unity

Bonus Features 4

Solution Walkthrough

Easy - Harder Enemy 2

Medium - Homing Rockets 6

Hard - Smashingly Good 21

Expert - Boss Battle 29

Easy - Harder Enemy

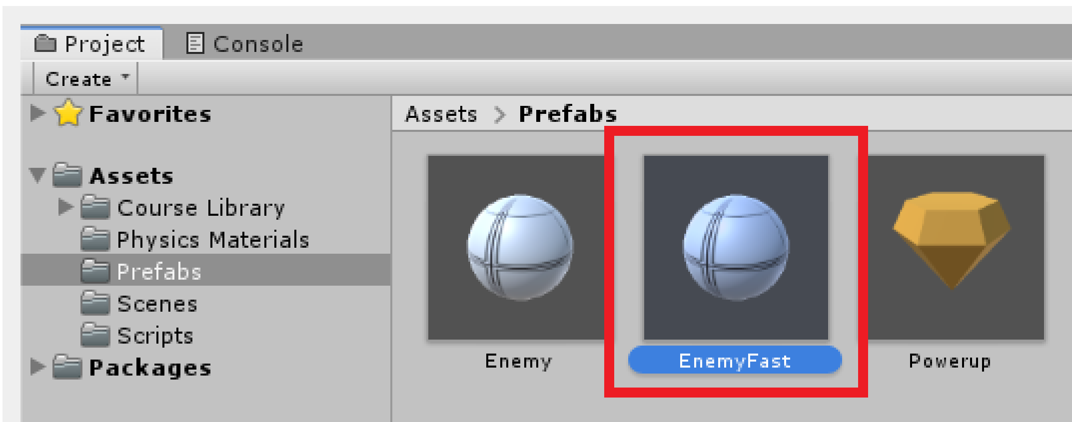
1. Navigate to the Scripts folder in the Project window and open up **SpawnManager.cs**. On line 7, change the *enemyPrefab* variable to be an array and rename it to *enemyPrefabs*.

```
public GameObject[] enemyPrefabs;
```

2. In the SpawnEnemyWave method, we need to determine which prefab from the array we will spawn. We can do this using Random.Range. The updated method looks like this:

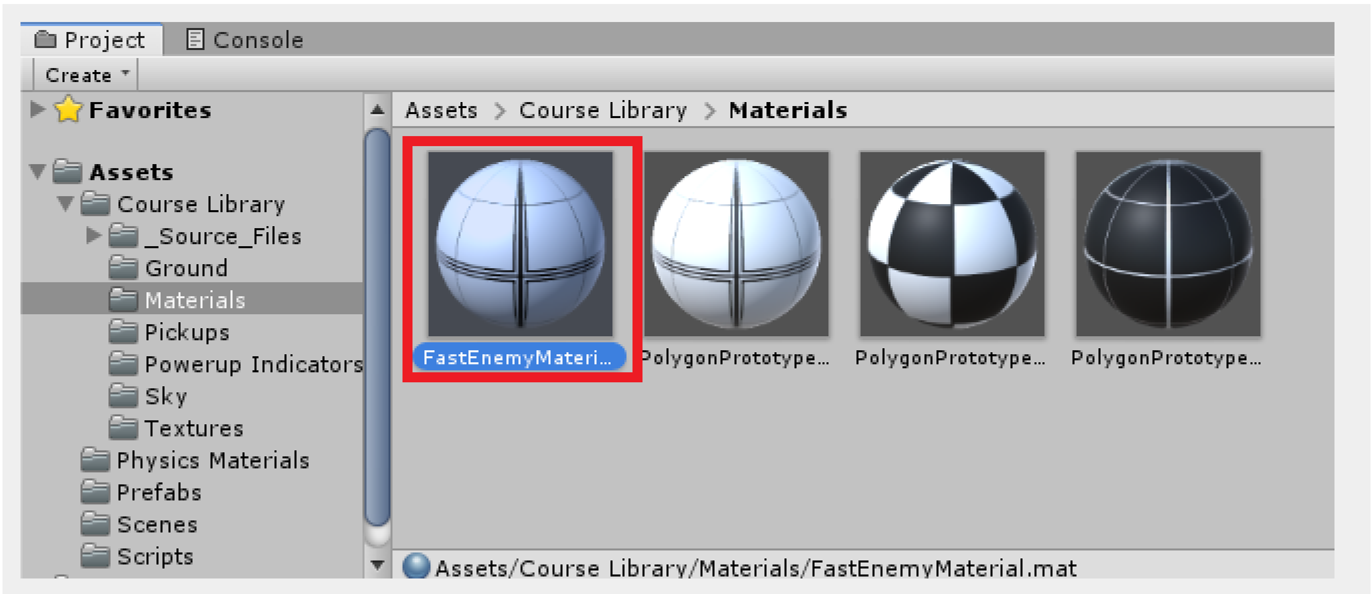
```
void SpawnEnemyWave(int enemiesToSpawn)
{
    for (int i = 0; i < enemiesToSpawn; i++)
    {
        int randomEnemy = Random.Range(0, enemyPrefab.Length);

        Instantiate(enemyPrefabs[randomEnemy], GenerateSpawnPosition(),
enemyPrefabs[randomEnemy].transform.rotation);
    }
}
```

3. Save the script and head back to Unity. Navigate to the Prefabs folder in the Project window. Left click on the **Enemy** prefab and then press **Ctrl + D** (PC) or **Cmd + D** (Mac) to duplicate the prefab. Rename it to "*EnemyFast*".



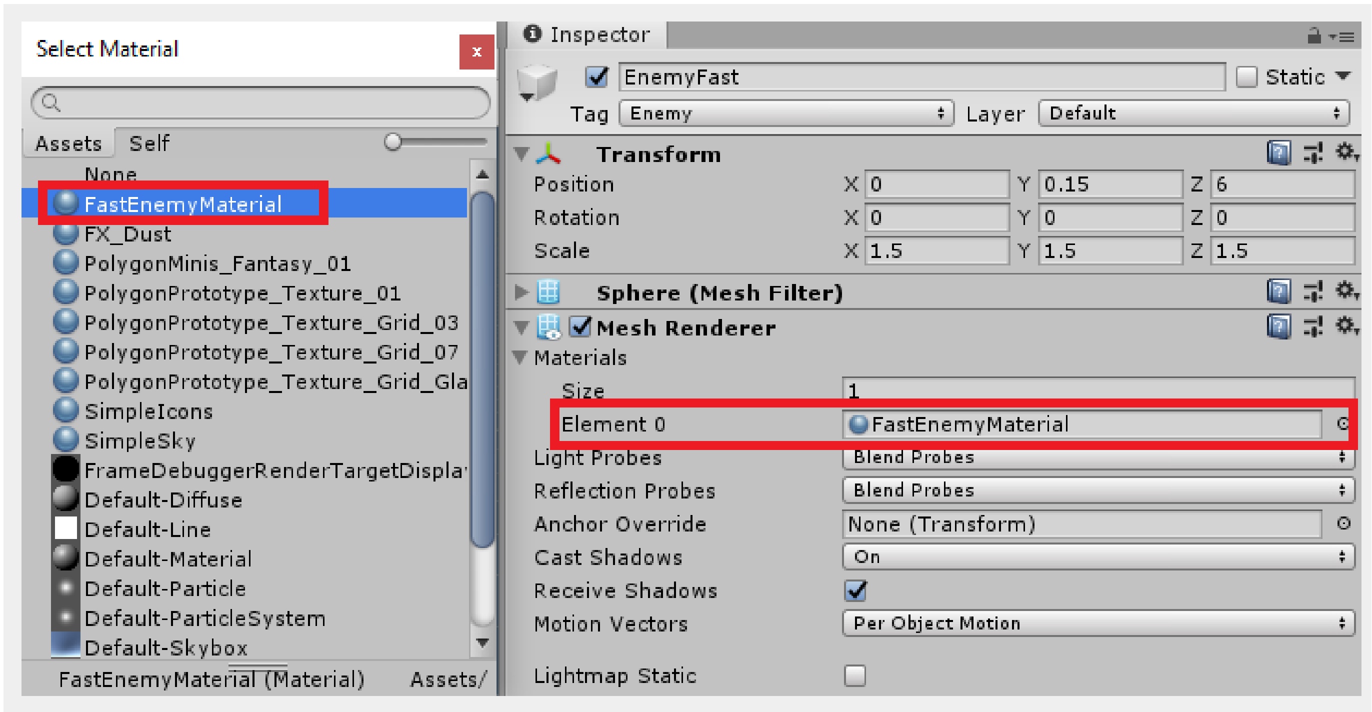
4. Select the *EnemyFast* prefab and in the Inspector adjust the **Speed** parameter of the **Enemy (Script) component**.

