnter` receives a Collider object which is a reference to the object that entered the bounds of the trigger

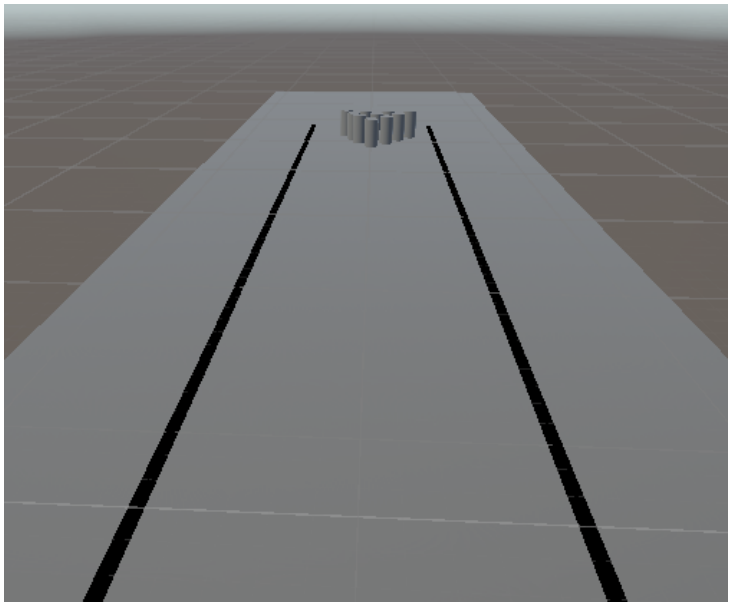
```
using UnityEngine;

public class Gutter : MonoBehaviour
{
    private void OnTriggerEnter(Collider triggeredBody)
    {

    }
}
```

Now, we can use the GetComponent function to get a reference to the RigidBody attached to our ball. Here we can "hijack" the RigidBody's movement, reset it and then propel it in the direction of the gutter.




```
using UnityEngine;

public class Gutter : MonoBehaviour
{
    private void OnTriggerEnter(Collider triggeredBody)
    {
        // we first get the rigidbody of the ball
        // and store it in a local variable ballRigidBody
        Rigidbody ballRigidBody = triggeredBody.GetComponent<Rigidbody>();

        //We store the velocity magnitude before resetting the velocity
        float velocityMagnitude = ballRigidBody.linearVelocity.magnitude;

        //It is important to reset the linear AND angular velocity
        //This is because the ball is rotating on the ground when moving
        ballRigidBody.linearVelocity = Vector3.zero;
        ballRigidBody.angularVelocity = Vector3.zero;

        //Now we add force in the forward direction of the gutter
        //We use the cached velocity magnitude to keep it a little realistic
        ballRB.AddForce(transform.forward * velocityMagnitude,
                        ForceMode.VelocityChange);
    }
}
```

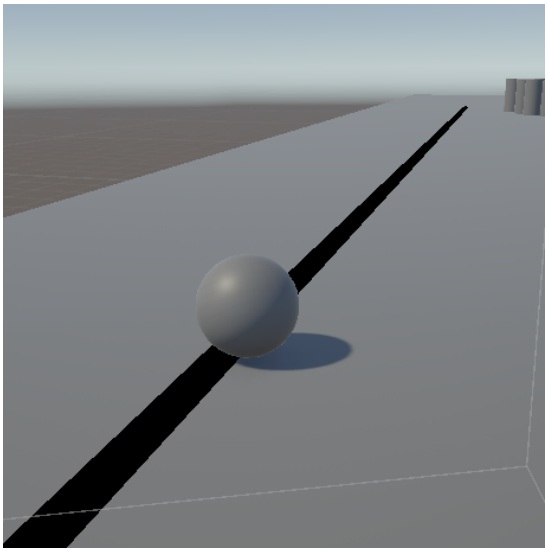
Attach the script to our gutter prefab. Place the gutter prefab on either side of the pins.



To quickly test it out orient your ball's forward transform towards the gutter. Press play and launch the ball

Tip

If your ball is just bouncing over the gutter, make sure you have enabled the is trigger option on the gutter collider!



Tip

Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Gutters implementation" would suffice.

3. Player Character and Camera Controls

We'll now create a basic player character to move left and right, and have a third person camera to aim the ball before firing.

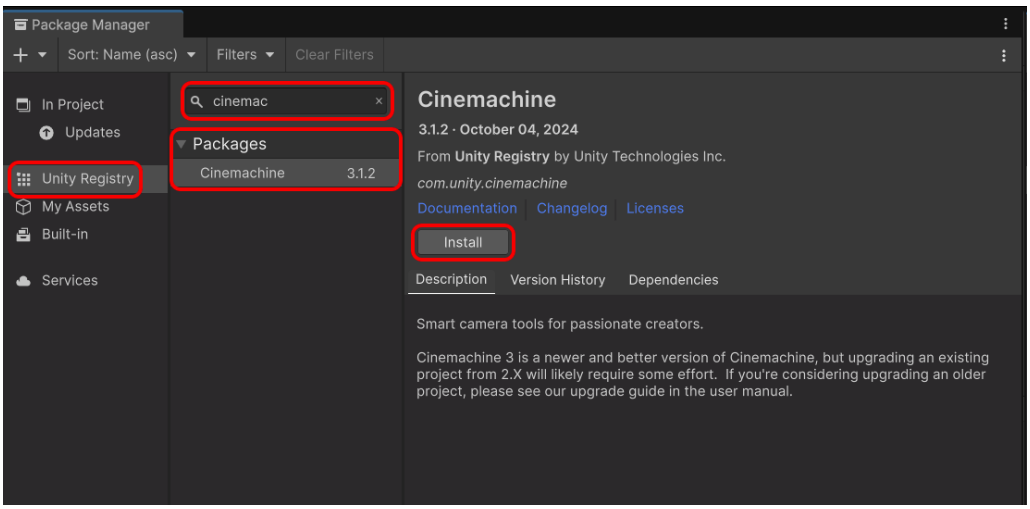
3.1. Cinemachine Camera

Cinemachine in Unity is a powerful tool for creating dynamic and professional camera behaviors without writing custom scripts. It allows you to easily set up cinematic camera transitions, tracking shots, and advanced features like dolly tracks or procedural camera movements. We will be using a Free-Look Cinemachine Camera to look around with our mouse.
Create an empty GameObject and rename it to "Player". Next make a child GameObject called "PlayerModel" with a Cylinder (`3d Object > Cylinder` or Capsule).

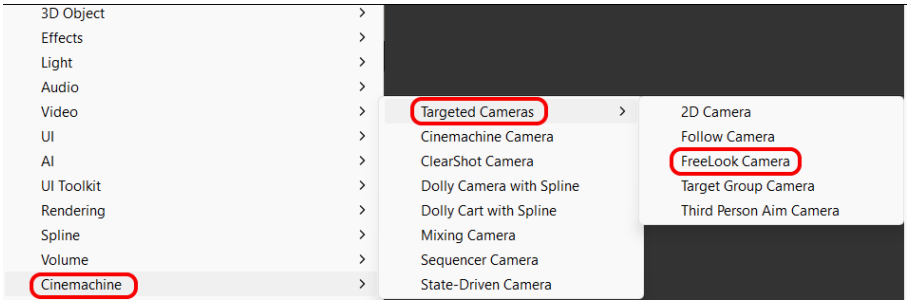
Hint

It's always a good practice to make the model a child of the main object. That way you separate the logic from the visuals. This also lets you polish your game later on without too much of a hassle.

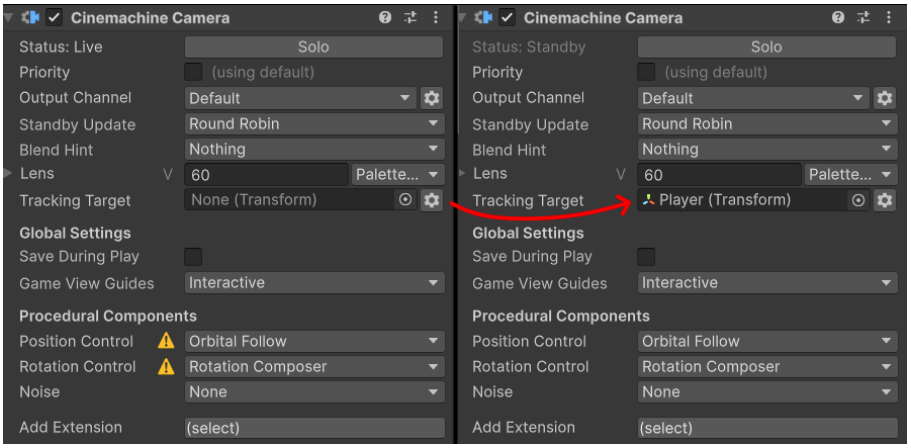
Now we'll import the Cinemachine package into our project. Open the package manager (`Top Menu Bar > Window > Package Manager`) and on the left most tab select the Unity Registry. Then search for "Cinemachine" in the search bar, and install the Cinemachine package into your project.



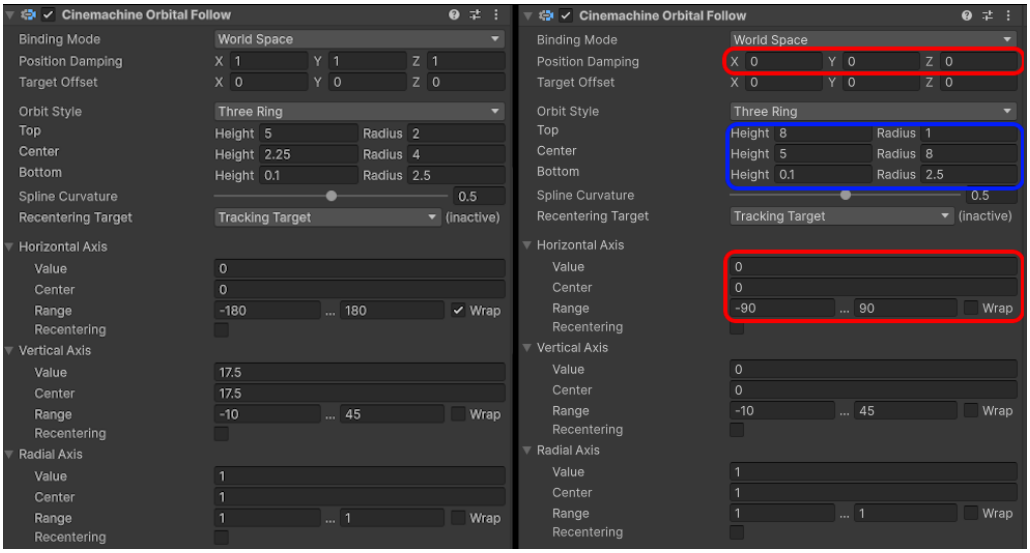
Right click in the hierarchy window, and you should be able to see a new Cinemachine menu option. Create a `Cinemachine > Targeted Cameras > FreeLook Camera`



This will automatically add a "Cinemachine Brain" component to the Main Camera, which controls its properties via the FreeLook Camera that we just created. Now we will have to configure the components on the FreeLook Camera. The first component "Cinemachine Camera" needs is a "Tracking Target". Drag and drop the Player into the field.



