Since we're checking `IsGround` in the `if` statement every frame in `Update()`, our player's jump skills are only allowed when touching the ground.

That's all you're going to do with the jump mechanic, but the player still needs a way to interact and defend themself against the hordes of enemies that will eventually populate the arena. In the following section, you'll fix that gap by implementing a simple shooting mechanic.

Shooting projectiles

Shooting mechanics are so common that it's hard to think of a first-person game without some variation present, and *Hero Born* is no different. In this section, we'll talk about how to instantiate GameObjects from Prefabs while the game is running, and use the skills we've learned to propel them forward using Unity physics.

Instantiating objects

The concept of instantiating a GameObject in the game is similar to instantiating an instance of a class—both require starting values so that C# knows what kind of object we want to create and where it needs to be created. To create objects in the scene at runtime, we use the `GameObject.Instantiate()` method and provide a Prefab object, a starting position, and a starting rotation.

Essentially, we can tell Unity to create a given object with all its components and scripts at this spot, looking in this direction, and then manipulate it as needed once it's born in the 3D space. Before we instantiate an object, you'll need to create the object Prefab itself, which is your next task.

Before we can shoot any projectiles, we'll need a Prefab to use as a reference, so let's create that now, as follows:

1. Select **+** | **3D Object** | **Sphere** in the **Hierarchy** panel and name it `Bullet`:
    - Change its **Scale** to `0.15` in the x, y, and z axes in the **Transform** component
2. Select the **Bullet** in the **Inspector** and use the **Add Component** button at the bottom to search for and add a **Rigidbody** component, leaving all default properties as they are.
3. Create a new material in the `Materials` folder using **Create** | **Material**, and name it `Bullet_Mat`:
    - Change the **Albedo** property to a deep yellow

- Drag and drop the material from the **Materials** folder onto the `Bullet` GameObject in the **Hierarchy** pane:

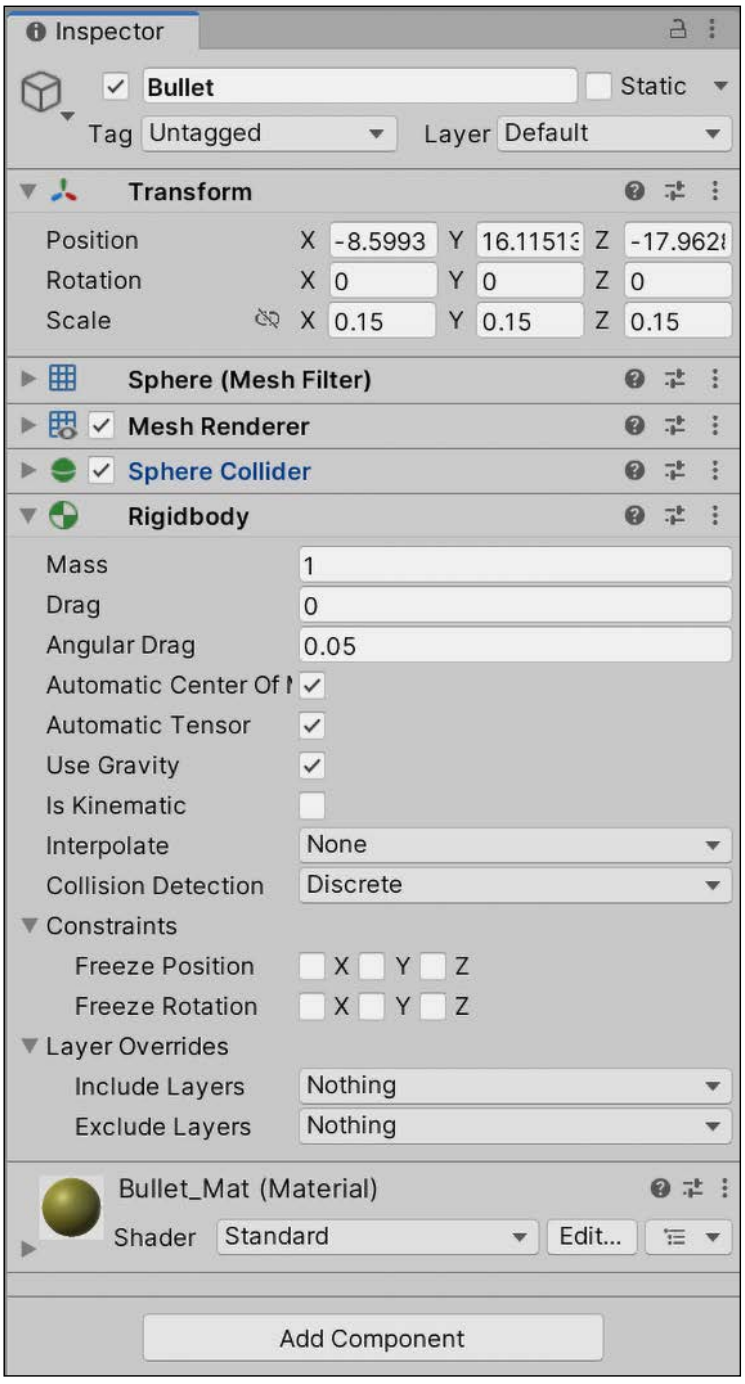

Figure 8.5: Setting projectile properties

4. Select the **Bullet** in the **Hierarchy** panel and drag it into the `Prefabs` folder in the **Project** panel (you can always tell when an object in the **Hierarchy** is a Prefab because it turns blue). Then, delete it from the **Hierarchy** to clean up the scene:

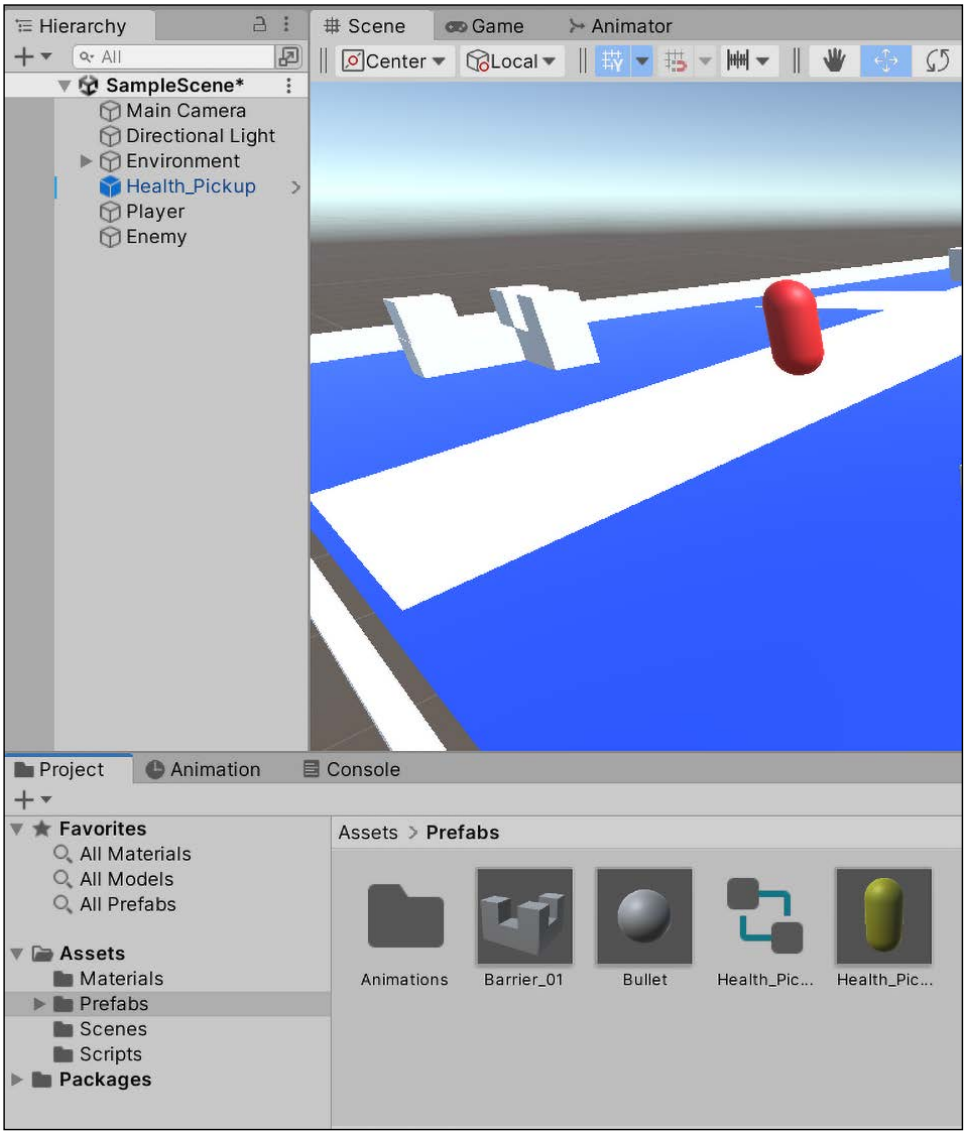

Figure 8.6: Creating a projectile Prefab

You created and configured a **Bullet** Prefab GameObject that can be instantiated as many times as you need in the game and updated as needed. This means you're ready for the next challenge—shooting projectiles.

Adding the shooting mechanic

Now that we have a Prefab object to work with, we can instantiate and move copies of the Prefab whenever we hit the spacebar key to create a shooting mechanic, as follows:

1. Update the `PlayerBehavior` script with the following code:

```
public class PlayerBehavior : MonoBehaviour
{
    // 1
    public GameObject Bullet;
    public float BulletSpeed = 100f;

    // 2
    private bool _isShooting;

    // ... No other variable changes needed ...

    void Start()
    {
        // ... No changes needed ...
    }

    void Update()
    {
        // 3
        _isShooting |=  Input.GetKeyDown(KeyCode.Space);
        // ... No other changes needed ...
    }

    void FixedUpdate()
    {
        // ... No other changes needed ...

        // 4
        if (_isShooting)
        {
            // 5
            GameObject newBullet = Instantiate(Bullet,
```

```
                this.transform.position + new Vector3(0, 0, 1),
                    this.transform.rotation);
            // 6
            Rigidbody BulletRB =
                 newBullet.GetComponent<Rigidbody>();

            // 7
            BulletRB.velocity = this.transform.forward *
                                                BulletSpeed;
        }

        // 8
        _isShooting = false;
    }

    private bool IsGrounded()
    {
        // ... No changes needed ...
    }
}
```

2. In the **Inspector**, drag the **Bullet** Prefab from the **Project** panel into the **Bullet** property of `PlayerBehavior`, as illustrated in the following screenshot:

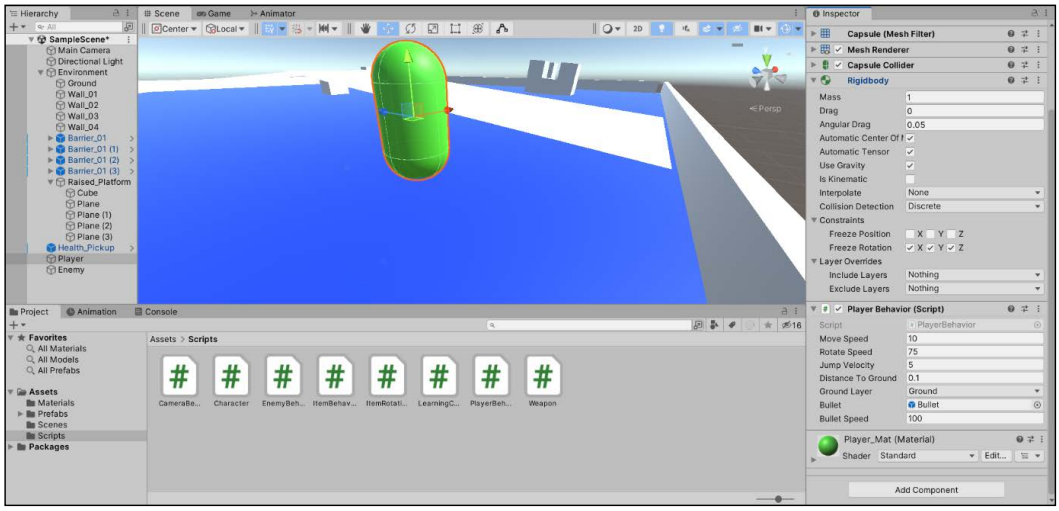

Figure 8.7: Setting the Bullet Prefab

3. Play the game and use the left mouse button to fire projectiles in the direction the player is facing!

Let's break down the code as follows:

1. We create two variables: one to store the **Bullet** Prefab, the other to hold the **Bullet** speed. The best practice is to always declare new variables as private unless there's a good reason to make them public.
2. Like our jumping mechanic, we use a Boolean in the `Update` method to check if our player should be shooting.
3. We set the value of `_isShooting` using the `or` logical operator and `Input.GetKeyDown(KeyCode.Space)`, just like we did for the jumping mechanic. Then, we check if our player is supposed to be shooting using the `_isShooting` variable.
4. We create a local GameObject variable every time the left mouse button is pressed:
    - We use the `Instantiate()` method to assign a GameObject to `newBullet` by passing in the `Bullet` Prefab. We also use the player capsule's position to place the new `Bullet` Prefab in front of the player (one unit forward along the *z* axis) to avoid any collisions.
    - We append it as a `GameObject` to explicitly cast the returned object to the same type as `newBullet`, which in this case is a GameObject.
5. We call `GetComponent()` to return and store the **Rigidbody** component on `newBullet`.
6. We set the `velocity` property of the **Rigidbody** component to the player's `transform.forward` direction multiplied by `BulletSpeed`:
    - Changing the `velocity` instead of using `AddForce()` ensures that gravity doesn't pull our bullets down in an arc when fired
7. Finally, we set the `_isShooting` value to `false` so our shooting input is reset for the next input event.

Again, you've significantly upgraded the logic the player script is using. You should now be able to use the mouse to shoot projectiles that fly straight out from the player's position.

However, the problem now is that your game scene, and **Hierarchy**, is flooded with spent **Bullet** objects. Your next task is to clean those objects up once they've been fired, to avoid any performance issues.

Managing object build-up

Whether you're writing a completely code-based application or a 3D game, it's important to make sure that unused objects are regularly deleted to avoid overloading the program. Our bullets don't exactly play an important role after they are shot; they just keep existing on the floor near whatever wall or object they collided with.

With a mechanic such as shooting, this could result in hundreds, if not thousands, of bullets down the line, which is something we don't want. Your next challenge is to destroy each bullet after a set delay time.

For this task, we can take the skills we've already learned and make the bullets responsible for their self-destructive behavior, as follows:

1. Create a new C# script in the `Scripts` folder and name it `BulletBehavior`.
2. Drag and drop the `BulletBehavior` script onto the `Bullet` Prefab in the `Prefabs` folder and add the following code:

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

public class BulletBehavior : MonoBehaviour
{
    // 1
    public float OnscreenDelay = 3f;

    void Start ()
    {
        // 2
        Destroy(this.gameObject, OnscreenDelay);
    }
}
```

Let's break down this code, as follows:

1. We declare a `float` variable to store how long we want the **Bullet** Prefabs to remain in the scene after they are instantiated.
2. We use the `Destroy()` method to delete the GameObject:

- `Destroy()` always needs an object as a parameter. In this case, we use the `this` keyword to specify the object that the script is attached to.
- `Destroy()` can optionally take an additional `float` parameter as a delay, which we use to keep the bullets on screen for a short amount of time.

Play the game again, shoot some bullets, and watch as they are deleted from the **Hierarchy** by themselves in the scene after a specific delay. This means that the bullet executes its defined behavior, without another script having to tell it what to do, which is an ideal application of the *Component* design pattern.

Now that our housekeeping is done, you're going to learn about a key component of any well-designed and organized project—the manager class.

Creating a game manager

A common misconception when learning to program is that all variables should automatically be made public, but in general, this is not a good idea. In my experience, variables should be thought of as protected and private from the start, and only made public if necessary. One way you'll see experienced programmers protect their data is through manager classes, and since we want to build good habits, we'll be following suit. Think of manager classes as a funnel where important variables and methods can be accessed safely.

When I say safely, I mean just that, which might seem unfamiliar in a programming context. However, when you have different classes communicating and updating data with each other, things can get messy. That's why having a single contact point, such as a manager class, can keep this to a minimum. We'll get into how to do that effectively in the following section.

Tracking player properties

Hero Born is a simple game, so the only two data points we need to keep track of are how many items the player has collected and how much health they have left. We want these variables to be private so that they can only be modified from the manager class, giving us control and safety. Your next challenge is to create a game manager for *Hero Born* and populate it with helpful functionality.

Game manager classes will be a constant facet of any project you develop in the future, so let's learn how to properly create one, as follows:

1. Create a new C# script in the `Scripts` folder and name it `GameBehavior`.