5. In the Project view, Navigate to **Assets** > **Course Library** > **Materials**. Select the Material called "*PolygonPrototype_Texture_Grid_03*" and press **Ctrl + D** (PC) or **Cmd + D** (Mac) to duplicate it. Rename it to *"FastEnemyMaterial"*.

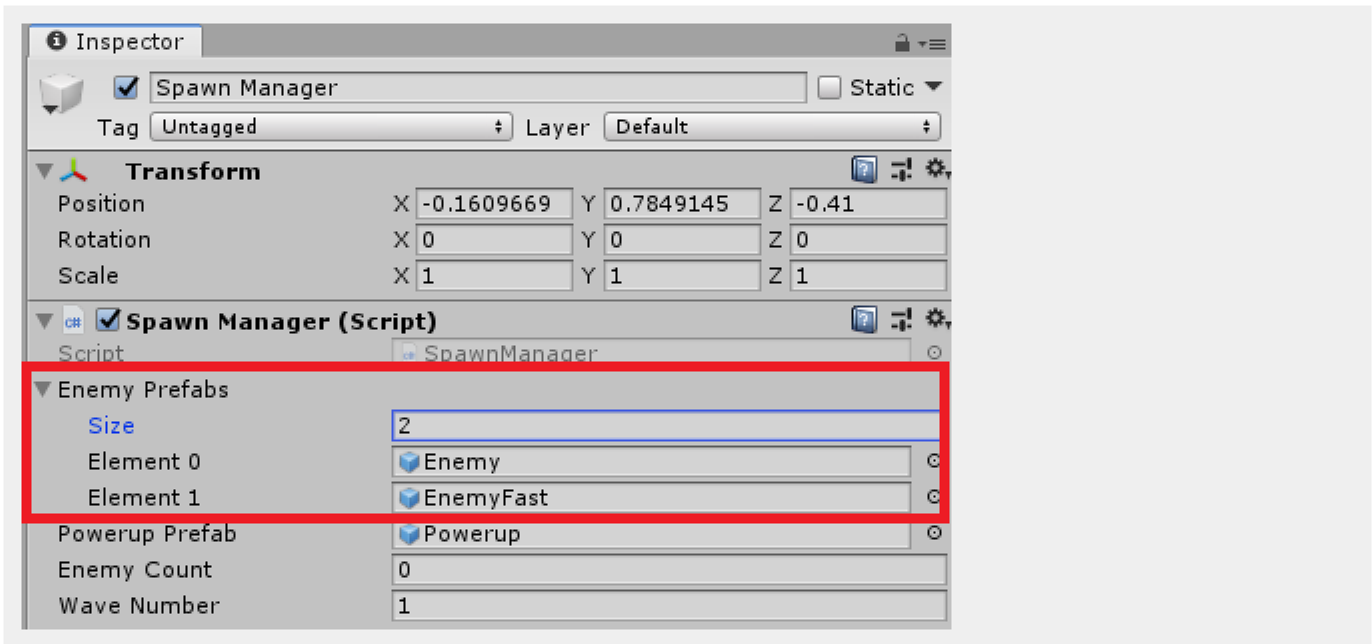


6. Select the new Material, and in the Inspector change the **Albedo Color** value to blue.

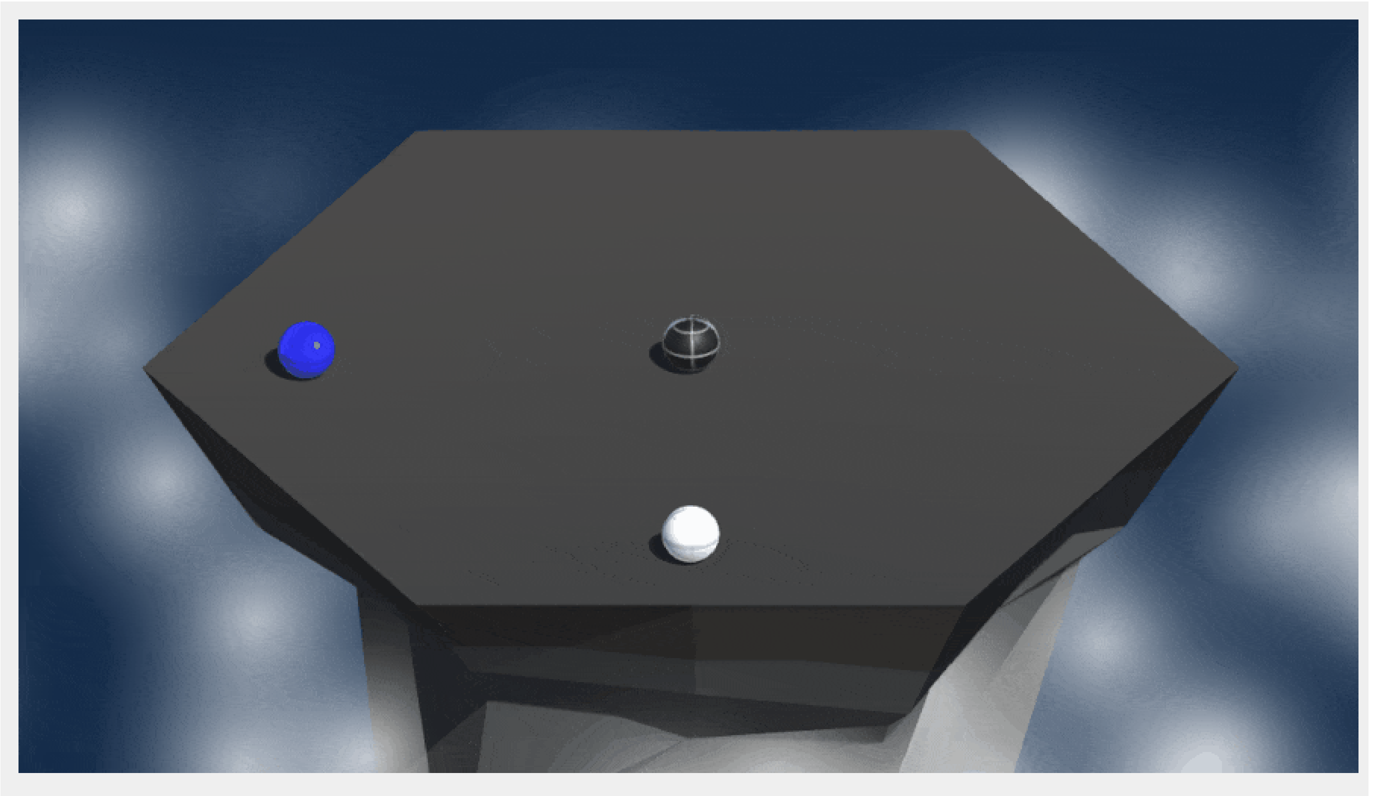
7. Go back to the *EnemyFast* prefab and assign the new Material to the **Mesh Renderer Material** parameter.



8. Select the *SpawnManager* in the Hierarchy. On the **Spawn Manager (Script)** component, change the size of the **Enemy Prefab** to **2** and assign the two enemy prefabs to the empty slots.