This makes the camera follow and look at the target. (If you click on the settings icon there, you have the option to follow a different target and look at a different target).
Next we want to configure the "Cinemachine Orbital Follow" component, which dictates the third-person rotational behavior of the camera around the target. The main two configuration changes here are to remove the dampening to have a fixed camera follower, and to lock the rotation of the camera 90° to the left and right of the player, and switch off warping. The dampening often causes jitter, and requires much more setup and tuning to function correctly.



You can also tune the orbital rings of the player (the range of motion for the top, bottom and middle rings of the third person camera controller) based on your personal preference.

Now if you press play, you will see your mouse controls the camera, which circles around your Player GameObject.

Tip

Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Moved to Cinemachine Camera" would suffice.

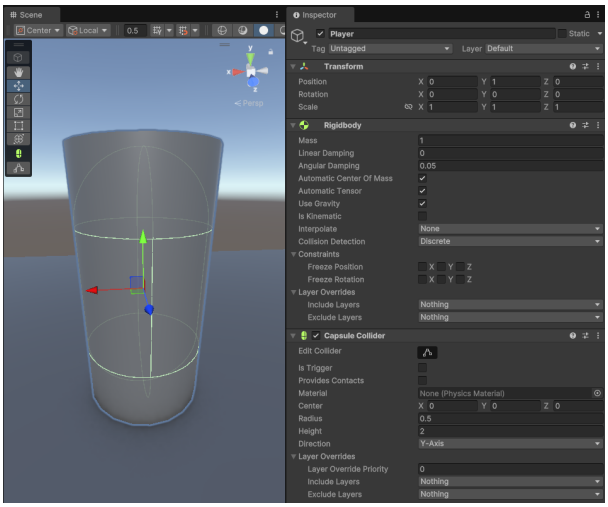
3.2. Player Controls

Now for the game, we would want the player to be able to place, aim and fire the ball. For the player controls, we'll be making a similar Rigidbody based controller as the one from the last assignment. To get started, we can edit the `InputManager` to check and broadcast inputs for A and D keys.

```
using UnityEngine;
using UnityEngine.Events;

public class InputManager : MonoBehaviour
{
    public UnityEvent<Vector2> OnMove = new UnityEvent<Vector2>();
    public UnityEvent OnSpacePressed = new UnityEvent();

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            OnSpacePressed?.Invoke();
        }

```

```
        Vector2 input = Vector2.zero;
        if (Input.GetKey(KeyCode.A))
        {
            input += Vector2.left;
        }
        if (Input.GetKey(KeyCode.D))
        {
            input += Vector2.right;
        }
        OnMove?.Invoke(input);
    }
}
```

Next, create an empty GameObject "Player". Right-click on Player, and create a Cylinder GameObject as a child, and rename that to "PlayerModel" Remove the Capsule Collider component from this model object (right click on the header bar of the component, and select "Remove Component"). Now on the Player (parent) object, add a capsule collider, and a Rigidbody component. The height of the capsule should be 2 to match the height of the cylinder model Finally, raise the whole player object out of the ground.



Now we'll create a Player Controller script.

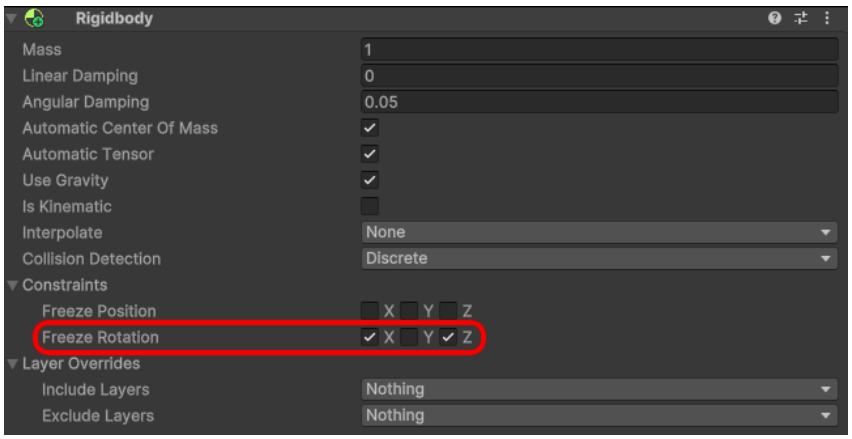
```
using UnityEngine;

public class Player : MonoBehaviour
{
    [SerializeField] private InputManager inputManager;
    [SerializeField] private float speed;

    private Rigidbody rb;

    private void Start()
    {
        //Adding MovePlayer as a listener to the OnMove event
        inputManager.OnMove.AddListener(MovePlayer);
```

```
14          rb = GetComponent<Rigidbody>();
15      }
16
17      //This is similar to our code from our Roll-A-Ball tutorial
18      //Only difference being, we only listen to Left and Right inputs
19      private void MovePlayer(Vector2 direction)
20      {
21          Vector3 moveDirection = new(direction.x, 0f, direction.y);
22          rb.AddForce(speed * moveDirection);
23
24      }
25  }
```

Attach this to the Player GameObject, and assign the InputManager reference (by dragging the object to the field) and set the speed to something decent like 10. However, if we press play right away to try it out (feel free to give it a shot), we'll notice a major problem. The player's body falls over as a result of the force being applied.

> **Note**
>
> It's not just the force that's causing this, but the combination of the force that's acting on the center of the cylinder, and the default friction present on the ground. The two oppose each other, causing it to fall over.

We don't want the player to be able to rotate in the X and Z axis at all. The Rigidbody component allows us to place constraints on the position and rotation of our object. For now, we just want to disable rotations in the X and Z axis.



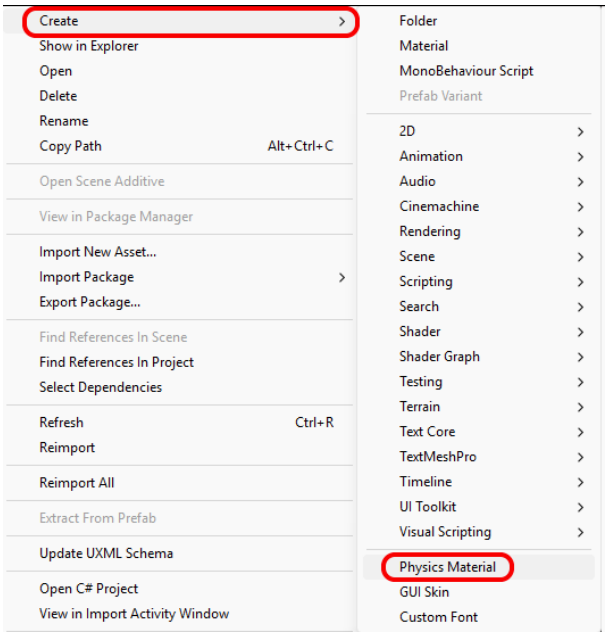
Press play to try it out

Tip

You may notice your player moving back and forth instead of left and right. In this case you have your bowling alley set up in the wrong axis (it might be stretching along z instead of x) You can rotate the environment to fix this, or alternatively swap the x and y components from the input when defining move direction. Your player could also be moving inverted, in which case a simple hack like adding a minus sign to the force value should suffice. This is of course not ideal, and should be rectified down the line by reorienting the environment.

The player object should now be moving without falling over. However it feels a bit slippery, as if moving on ice. Modifying a global friction property would not be the solution, since we do want the ball to feel like it can slide smoothly. Hence we'll be setting up some physics materials to control the strength of friction on various objects. Physics Materials mainly control two properties of physics objects: their friction and their bounciness (coefficient of restitution).

Right click in the Project window, and create two Physics Materials. Name them "Smooth" and "Rough".

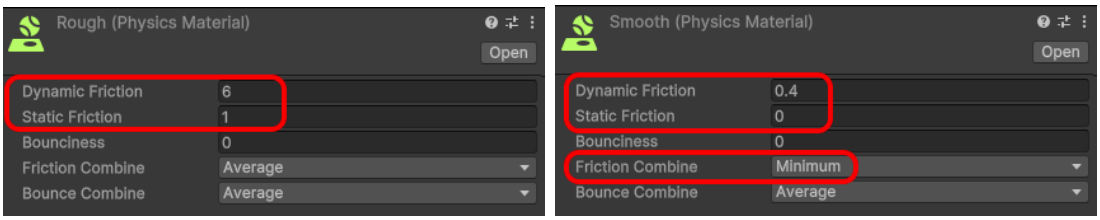


Now if you select (either) Physics Material, you'll see a few parameters you can edit.

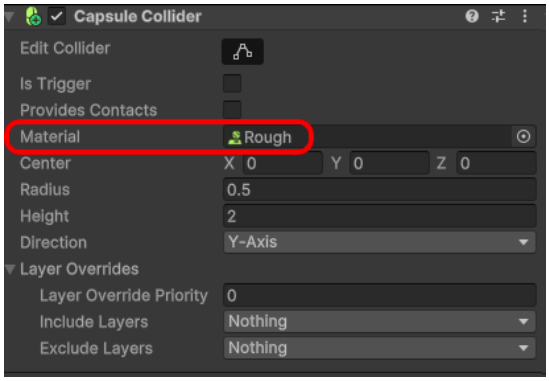
- The Dynamic friction dictates how much friction there is between two moving objects. This is what controls the strength of "slowing down"
- The Static friction controls how much friction there is between two still objects. This is what controls the amount of force required to start moving an object.
- The Bounciness controls the amount of velocity the object should preserve on a collision. 0 would mean it would come to a standstill, while 1 would cause a full bounce.

- Friction Combine dictates the friction combination logic to be followed when interacting with another physics object, i.e. should the two frictions be averaged, be multiplied, only the minimum of the two should be applied or the maximum of the two.
- Finally, Bounce Combine does the same but for collisions and bouncing.

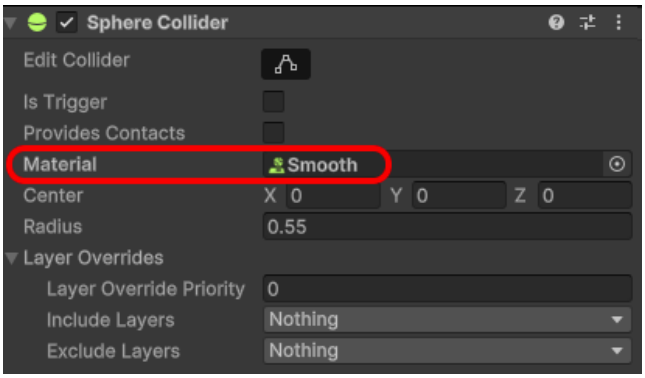
We would ideally like a responsive player character. This means being able to get up to speed quickly, and then slow down quickly as well. Based on the above, description, that would mean a low static friction value of 1 and a high dynamic friction value of 6 should work nicely. Our smooth material which would be on the ball should have a low dynamic friction and 0 static friction, since it should pick up speed instantly, and then maintain that momentum. Setting dynamic friction to 0.4 and static to 0 will suffice. If the friction combine mode is average on the ball, it would mean that it would average it with the ground, which is 0.6 by default. But we want the friction to be lower than that. We could either do the math and set it to 0.2, or set the friction combine mode to minimum to automatically pick 0.4 as the minimum value.



Finally, attach these materials in the physics material by dragging and dropping the rough material to the Player GameObject's Capsule Collider and the smooth material for the Ball GameObject's Sphere Collider.



(a) Player Collider



(b) Ball Collider

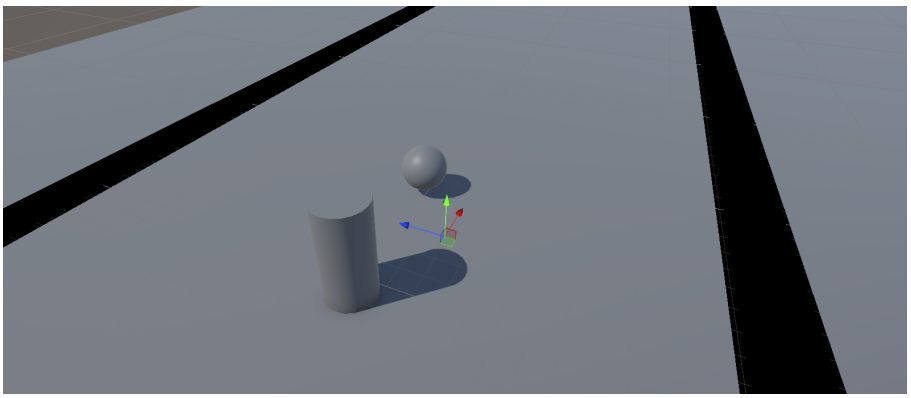
Give it a try now.

Tip

Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Player movement" would suffice.

3.3. Moving Ball with Player

We do have a moving player, but the ball doesn't move with the player. A simple way to rectify this would be to make the ball a child of the player, and once the ball needs to be fired, unparent the ball and fire it. That way the ball isn't affected by the player's movement post firing. Let's first create an anchor which the ball will be positioned to as a child of the Player. Right click on the Player GameObject, and create a new empty GameObject called "BallAnchor". Position this to the forward and right of the player in the scene view using the move transform gizmos, and make sure it's got a little bit of height off the ground so the ball doesn't get placed underneath the ground.



We can modify the ball controller as follows for this parenting and unparenting behavior.

```
using UnityEngine;
using UnityEngine.Events;

public class BallController : MonoBehaviour
{
    [SerializeField] private float force = 1f;
    [SerializeField] private Transform ballAnchor;

    private bool isBallLaunched;
    private Rigidbody ballRB;
    private InputManager inputManager;

    void Start()
    {
        ballRB = GetComponent<Rigidbody>();
        inputManager.OnSpacePressed.AddListener(LaunchBall);
        transform.parent = ballAnchor;
        transform.localPosition = Vector3.zero;
    }


    private void LaunchBall()
    {
        if (isBallLaunched) return;
        isBallLaunched = true;
```

```
        transform.parent = null;
        // this sets the object to the outermost layer of the hierarchy
        ballRB.AddForce(transform.forward * force, ForceMode.Impulse);
    }
}
```

Drag and drop the reference for the "BallAnchor" to the `BallController` component. However, if you press play and move around, you'll see that the ball is stationary on the ground. There's one final change we have to make.
Rigidbodies preserve the physics of the object associated with it. However, we can toggle a boolean called `isKinematic`, which basically says "control this object only via code and not with physics". However, a Rigidbody will always detect collisions as long as the GameObject's collider is active, regardless of whether the Rigidbody is set to kinematic or not. We can set isKinematic to be true before launching the ball on `Start` and then set it to false after launching the ball and before applying the force

```
using UnityEngine;
using UnityEngine.Events;

public class BallController : MonoBehaviour
{
    [SerializeField] private float force = 1f;
    [SerializeField] private Transform ballAnchor;

    private bool isBallLaunched;
    private Rigidbody ballRB;
    private InputManager inputManager;

    void Start()
    {
        ballRB = GetComponent<Rigidbody>();
        inputManager.OnSpacePressed.AddListener(LaunchBall);
        transform.parent = ballAnchor;
        transform.localPosition = Vector3.zero;
        ballRB.isKinematic = true;
    }


    private void LaunchBall()
    {
        if (isBallLaunched) return;
        isBallLaunched = true;
        transform.parent = null;
        ballRB.isKinematic = false;
        ballRB.AddForce(transform.forward * force, ForceMode.Impulse);
    }
}
```

Now if you press play and move around, the ball should move with the player, and pressing the space key should launch the ball forward. Keep an eye on the inspector, and you

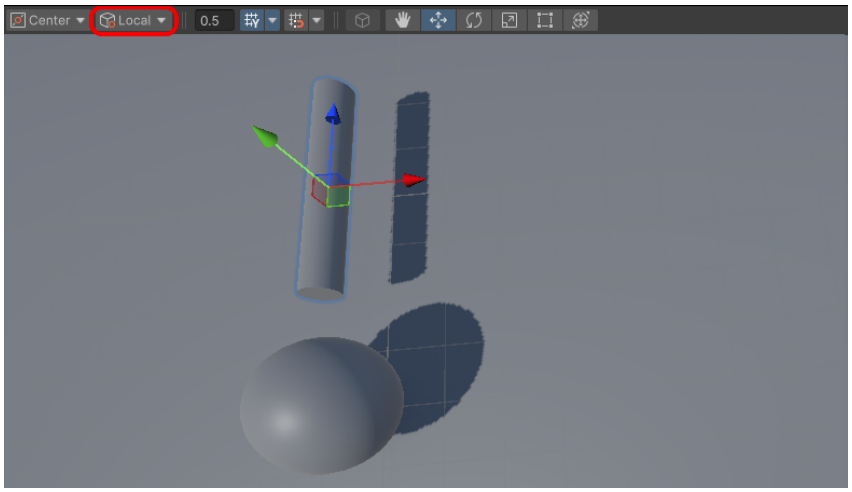
should see the Ball initially be a child of the anchor object we created, and after firing the ball it should set itself to the outermost layer of the hierarchy.

> **Tip**
>
> Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Ball moves with player" would suffice.

3.4. Aiming Controls

Next we want to have the ability to aim the ball with our camera. Currently the ball fires in the direction of its transform forward. We would instead like to aim with our camera, and have an indicator to show which way our ball is going to fire.
Create an empty child under the Ball GameObject, and call it "LaunchIndicator" Right click that and make a child cylinder called "IndicatorModel". Orient it such that the forward of the "LaunchIndicator" is along the length of the cylinder. Place it in front of the ball.

> **Tip**
>
> Make sure the transform gizmo is showing the local orientation to understand which way the transform.forward is for the IndicatorModel



Next, create a new script `LaunchIndicator.cs`

```
1  using Unity.Cinemachine;
2  using UnityEngine;
3
4  public class ArrowIndicator : MonoBehaviour
5  {
6      [SerializeField] private CinemachineCamera freeLookCamera;
7
8      void Update()
9      {
```

```
        transform.forward = freeLookCamera.transform.forward;
        transform.rotation = Quaternion.Euler(0,transform.rotation.eulerAngles.y,0);
    }
}
```

Hint

There are two common notations for representing rotations:

1. **Euler Angles** – Expressed using three values (X, Y, and Z). This notation is intuitive for humans (e.g., an object rotated by 10° around the X-axis and 5° around the Y-axis).
2. **Quaternions** – Represented using four values (X, Y, Z, and W). Unlike Euler angles, quaternions do not suffer from gimbal lock, making them the preferred choice in computer graphics for stable and smooth rotations.

In Unity, rotations displayed in the Inspector use Euler angles. However, Unity internally converts them into quaternions for computation. To perform this conversion, Unity provides the `Quaternion.Euler()` function. If you are interested in Quaternions vs Euler notations, here's a fun video from 3Blue1Brown for more info!

We can extract the y axis rotation value by getting the Euler angles property of the rotation, and set x and z values to 0. This way we get the orientation of the camera, but we only preserve the y axis rotation. Attach this as a component to the LaunchIndicator object, and drag in the reference to the Cinemachine GameObject. **Press Play to try it out**

We can now use this launch indicator to determine the vector to fire our ball in. Make the following updates to the `BallController` script:

```
using UnityEngine;
using UnityEngine.Events;

public class BallController : MonoBehaviour
{
    [SerializeField] private float force = 1f;
    [SerializeField] private Transform ballAnchor;
    [SerializeField] private Transform launchIndicator;

    .
    .
    .

    private void LaunchBall()
    {
        if (isBallLaunched) return;
        isBallLaunched = true;
        transform.parent = null;
        ballRB.AddForce(launchIndicator.forward * force, ForceMode.Impulse);
```

```
20          launchIndicator.gameObject.SetActive(false);
21      }
22  }
```

We can disable the indicator once the ball has been launched. Make sure to add the reference to the `launchIndicator` transform.
Press Play to try it out

> **Tip**
>
> Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Launch Indicator" would suffice.

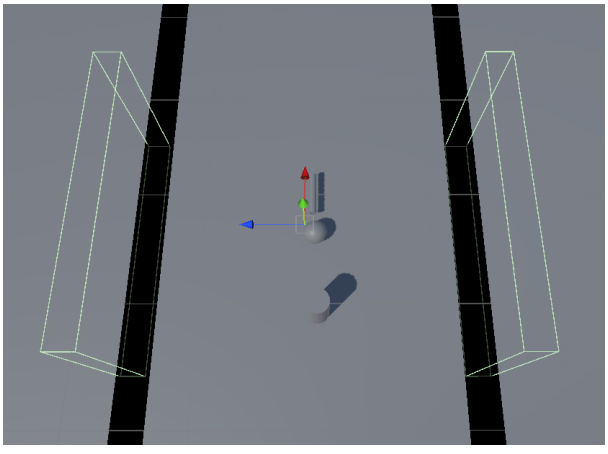
We can now move the player, aim the ball and fire! There's just one main issue remaining at this point: The player can move out of the bowling alley past either gutter. This is a great time to implement some basic invisible walls.

3.5. Invisible Walls

There are a few ways to restrict the movement of the player. We can implement the simplest of them all for our purposes, which are invisible walls.

An invisible wall is just a box collider without a mesh. Create two cubes, and adjust them to act as walls along the gutters so that the player can't cross over. Then select both objects, and remove the Mesh Renderer and Mesh Filter components.