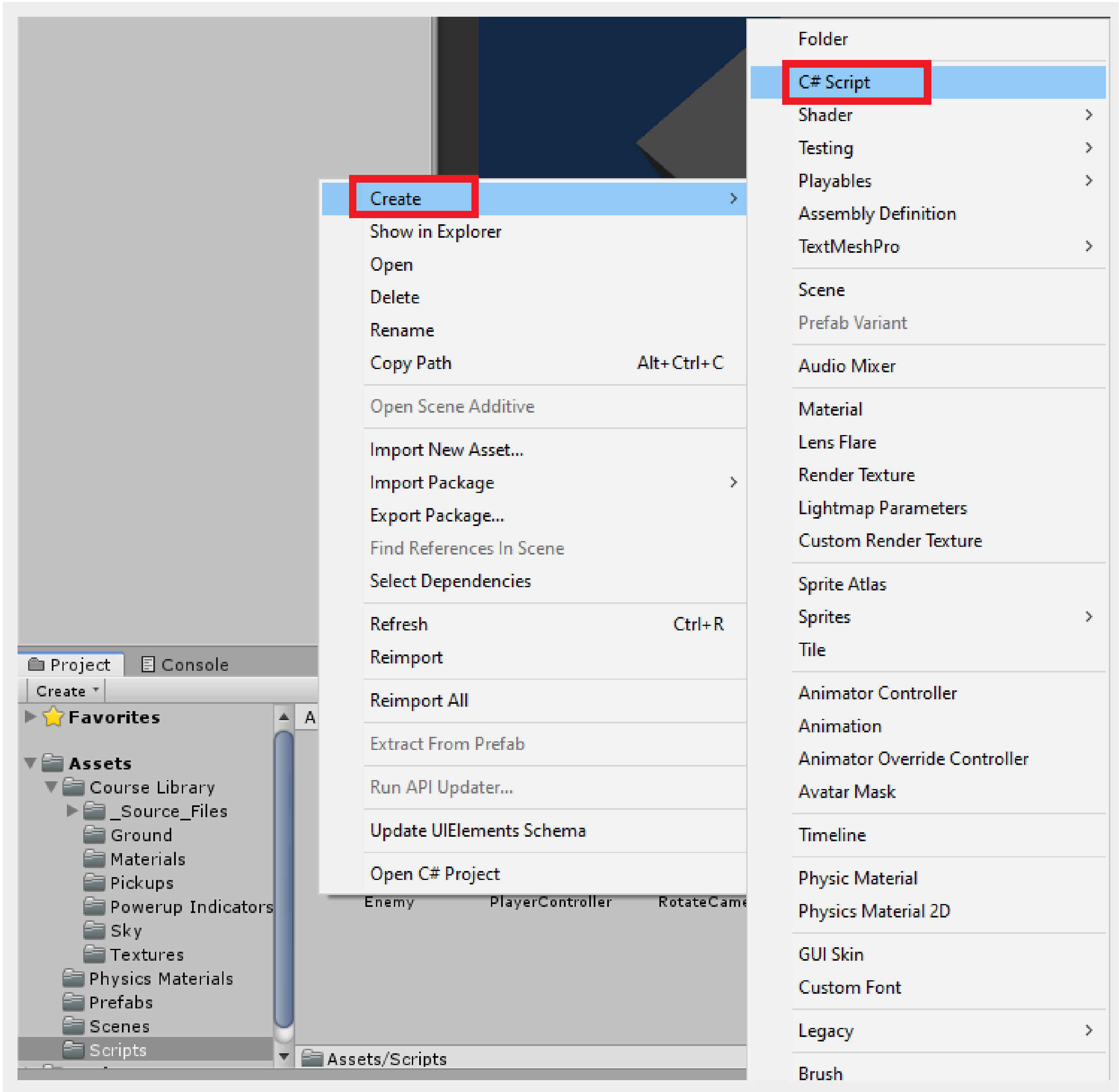


9. Save the scene and test out the game. There is now an enemy that rolls towards you faster.



Medium - Homing Rockets

1. Let's first create a script for our Powerups. In the Project window, navigate to the scripts folder. Right-click and select **Create** > **C# Script**. Name this script "*PowerUp*".



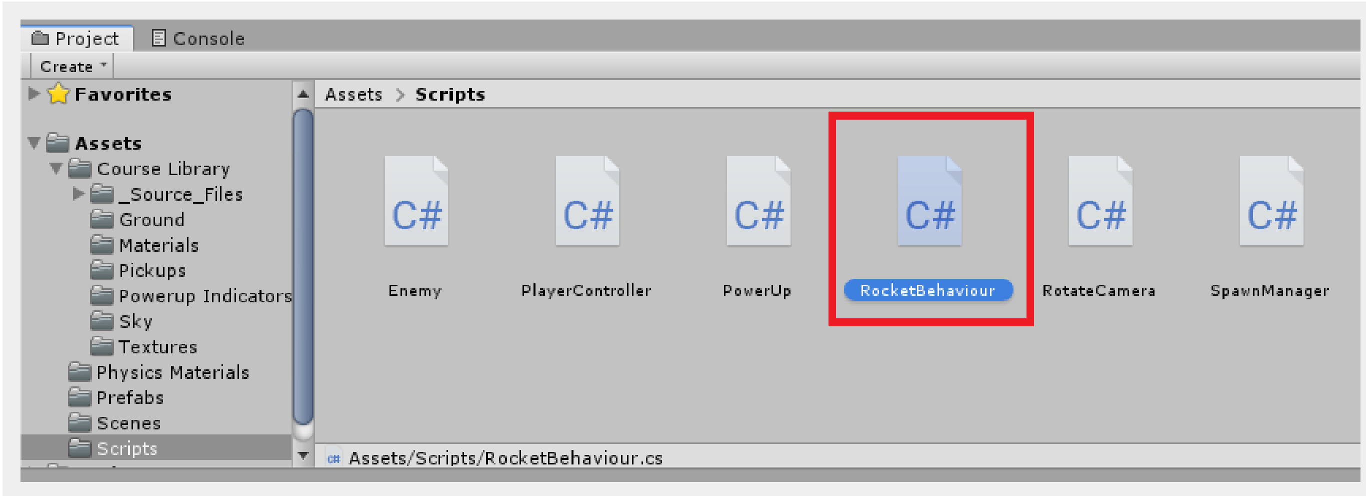
2. Double-click the script to open it. We're using an enum for the power up type to allow for more power ups to be added easily. Replace the contents of the script with this:

```
using UnityEngine;

public enum PowerUpType { None, Pushback, Rockets }
```

```
public class PowerUp : MonoBehaviour
{
    public PowerUpType powerUpType;
}
```

Save the script and head back to Unity.

3. In the Scripts folder, create a new script called *RocketBehaviour*. This script will hold the logic for the rocket.



4. The first thing we need to do in this script is set up the variables. Below the class definition, add the following:

```
private Transform target;
private float speed = 15.0f;
private bool homing;

private float rocketStrength = 15.0f;
private float aliveTimer =  5.0f;
```

5. Delete the **Start** method as we won't be using it. Below the **Update** method, create a new method called *Fire*. This will be called by our player when we spawn in the rockets, so needs to be public.

```
public void Fire(Transform newTarget)
{
    target = homingTarget;
    homing = true;
    Destroy(gameObject, aliveTimer);
}
```

This method takes in a Transform that we will set as the target. It will set the homing boolean to true and then set the GameObject to be destroyed after 5 seconds (as defined by aliveTimer).

6. Next we will set up the code for moving and rotating the missile towards the target. Update the **Update** method to look like the following:

```
void Update()
{
    if(homing && target != null)
    {
        Vector3 moveDirection = (target.transform.position -
transform.position).normalized;
        transform.position += moveDirection * speed * Time.deltaTime;
        transform.LookAt(target);
    }
}
```

7. Now we will add in the **OnCollisionEnter** method. This will add a force to whatever is hit.

```
void OnCollisionEnter(Collision col)
{
    if (target != null)
    {
        if (col.gameObject.CompareTag(target.tag))
        {
            Rigidbody targetRigidbody = col.gameObject.GetComponent<Rigidbody>();
            Vector3 away = -col.contacts[0].normal;
            targetRigidbody.AddForce(away * rocketStrength, ForceMode.Impulse);
            Destroy(gameObject);
        }
    }
}
```

This method first checks if we have a target. If we do, we compare the tag of the colliding object with the tag of the target. If they match, we get the rigidbody of the target. We then use the normal of the collision contact to determine which direction to push the target in. Finally we apply the force to the target and destroy the missile.

Save the script and head back to Unity.

8. Open up the **PlayerController.cs** script. We will need some new variables. Below the current variable declarations, add the following:

```
public PowerUpType currentPowerUp = PowerUpType.None;

public GameObject rocketPrefab;
private GameObject tmpRocket;
private Coroutine powerupCountdown;
```

You will notice that we are using the **PowerUpType** enum here. This is used to help determine which logic to enable for the player when a power up is collected.
We are also declaring two GameObject variables. The public one is used for the homing rocket prefab. The private one will be used for spawning in the homing rockets.