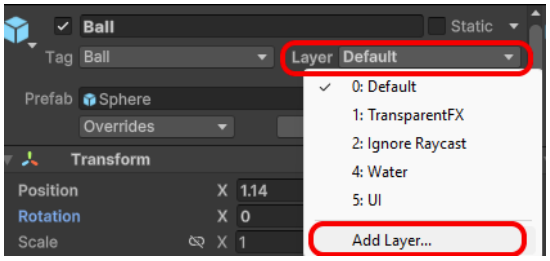
> **Note**
>
> Since we need collision functionality make sure the `IsTrigger` property of the collider is **Set to False**



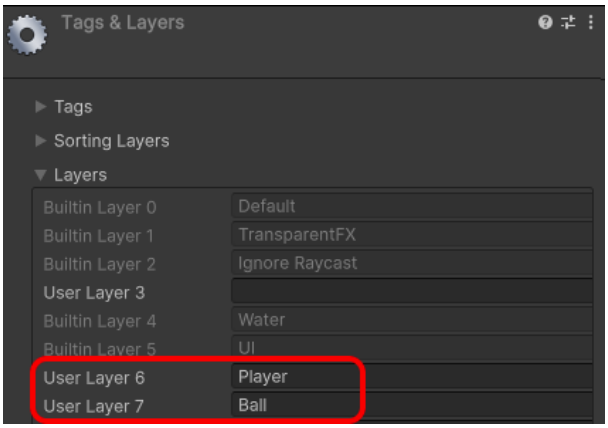
Press Play and try it out.
This works fine, but now there's a new strange edge case. If the player decides to turn to the absolute right or left, they would fire the ball into the invisible wall, which would prevent it from connecting with the gutter. We can set up physics layer to avoid the ball colliding with the walls meant for the player.

3.5.1 Physics Layers

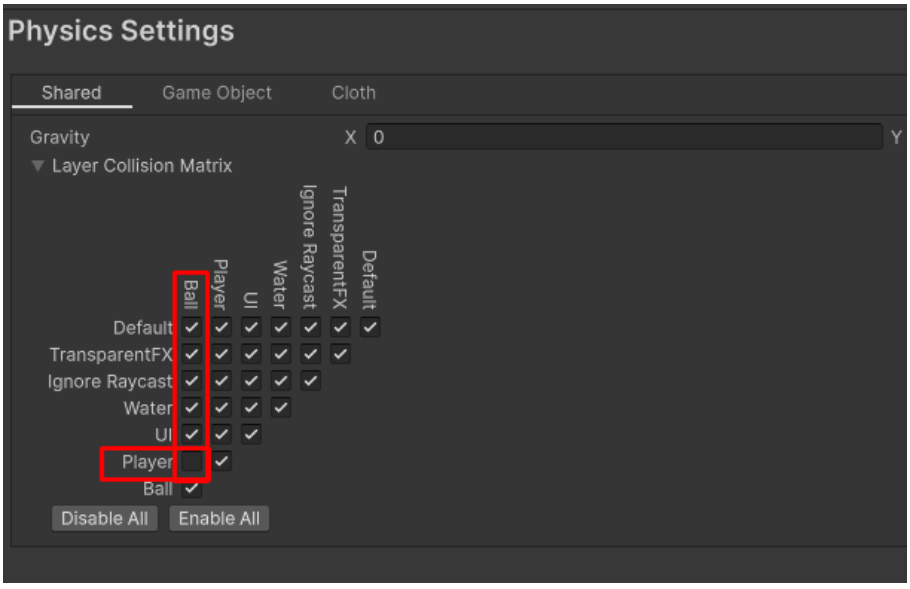
Physics layers control which physics objects interact with each other. By placing objects in specific layers, you can determine which First select the Ball gameobject, open the "Layer" dropdown, and select "Add Layer..."



Next, we can add two layers, one for the Player and one for the Ball.



Now, from the top bar select `Edit > Project Settings > Physics > Settings`. Under the shared tab, you will be able to see the collision matrix. We don't want objects in the ball layer to interact with the player layer, so we'll uncheck that combination.



Finally, put the Player object (and all its children) in the player layer, the invisible walls

in the player layer, and the ball (and all its children) in the ball layer. This way, objects in ball layer and player layer cannot interact with each other
Now if you press play and launch the ball directly to the left or right of the player, it won't interact with the invisible walls at all!

Tip

Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Basic Gameplay Complete" would suffice.

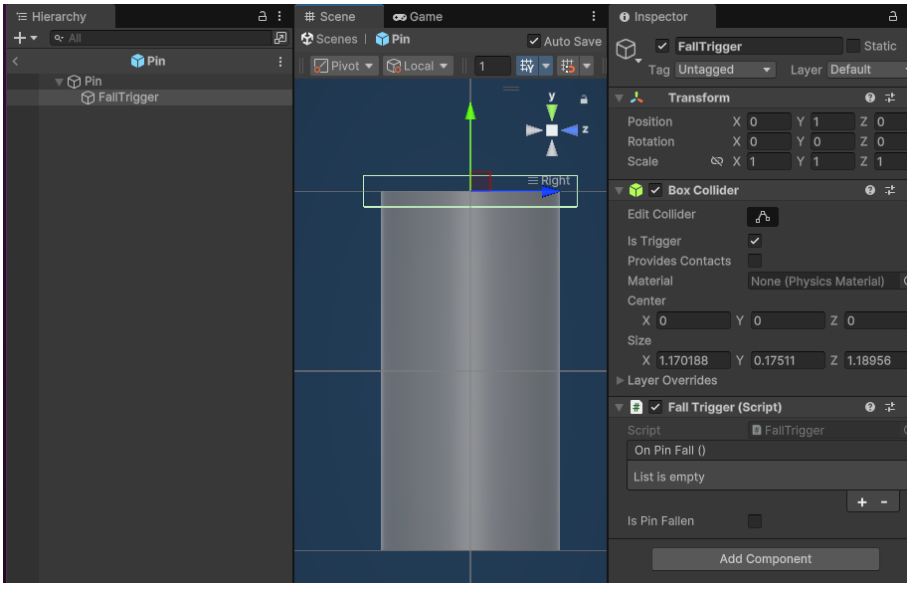
Now our game is just about playable, but we would like to have a few more features, namely a score keeping feature, and the ability to Reset the game without having to exit play mode, and keep a persistent score.

4. Game Management

A game often requires a manager script which controls data and state between different objects. It acts as an orchestrator and ensures that different unrelated GameObject can communicate with each other.

4.1. Detect Pin Fall

To detect a pin fall, we will create a child object named "FallTrigger" on the pin prefab. Attach a box collider to the FallTrigger and position it near the top of the pin. Ensure that the `IsTrigger` property of the collider is set to `true`.

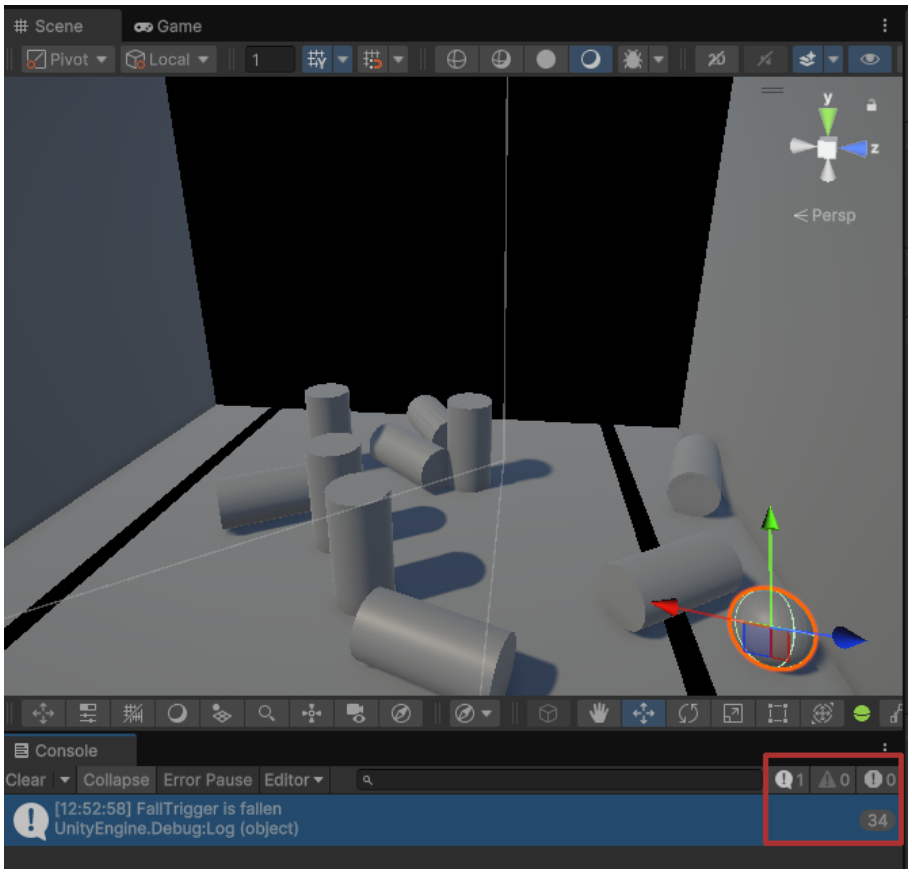


Next, we will create a script called `FallTrigger.cs` to handle the `OnTriggerEnter` event for the collider.

```
1 using System;
2 using UnityEngine;
```

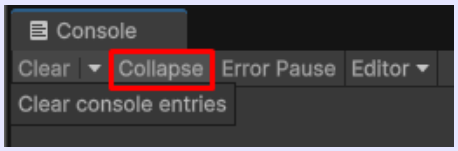
```
using UnityEngine.Events;

public class FallTrigger : MonoBehaviour
{
    //Adding an OnPinFall public event in case any other object
    //needs to know whether a Pin has Fallen i.e. Our GameManager
    public UnityEvent OnPinFall = new();
    public bool isPinFallen = false;
    private void OnTriggerEnter(Collider triggeredObject)
    {
        isPinFallen = true;
        OnPinFall?.Invoke();
        Debug.Log($"{gameObject.name} is fallen");
        // using $"" is C#'s string formatting
        // similar to Python's f-string
        // or Java's String.format()
    }
}

```

However, there is a flaw in this implementation. The FallTrigger object detects the pin as fallen as soon as it interacts with any collider, regardless of the collider's type. For instance, even though only six pins should have been detected as fallen, the debug log statement (and consequently, the `OnPinFall` event) was triggered 34 times, indicating multiple unintended detections.