9. To add in our new logic, we need to update the **OnTriggerEnter** method. The new lines are highlighted

below.

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Powerup"))
    {
        hasPowerup = true;
        currentPowerUp = other.gameObject.GetComponent<PowerUp>().powerUpType;
        powerupIndicator.gameObject.SetActive(true);
        Destroy(other.gameObject);

        if(powerupCountdown != null)
        {
            StopCoroutine(powerupCountdown);
        }
        powerupCountdown = StartCoroutine(PowerupCountdownRoutine());
    }
}
```

10. We will need to have a way to return our currentPowerUp to none, we can do this within the **PowerupCountdownRoutine** Method.

```
IEnumerator PowerupCountdownRoutine()
{
    yield return new WaitForSeconds(7);
    hasPowerup = false;
    currentPowerUp = PowerUpType.None;
    powerupIndicator.gameObject.SetActive(false);
}
```

11. We are also going to need to change the logic in the **OnCollisionEnter** method. We need to replace the hasPowerup boolean check with a currentPowerUp check. We will also update the Debug.Log output to include the power up we are using. The updated OnCollisionEnter method will look like the following:

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Enemy") && currentPowerUp ==
PowerUpType.Pushback)
    {
        Rigidbody enemyRigidbody = collision.gameObject.GetComponent<Rigidbody>();
        Vector3 awayFromPlayer = collision.gameObject.transform.position -
transform.position;
        enemyRigidbody.AddForce(awayFromPlayer * powerUpStrength,
ForceMode.Impulse);
        Debug.Log("Player collided with: " + collision.gameObject.name + " with
powerup set to " + currentPowerUp.ToString());
    }
}
```

12. After the **OnCollisionEnter** method, add the following:

```
void LaunchRockets()
{
    foreach(var enemy in FindObjectsOfType<Enemy>())
    {
        tmpRocket = Instantiate(rocketPrefab, transform.position + Vector3.up,
Quaternion.identity);
        tmpRocket.GetComponent<RocketBehaviour>().Fire(enemy.transform);
    }
}
```

Here we are using the same logic as our spawn manager to find all the enemies. We are then launching our missiles at each one. We launch the missiles from above the player, to stop the collision from pushing us back.

13. The last thing we need to do in the **PlayerController**, is add the logic to the **Update** method. We're going to check if the currentPowerUp is the rocket, and if the "**F**" key is pressed. If both are true, we will call the **LaunchRockets** method.

```
void Update()
{
    float forwardInput = Input.GetAxis("Vertical");
    playerRb.AddForce(focalPoint.transform.forward * forwardInput * speed);
    powerupIndicator.transform.position = transform.position + new Vector3(0,
-0.5f, 0);

    if (currentPowerUp == PowerUpType.Rockets && Input.GetKeyDown(KeyCode.F))
    {
        LaunchRockets();
    }
}
```

Save the script and return to Unity

14. Now that we have the logic set up for the missiles, we will need to adjust the SpawnManager to allow for two different types of power ups. Double click **SpawnManager.cs** to open it up. Find the variable named powerUpPrefab and change it to the following:

```
public GameObject[] powerupPrefabs;
```

15. Find the **Start** method and adjust it to look like the following:

```
void Start()
{
    int randomPowerup = Random.Range(0, powerupPrefabs.Length);
    Instantiate(powerupPrefabs[randomPowerup], GenerateSpawnPosition(),
powerupPrefabs[randomPowerup].transform.rotation);
    SpawnEnemyWave(waveNumber);
}
```

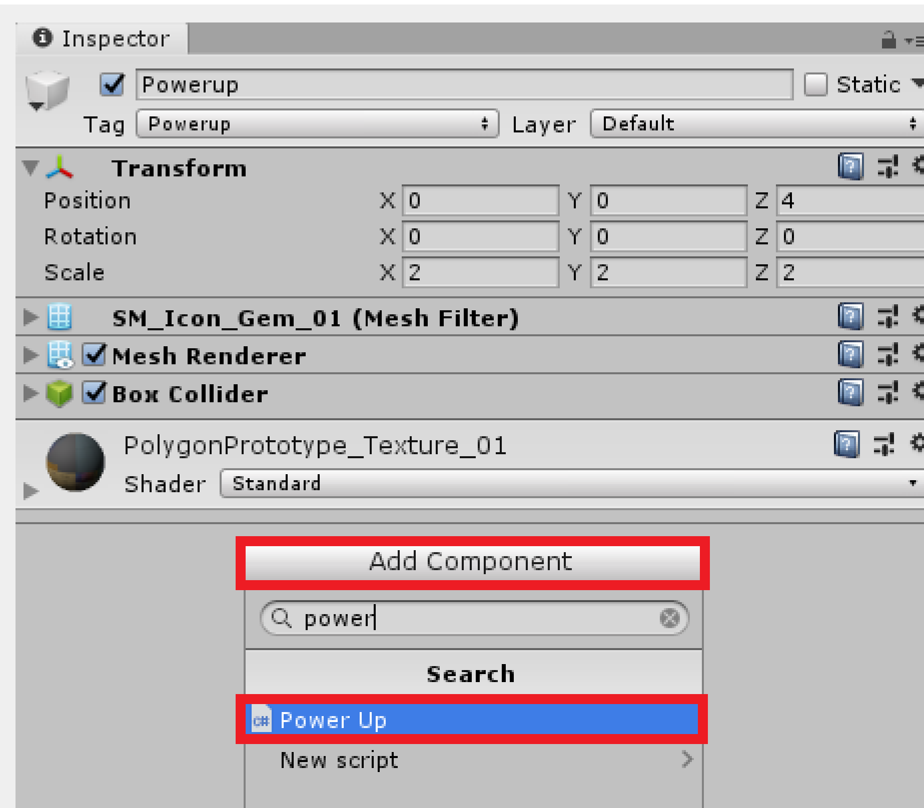
What we are doing here, is getting a random number based on the length of the powerupPrefabs array, and then spawning the object at that position in the array.

We'll also need to add the same code to replace the existing powerup spawning in the Update method. The Update method should look like this:

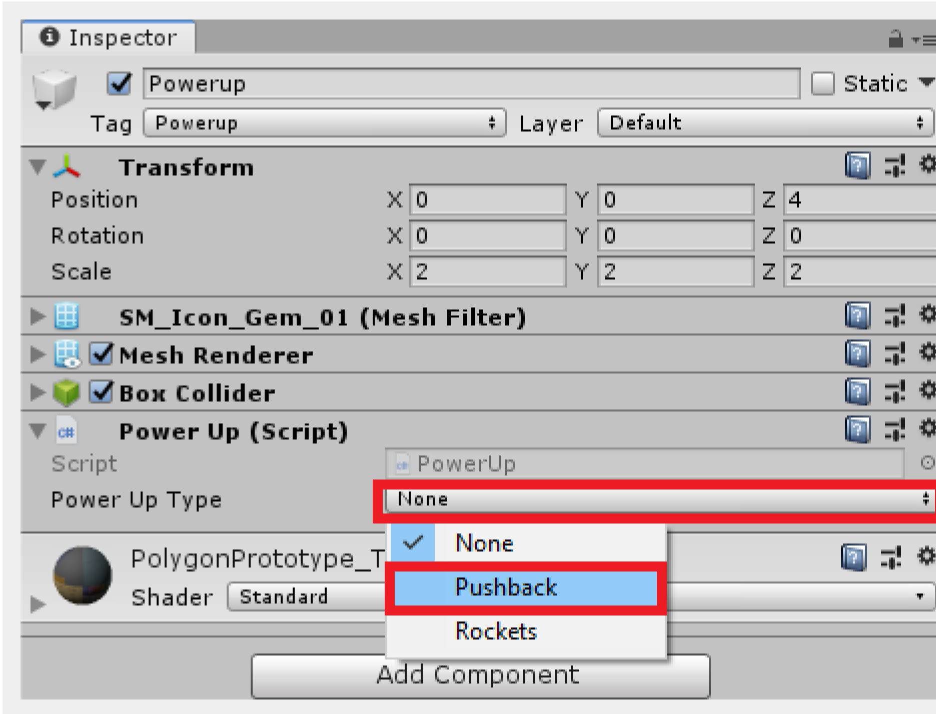
```
void Update()
{
    enemyCount = FindObjectsOfType<Enemy>().Length;
    if (enemyCount == 0)
    {
        waveNumber++;
        SpawnEnemyWave(waveNumber);
        int randomPowerup = Random.Range(0, powerupPrefabs.Length);
        Instantiate(powerupPrefabs[randomPowerup], GenerateSpawnPosition(),
powerupPrefabs[randomPowerup].transform.rotation);
    }
}
```

Save the script and head back to Unity.