Hint

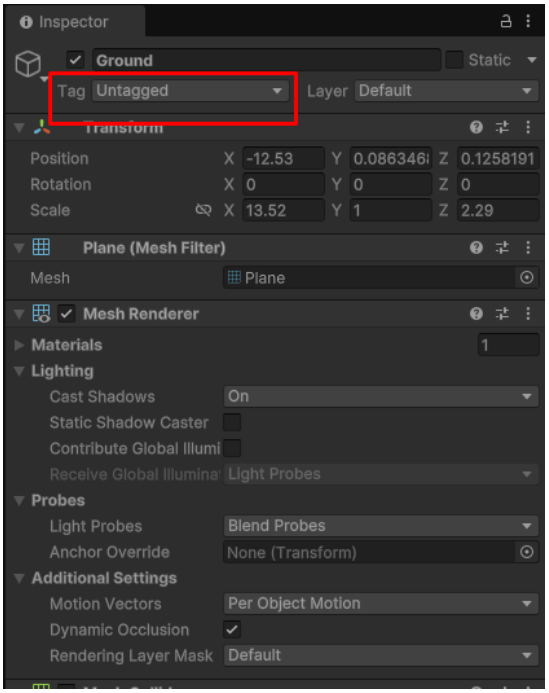
If your console is not displaying the messages as shown in the screenshot above, check the "Collapse" button in the Console window. Clicking this button toggles the grouping of identical log messages.



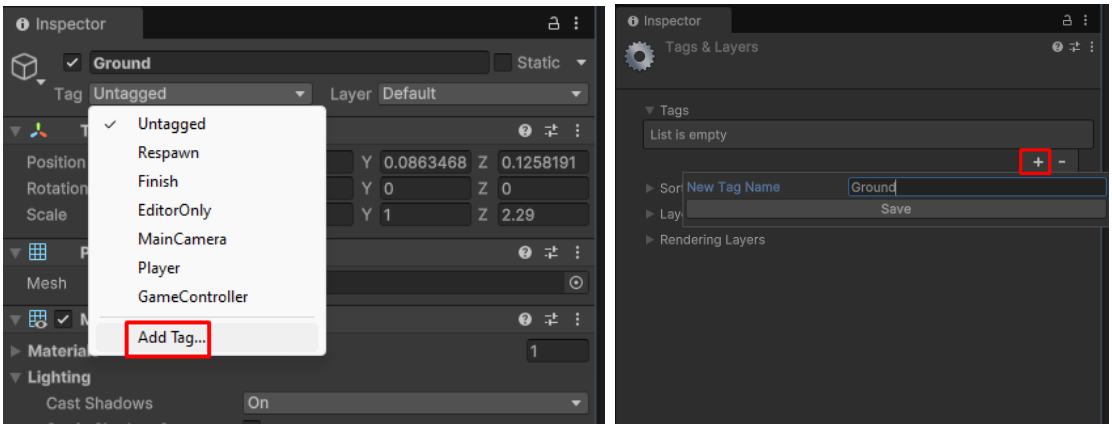
4.1.1 Tags

To resolve this issue, we must ensure the FallTrigger interacts only with the ground to register a fall. Unity allows objects to be assigned "Tags," which can be used to identify specific objects during collisions. We can compare these tags to verify if the interaction is with the intended object.

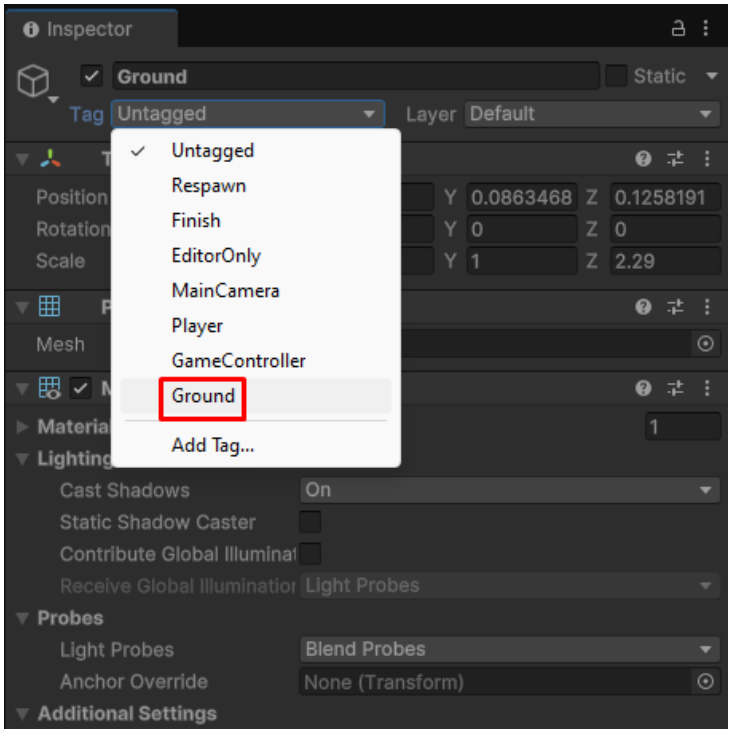
Select your Ground GameObject (i.e. the Ground Plane). Towards the top of the Inspector window, there is a small field called "Tag".



Unity comes with a set of default tags like "Respawn", "Finish", "EditorOnly", "Player" and so on. We can use any of the default tags and it'll serve our purpose. However, it is best practice to create a new tag if its going to serve a specific purpose (In our case, to detect if our Fall Trigger has hit the ground)



Now that we have added a ground tag to our project, we need to go back to Ground GameObject and assign it the "Ground" Tag.



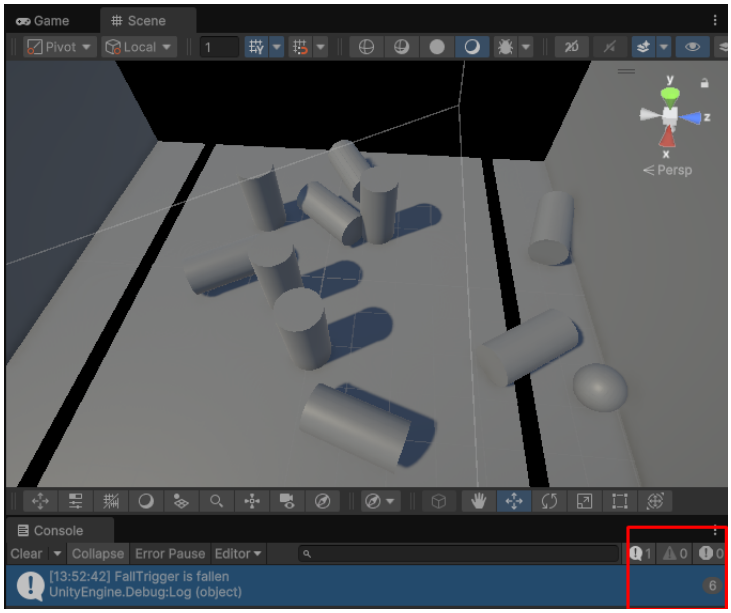

```
public class FallTrigger : MonoBehaviour
{
    public UnityEvent OnPinFall = new();
    public bool isPinFallen = false;
    private void OnTriggerEnter(Collider triggeredObject)
    {
    // We can use the CompareTag() function
    // to compare the collided object's tag to a string
    // We also make sure that the OnPinFall is not invoked more than once
    // For example if a pin were to bounce off the ground and hit it twice
        if (triggeredObject.CompareTag("Ground") && !isPinFallen)
        {
            isPinFallen = true;
```

```
            OnPinFall?.Invoke();
            Debug.Log($"{gameObject.name} is fallen");
        }
    }
}
```

Hint

Looking back at the script we created in Section 2.3, you may notice that it lacks validation to ensure only the ball is launched in the gutter's direction. As a result, any rigidbody entering the gutter—including unintended objects like pins—would also be launched, potentially causing unexpected behavior or bugs. To address this, it is a good time to create a tag called "Ball" and implement a safety check, similar to the one used in the `FallTrigger` script, to ensure only the intended object is launched by the gutter.

This change should ideally trigger the fall condition the intended amount of times (in this case 6 times).



Press play and try it out

4.2. Score

To maintain the state of our game, we need a `GameManager.cs` that not only manages the state of our game but also acts as a intermediary to communicate between different GameObjects (e.g., if we want to detect pins and increment the score of the player, change levels etc).

```
using UnityEngine;

public class GameManager : MonoBehaviour
{
```

```
    [SerializeField] private float score = 0;
}

```

Attach this script to our "GameManager" GameObject that we have created previously. Since we can accurately detect when each pin falls, we can increment the score by listening to the `OnPinFall` event on all the pins. To do so, we will use the prebuilt `FindObjectsByType` function, which takes a parameter **FindObjectsInactive.** This determines whether to include inactive objects in the search. Setting this to:

- **FindObjectsInactive.Include** ensures that both active and inactive pins are included in the search
- **FindObjectsInactive.Exclude** limits the search to active objects only.

The method also has a placeholder `<T>`. The method returns an array of type `T[ ]`, where T is the type of objects being searched for (e.g., GameObject, a custom script, component, etc.). We can define what type `<T>` can be at runtime (in our case `FallTrigger`).

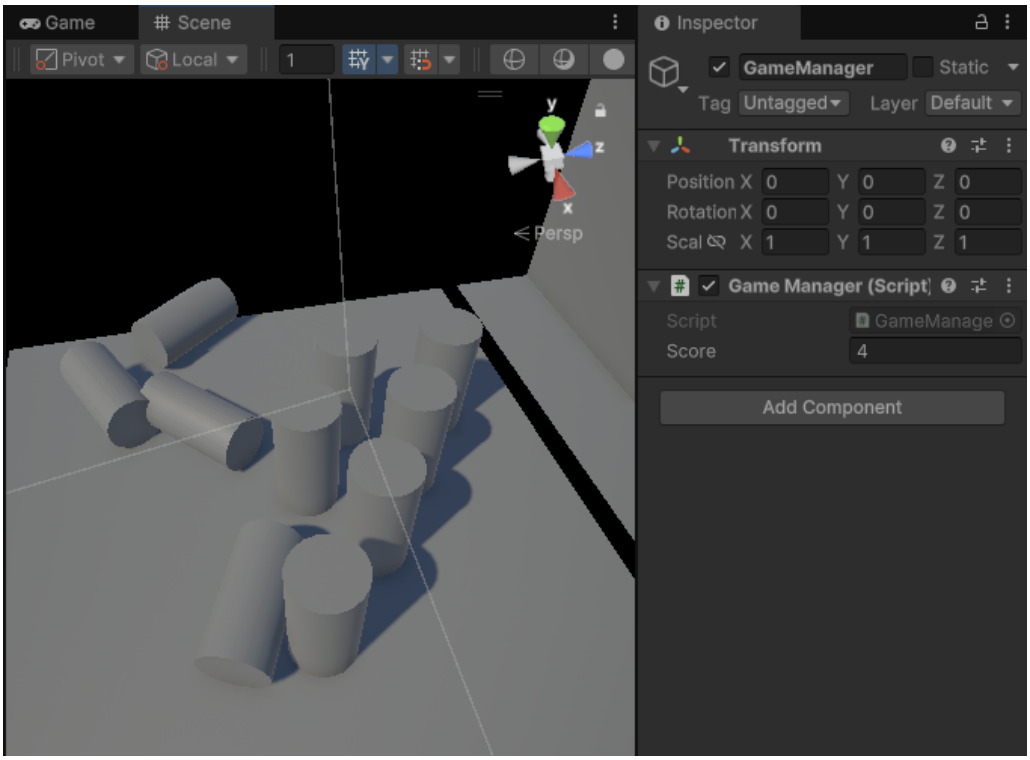
> **Note**
>
> Instead of writing duplicate code for each type, generics allow the use of placeholders (like <T> in C#) that are replaced with specific types at compile-time. Generics are commonly used across many programming languages, and in Unity, they play a crucial role in simplifying code. For example, the GetComponent<> function in Unity is a generic method that ensures type safety, allowing you to retrieve components of any type without the need for multiple function overloads.