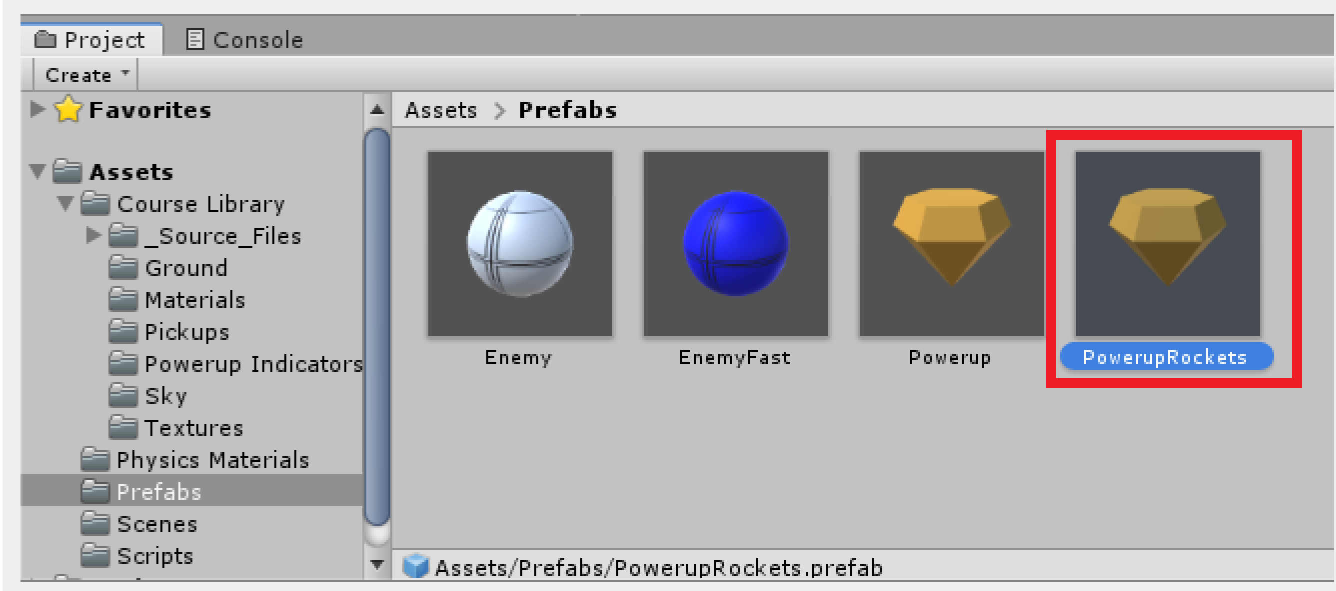
16. Navigate to the Prefabs folder and select the *Powerup* prefab. In the Inspector, click **Add Component** and search for the **PowerUp** script.



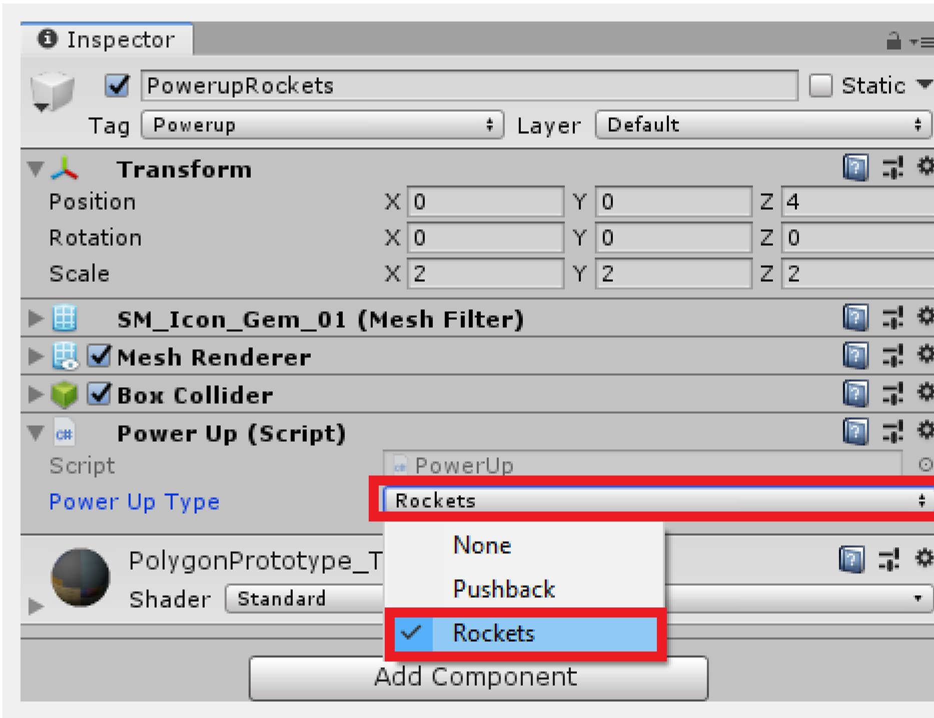
After adding the **PowerUp** script, change the **Power Up Type field** to **Pushback**.



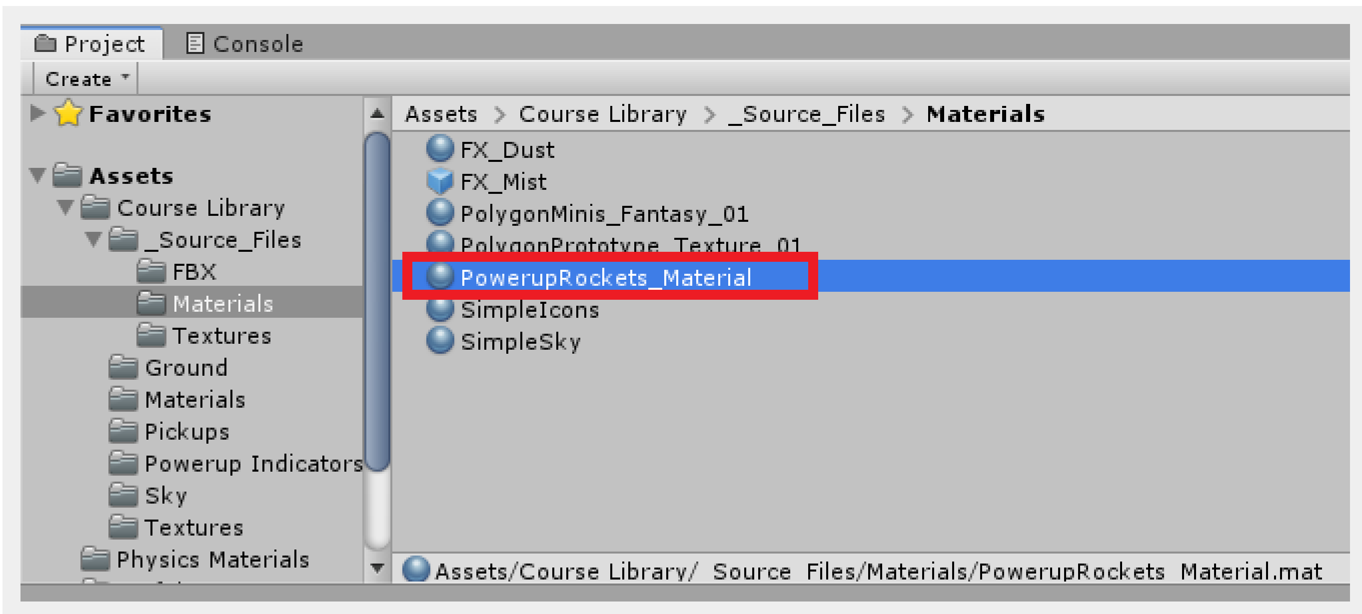
17. Duplicate the *Powerup* prefab in the project view, and rename the duplicate to *"PowerupRockets"*.