Here's how we can use this function to find all pins in the scene and subscribe to the `OnPinFall` event:

```
using UnityEngine;

public class GameManager : MonoBehaviour
{
    [SerializeField] private float score = 0;
    private FallTrigger[] pins;

    private void Start()
    {
    //We find all objects of type FallTrigger
        pins = FindObjectsByType<FallTrigger>(FindObjectsInactive.Include);

        //We then loop over our array of pins and add the
        // IncrementScore function as their listener
        foreach (FallTrigger pin in pins)
        {
            pin.OnPinFall.AddListener(IncrementScore);
        }
    }
```

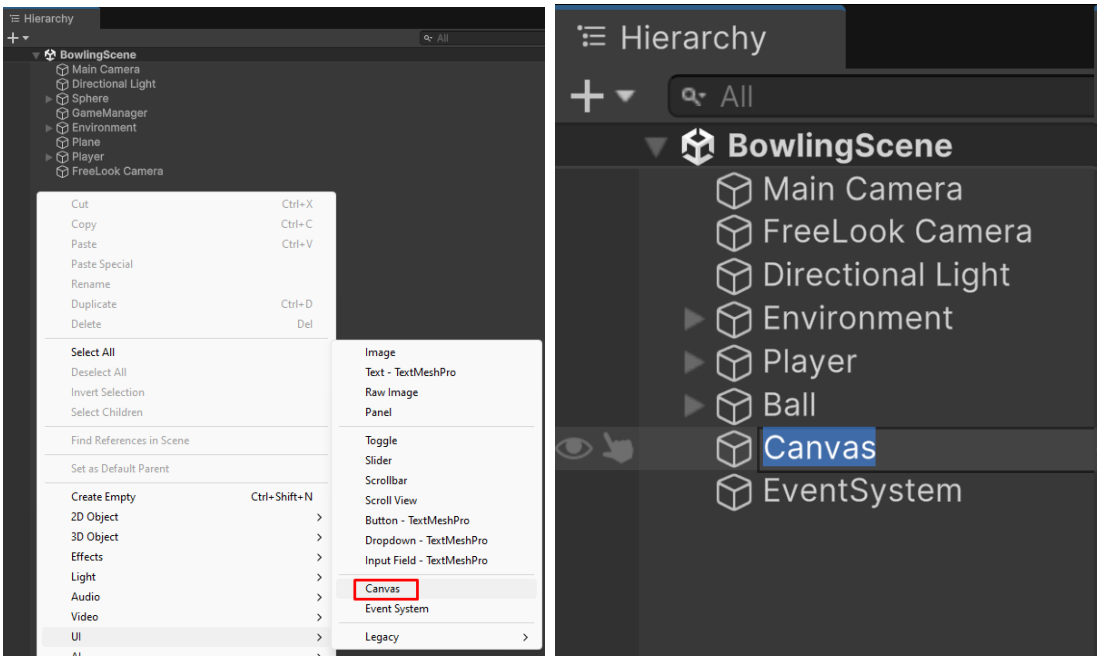
```
20
21      private void IncrementScore()
22      {
23          score++;
24      }
25  }
```

Press Play and test it out
Ideally as the pins drop, the score should increment in the inspector of the GameManager GameObject.



4.2.1 Unity UI

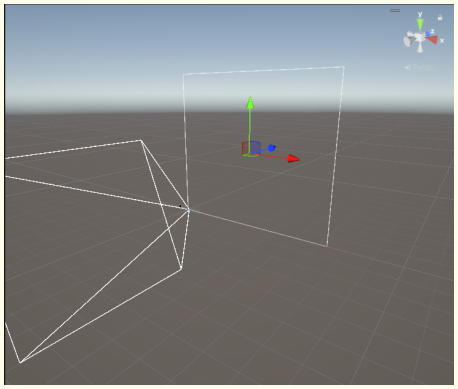
We need a way to display the score on the UI to show the player the score. For this we will use a "Canvas", which is a core unity component to create user interfaces in Unity. To create a canvas, `Right-Click in Hierarchy Window > UI > Canvas`.



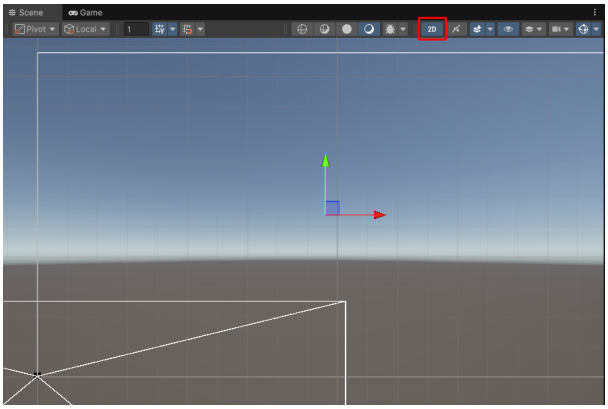
After creating a canvas, you will notice that a GameObject named "Event System" is created automatically by Unity. This is essential for the Canvas to work, so DO NOT delete it.

Note

When using the Canvas in World Space mode, you may notice that it appears significantly larger than your GameObjects and cannot be moved like other GameObjects. This is because the Canvas essentially functions as its own World Space. Unity handles the rendering of the Canvas separately from the rest of the scene's World Space and then overlays it on top of the Main Camera's view.

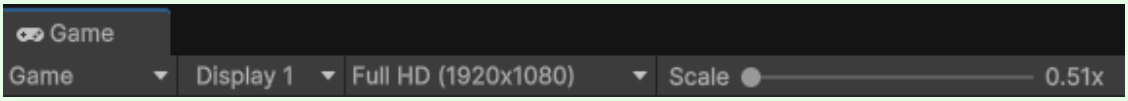


You can use the 2D mode in the Scene View to arrange your UI Elements in the canvas.

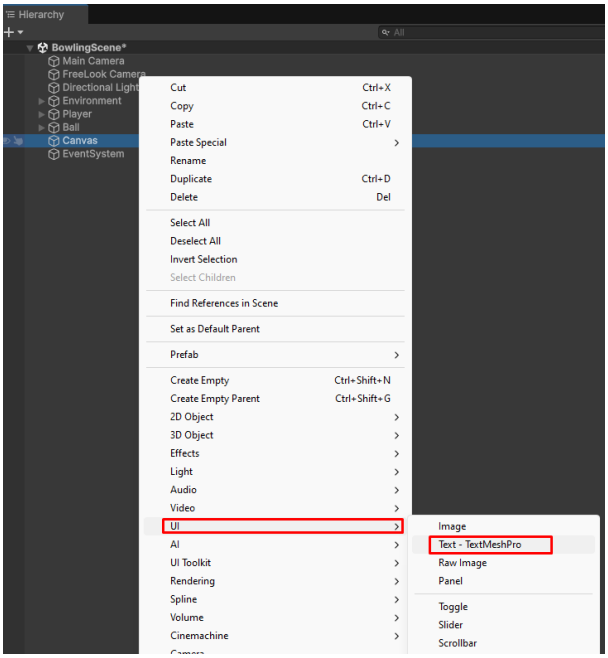


Tip

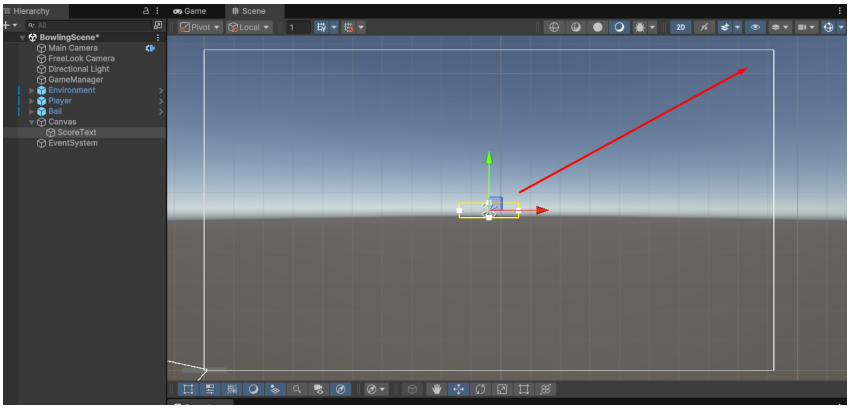
If you don't see your canvas properly, make sure in your game view tab that your aspect ratio is set to 1920x1080, and your scale slider is at the leftmost limit.



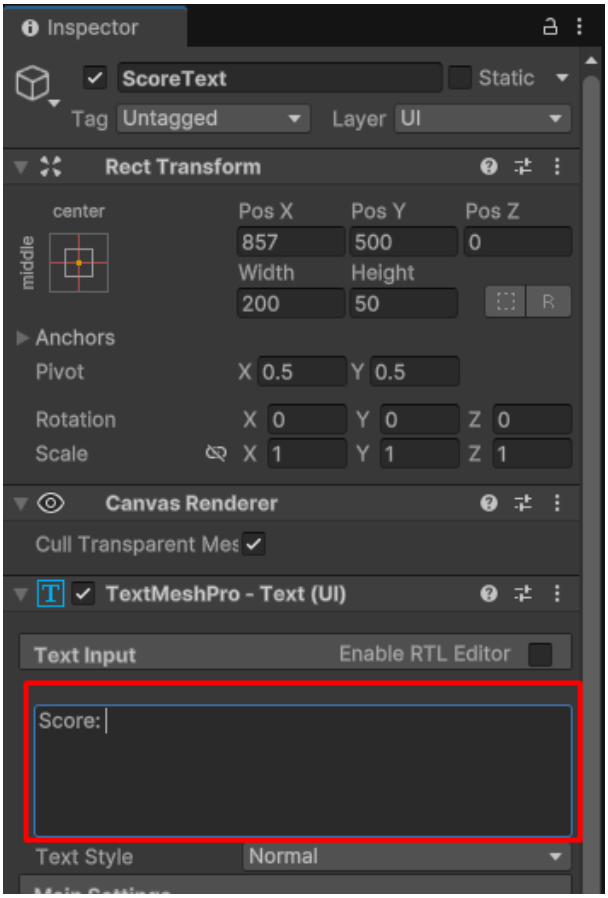
To display text to the user, we will utilize TextMeshPro, a built-in Unity component designed for rendering text. To add it, simply `Right Click on Canvas GameObject > UI > TextMeshPro`.