In the inspector for the *PowerupRockets*, change the **Power Up Type** to **Rockets**.

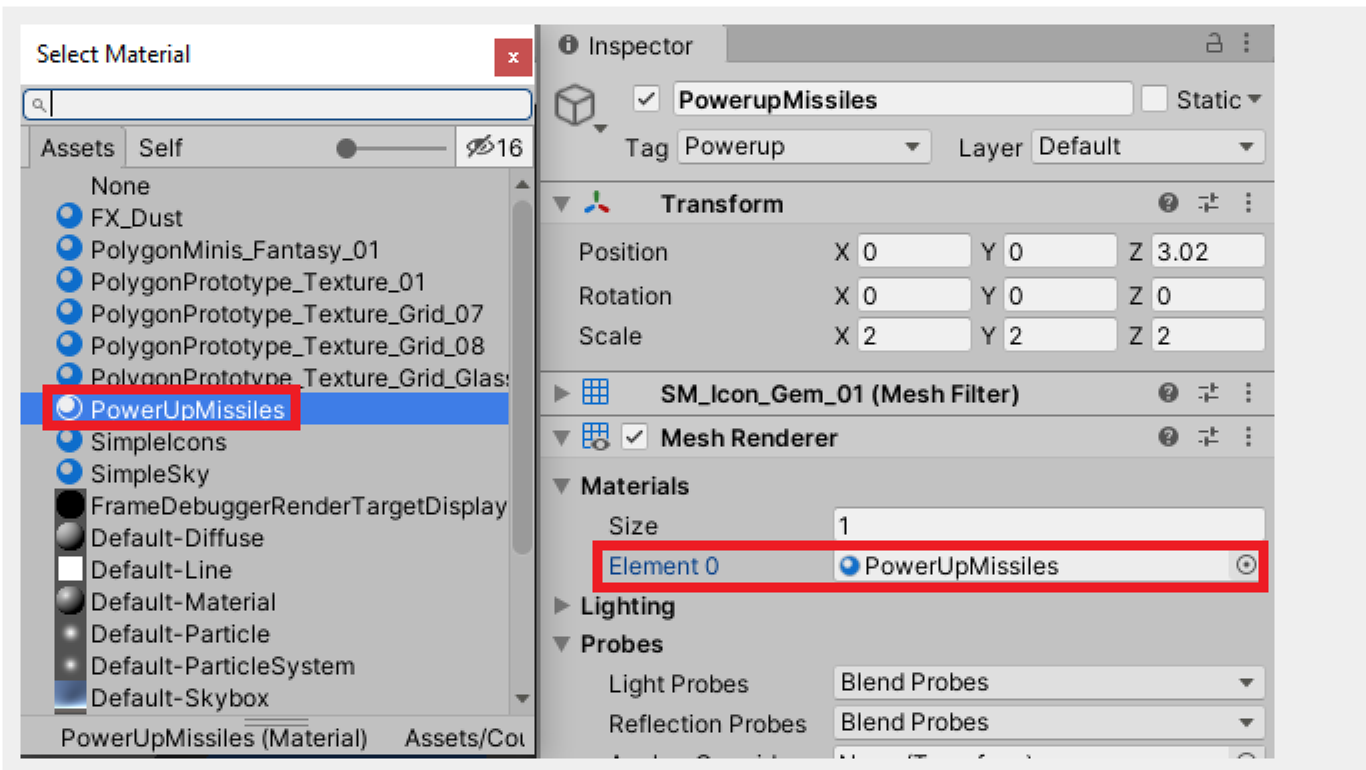


18. In the Project window, Navigate to **Assets** > **Course Library** > **_Source_Files** > **Materials**. Duplicate the Material called *PolygonPrototype_Texture_01* and rename it to *PowerUpRockets_Material*.

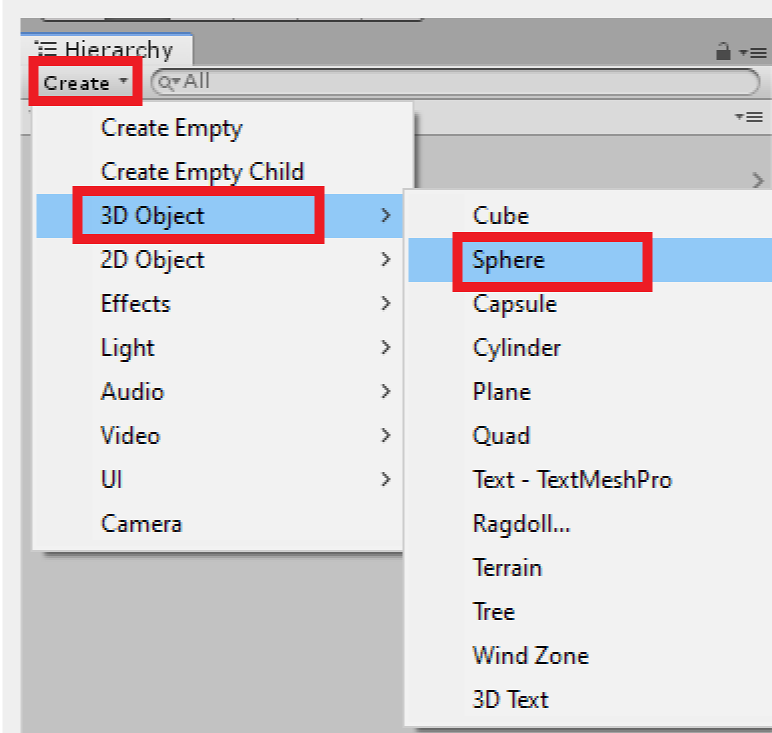


19. In the Inspector for the Material called *PowerUpRockets_Material*, adjust the Albedo Color. This will help the player distinguish the power ups from each other.

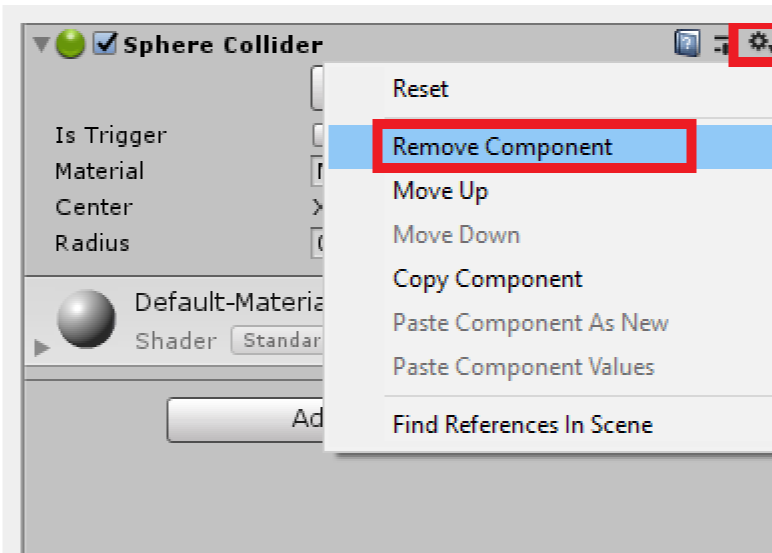
20. Navigate to **Assets** > **Prefabs**, and select the **PowerUpMissiles** prefab. In the Inspector, change the **Material** on the **Mesh Renderer** component to the PowerUpMissiles Material that was just created.