Now, we'll move the TextMeshPro to the top-right of our screen similar to how we move other GameObjects.



And replace the text with "Score:"

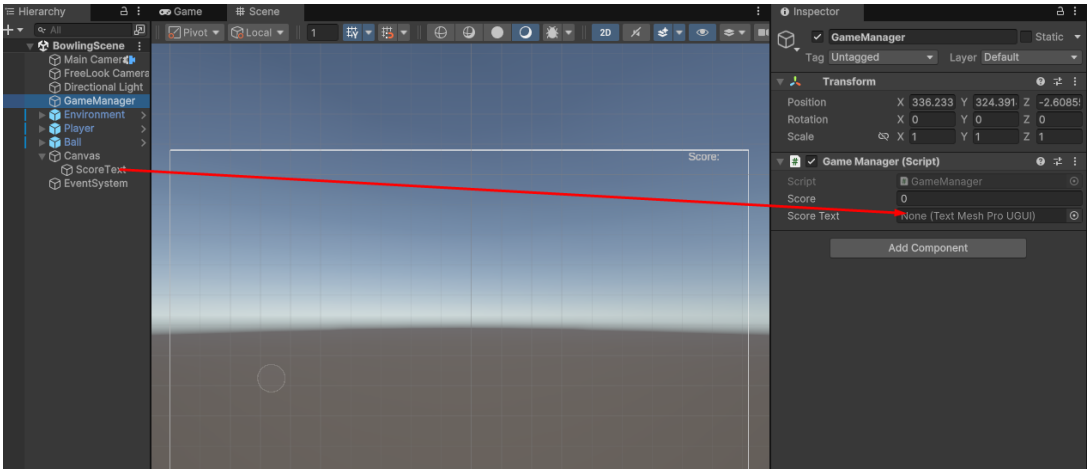


Now, we'll modify our GameManager script to have a reference of this newly created TextMeshPro and modify its text every time the score is updated.

```
using TMPro;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    [SerializeField] private float score = 0;
    [SerializeField] private TextMeshProUGUI scoreText;
    private FallTrigger[] pins;
```

```
9
10      private void Start()
11      {
12          pins = FindObjectsByType<FallTrigger>(FindObjectsInactive.Include);
13          foreach (FallTrigger pin in pins)
14          {
15              pin.OnPinFall.AddListener(IncrementScore);
16          }
17      }
18
19      private void IncrementScore()
20      {
21          score++;
22          scoreText.text = $"Score: {score}";
23      }
24  }
```

Now simply drag and drop your TextMeshPro object into GameManager's inspector.



Press play button and test it out
It should automatically update the score everytime a pin falls

> **Tip**
>
> Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Added scoring system" would suffice.

4.3. Reset

Now, we'll add Reset functionality when the player presses the "R" key. So, We'll quickly add a event upon pressing the R key to the `InputManager`.

```
1  public class InputManager : MonoBehaviour
2  {
3      public UnityEvent<Vector2> OnMove = new UnityEvent<Vector2>();
```

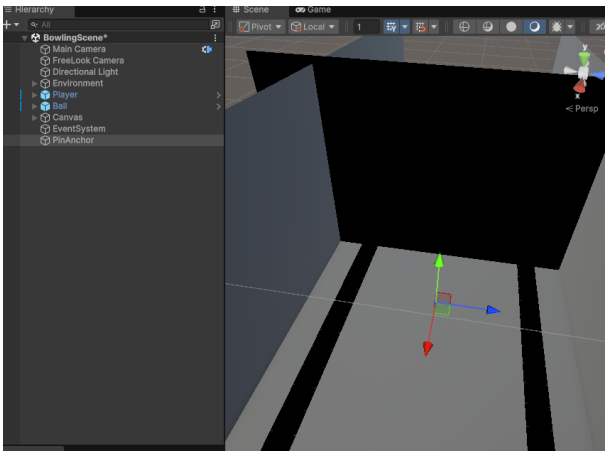
```
    public UnityEvent OnSpacePressed = new UnityEvent();
    public UnityEvent OnResetPressed = new UnityEvent();

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            OnSpacePressed?.Invoke();
        }

        Vector2 input = Vector2.zero;
        if (Input.GetKey(KeyCode.A))
        {
            input += Vector2.left;
        }
        if (Input.GetKey(KeyCode.D))
        {
            input += Vector2.right;
        }
        OnMove?.Invoke(input);

        if (Input.GetKeyDown(KeyCode.R))
        {
            OnResetPressed?.Invoke();
        }
    }
}
```

Considering the mechanics we have implemented till now the only things that need to be reset right now are the pins and the ball but at the same time we want to persist the score.
For this we'll need to add a public function to reset the ball's state in `BallController`. Two things we need to account for now is setting the parent back to the Ball Anchor, and enabling the launch indicator again.

```
using UnityEngine;
using UnityEngine.Events;

[RequireComponent(typeof(Rigidbody))]
public class BallController : MonoBehaviour
{
    .
    .
    .


    void Start()
    {
        ballRB = GetComponent<Rigidbody>();
```

```
        Cursor.lockState = CursorLockMode.Locked;
        inputManager.OnSpacePressed.AddListener(LaunchBall);
        //We bundle the last few lines of code relevant for
        //resetting the state into ResetBall() function
        ResetBall();
    }

    public void ResetBall()
    {
        isBallLaunched = false;

        //We are setting the ball to be a Kinematic Body
        ballRB.isKinematic = true;
        launchIndicator.gameObject.SetActive(true);
        transform.parent = ballAnchor;
        transform.localPosition = Vector3.zero;
    }

    .
    .
    .
    .

}
```

Now, in the `GameManager.cs` script, we'll add a reference for BallController GameObject, PinCollection prefab. We need these references to reset our level. Additionally, create an empty GameObject named PinAnchor, which will serve as the spawn point for new pins. This anchor's position will be used to instantiate fresh pins after the existing ones are destroyed during a level reset. Destroy the existing pins in your scene as we'll spawn new ones when the level starts.

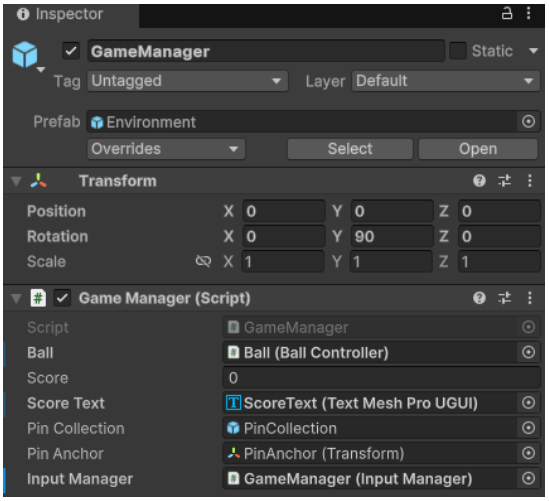



```
using TMPro;
using UnityEngine;

public class GameManager : MonoBehaviour
```

```
{
    [SerializeField] private float score = 0;

    //A reference to our ballController
    [SerializeField] private BallController ball;

    //A reference for our PinCollection prefab we made in Section 2.2
    [SerializeField] private GameObject pinCollection;

    //A reference for an empty GameObject which we'll
    //use to spawn our pin collection prefab
    [SerializeField] private Transform pinAnchor;

    //A reference for our input manager
    [SerializeField] private InputManager inputManager;

    [SerializeField] private TextMeshProUGUI scoreText;
    private FallTrigger[] fallTriggers;
    private GameObject pinObjects;

    private void Start()
    {
        // Adding the HandleReset function as a listener to our
        // newly added OnResetPressedEvent
        inputManager.OnResetPressed.AddListener(HandleReset);
        SetPins();
    }

    private void HandleReset()
    {
        ball.ResetBall();
        SetPins();
    }

    private void SetPins()
    {

        // We first make sure that all the previous pins have been destroyed
        // this is so that we don't create a new collection of
        //standing pins on top of already fallen pins

        if(pinObjects)
        {
            foreach (Transform child in pinObjects.transform)
            {
                Destroy(child.gameObject);
            }

```

```
            Destroy(pinObjects);
        }

        // We then instatiate a new set of pins to our pin anchor transform
        pinObjects = Instantiate(pinCollection,
                                 pinAnchor.transform.position,
                                 Quaternion.identity, transform);

        // We add the Increment Score function as a listener to
        // the OnPinFall event each of new pins
        fallTriggers = FindObjectsByType<FallTrigger>(FindObjectsInactive.Include,
                                                      FindObjectsSortMode.None);
        foreach (FallTrigger pin in fallTriggers)
        {
            pin.OnPinFall.AddListener(IncrementScore);
        }
    }

    private void IncrementScore()
    {
        score++;
        scoreText.text = $"Score: {score}";
    }
}
```

Finally, make sure that all the newly added references have been assigned from the editor



Tip

Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Added Reset Functionality" would suffice.

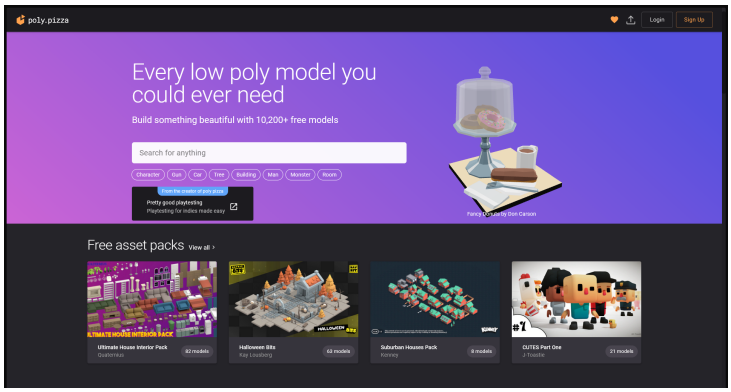
5. Polish

5.1. Importing Assets

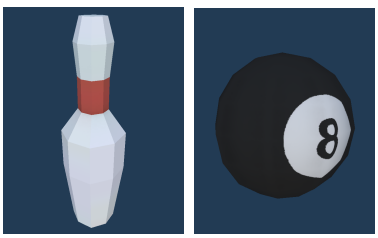
During Unity development, you will have to import various assets throughout your project's development. This can include items such as 3D Models, Audio files, Textures and Images, and so on. When importing assets for personal use, always make sure to acquire assets which you have either personally created, or have a valid license for using. One such popular valid license is the CC0 license, which is "No Rights Reserved".

> **Note**
>
> When importing and using assets for this course, always make sure to use CC0 licensed assets or make your own. You will be uploading your projects to GitHub for grading purposes, which puts your content in the open source domain. You do not want to accidentally infringe on copyright while doing so.

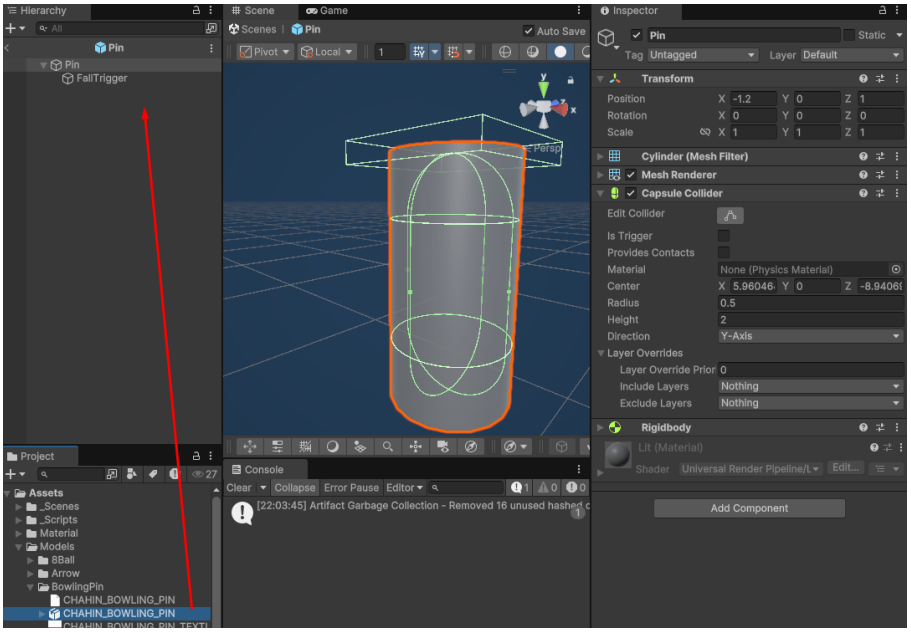
There are many websites which offer free CC0 licensed assets. For this guide, we will be going to poly.pizza which is an awesome repository of various low-poly models which we can use for prototypes.



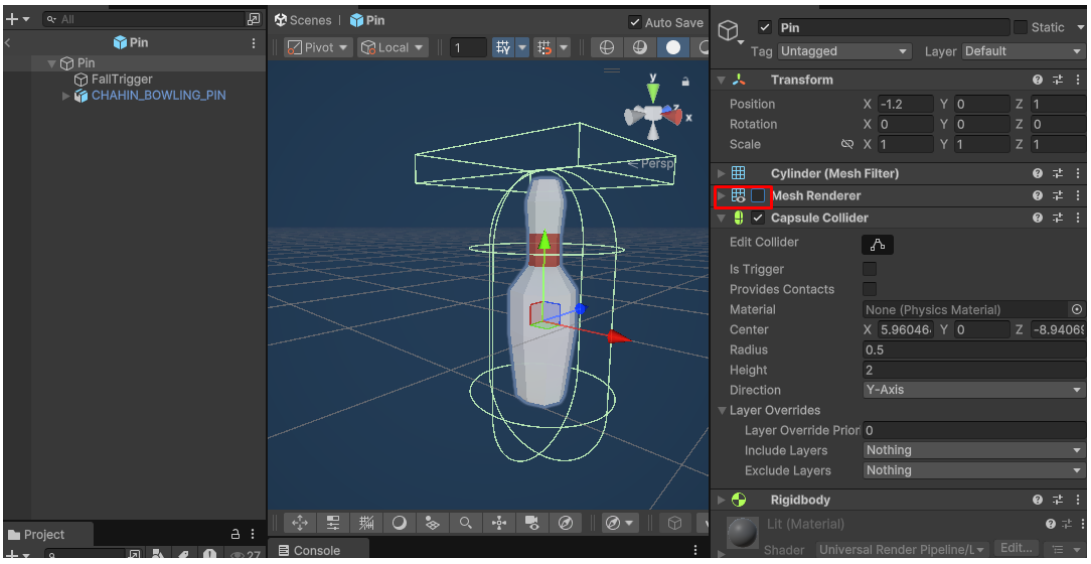
For this guide, we have used this bowling pin asset and this ball asset (since there isn't a specific bowling ball model, this will have to suffice). Download them (or something else you'd like) in **obj** or **fbx** model formats (these two file types are most compatible with different render pipelines and versions of Unity, but others should work as well).



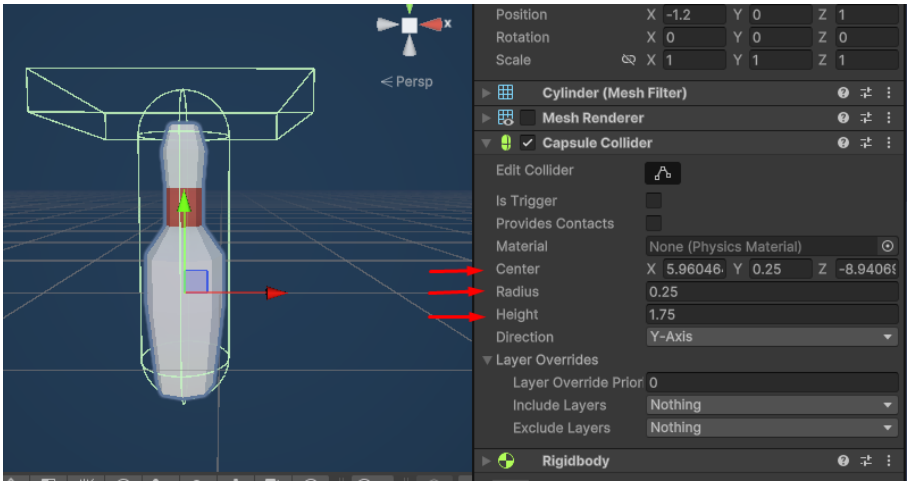
To import them, unzip and simply drag and drop the files into the Unity project window. Now we will be using these models instead of the default Unity objects.
Open your Pin Prefab. Drag and drop the downloaded pin model into the pin prefab.



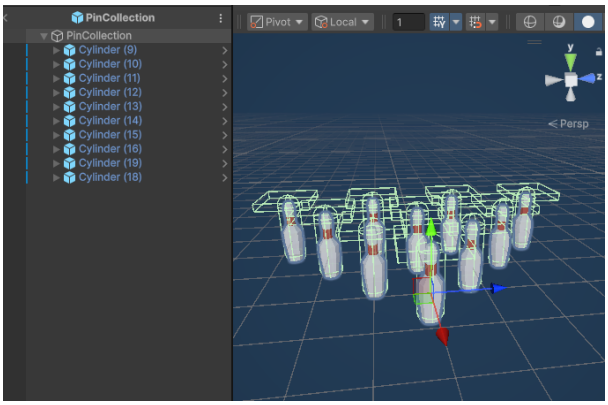
Now, you'll notice that the Cylinder mesh overlaps with the Pin Model. Simply disable the MeshRenderer on the Cylinder to stop rendering the default unity cylinder object. The pin will then be visible.



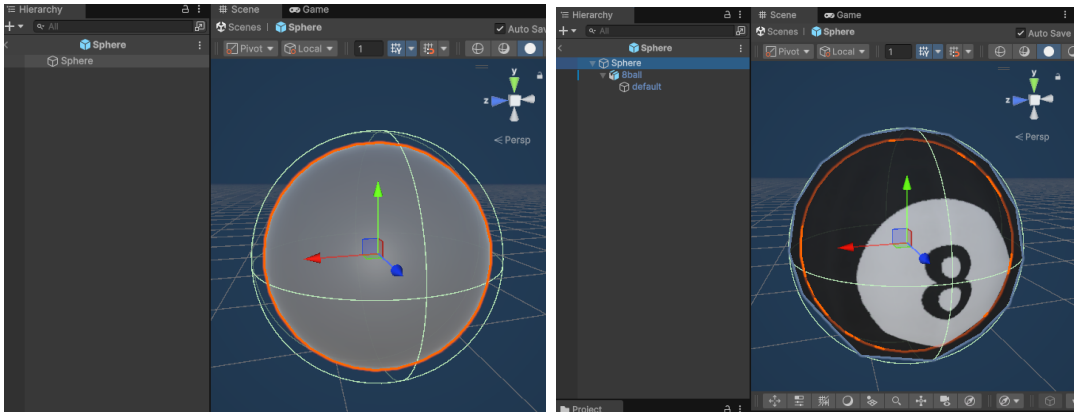
Finally adjust the collider bounds to match the cylinders shape by changing the radius, height and center values.



Now since we have changed the prefab itself, it will apply the change to the pin collection prefab as well.

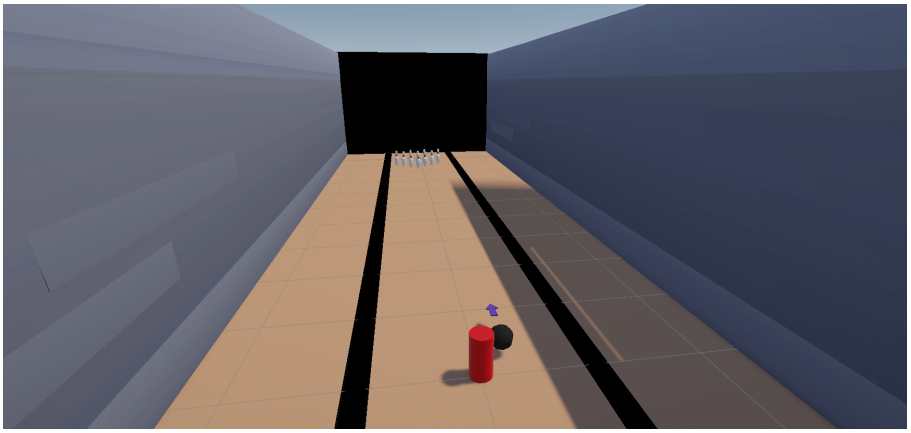


Try to apply the same change on our ball prefab as well.



Hint

Try and find a nice arrow model to use for our Launch Indicator! You can also try to find assets on websites like kenney.nl and quaternius.com



Tip

Now would be a good time to make a git commit. **Remember to save your scene before you make a commit!** Something to the effect of "Updated Models" would suffice.

6. Conclusion

That's all for this guide. At this point you should have a solid fundamental overview of the basic routine of physics and scripting logic with Unity. Hopefully by this time you see the value of having a game engine like Unity do the heavy lifting of rendering graphics, handling physics, organizing scenes, and letting the developer focus on game logic. Implementing a prototype bowling game like this would have taken weeks if we started with trying to figure out 3 dimensional rendering first!

There was a lot that was covered in this guide. It can take some time to wrap your head around a lot of these concepts. Make sure to look for external resources on these topics (remember, the community support and knowledge around Unity is it's biggest strength). Above all else, please don't hesitate to ask us questions and more clarifications during lecture hours (Wednesdays and Thursdays from 10-11).