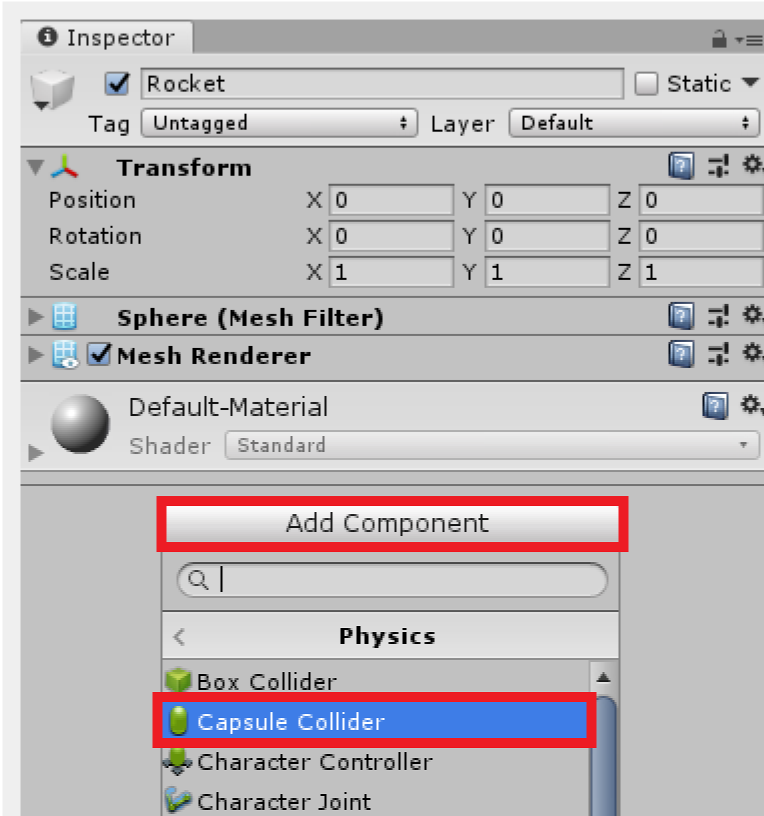
21. Next, we are going to set up the missile prefab. In the Hierarchy, click **Create** > **3D Object** > **Sphere**. Rename the sphere to *Rocket*.



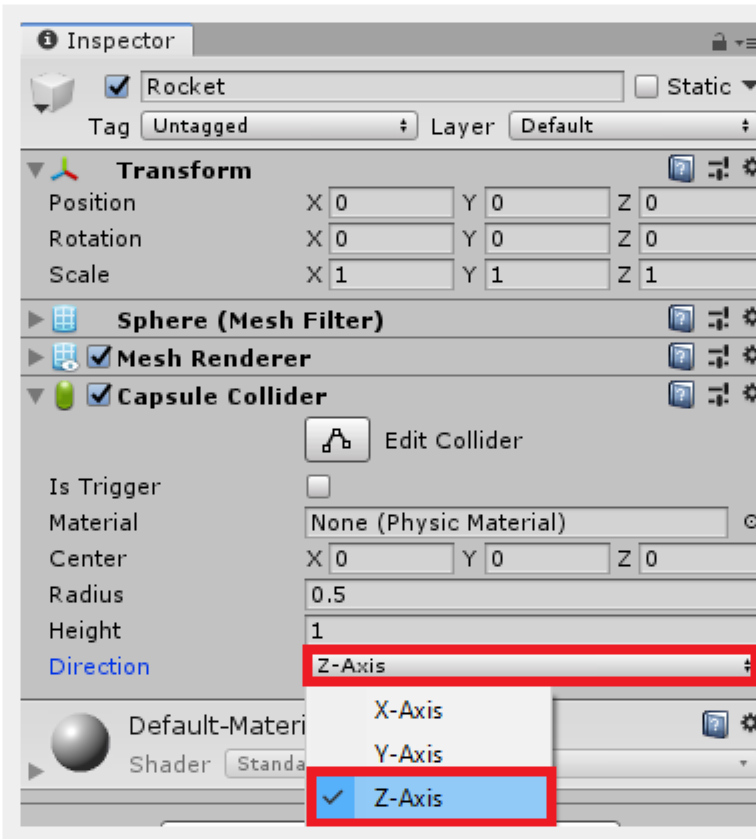
22. Instead of a Sphere Collider, we will use a Capsule Collider as we will be adjusting the scale of the GameObject. On the Rocket GameObject, Remove the **Sphere Collider** by clicking the **Gear icon** in the top right of the component and selecting **Remove Component**.



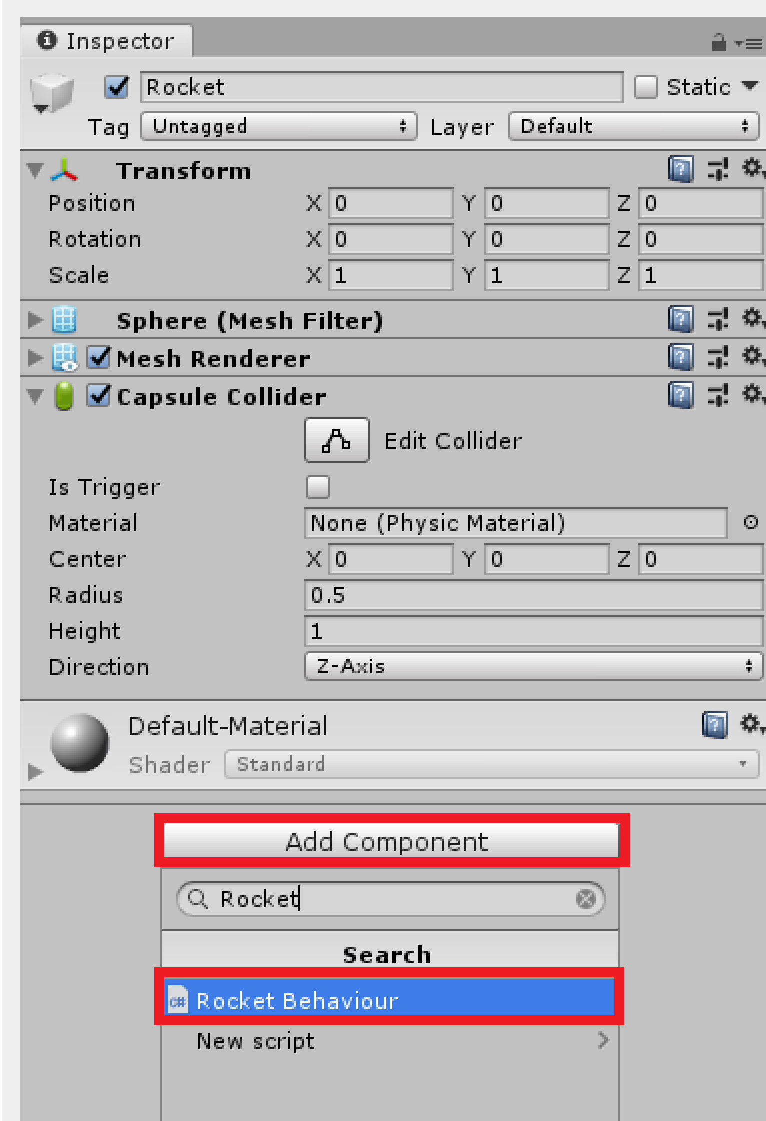
23. Add a **Capsule Collider** by selecting **Add Component** > **Physics** > **Capsule Collider**



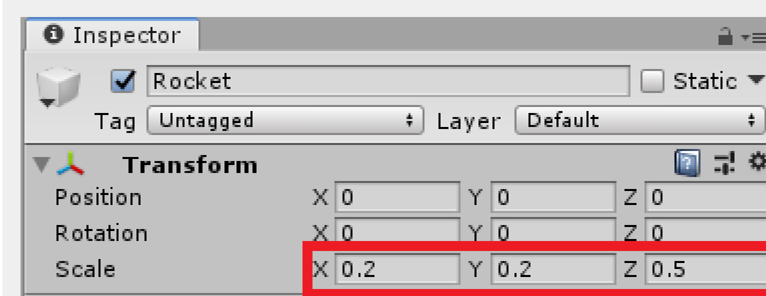
24. On the Capsule Collider, Change the **Direction** to **Z-axis**. We do this because our code is going to rotate the missile to face the enemy using the forward (Z) direction.



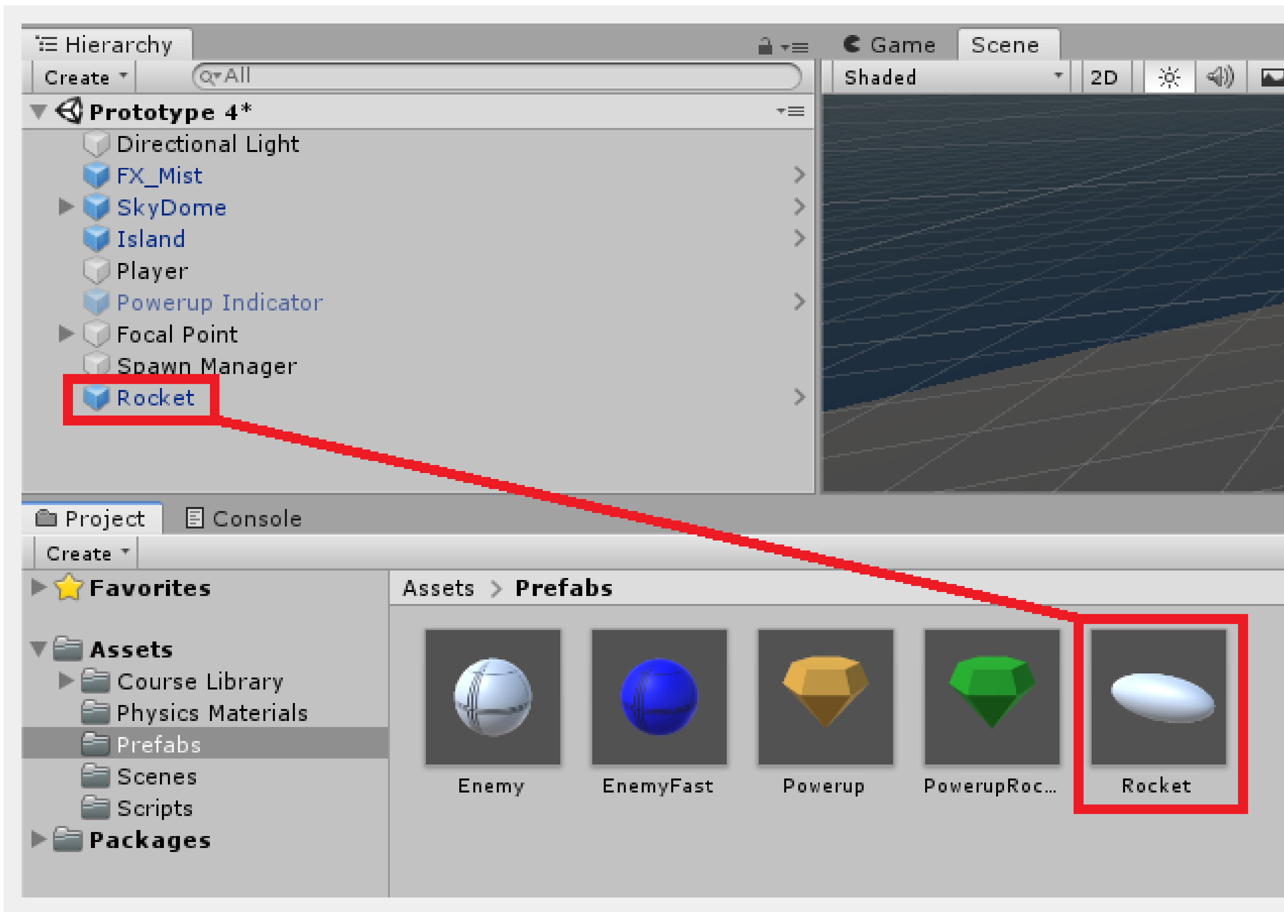
25. Add the **Rocket Behaviour** script to the Rocket, by selecting **Add Component** and searching for it.



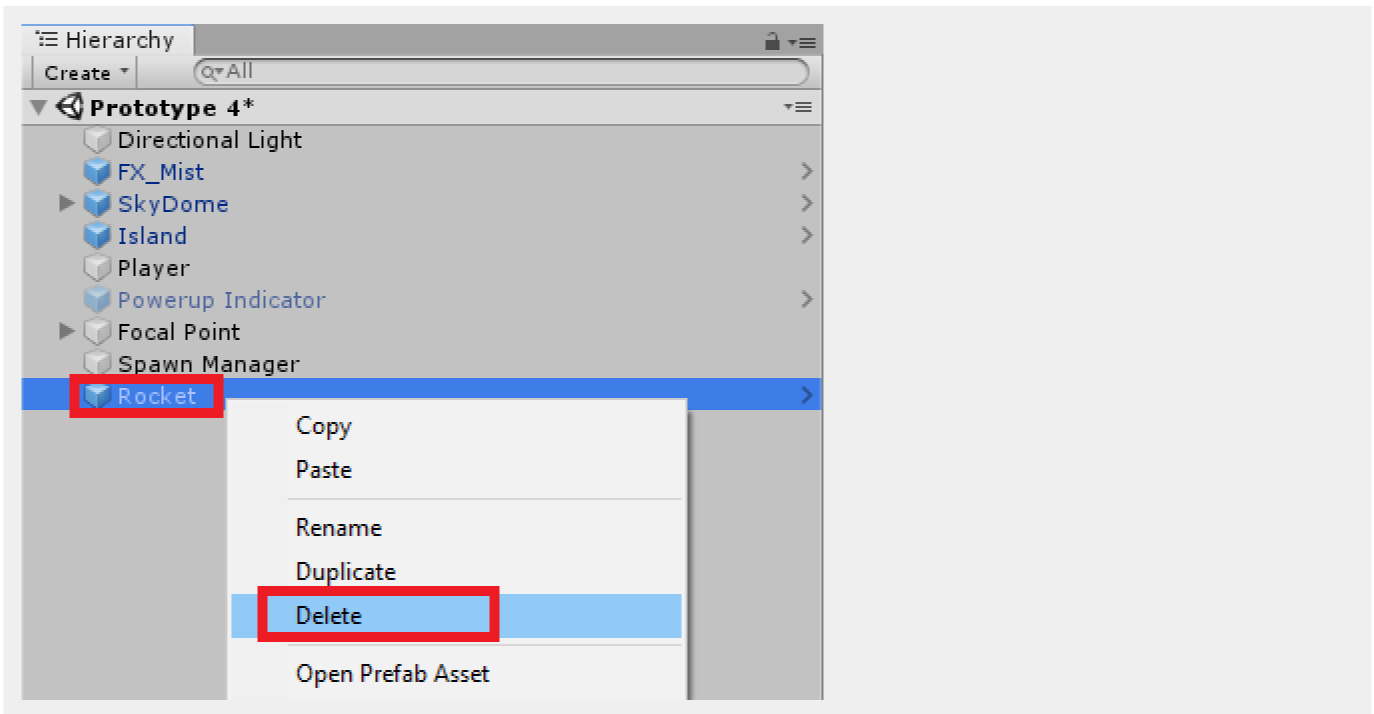
26. Adjust the scale of the Rocket to make it look more like a Rocket.



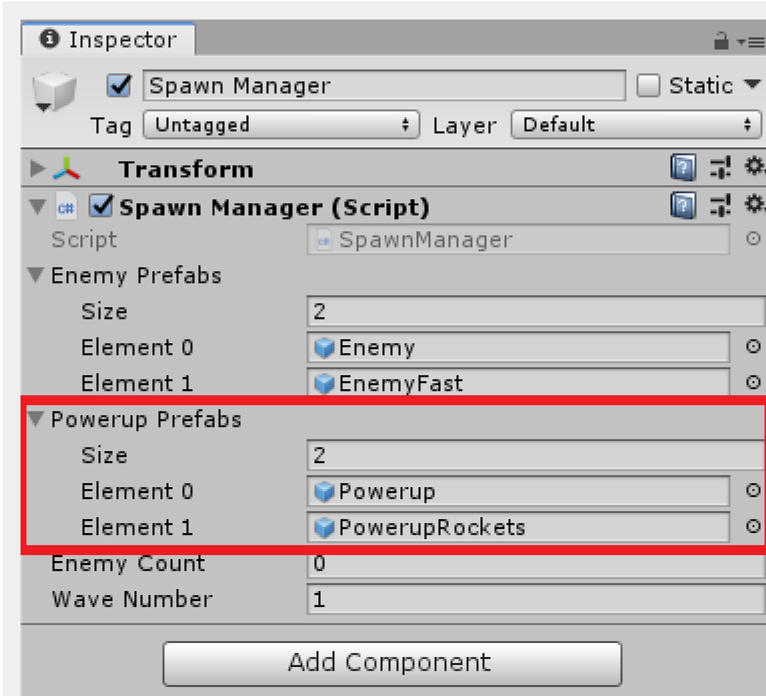
27. Drag the Rocket GameObject from the Hierarchy into the **Prefabs** folder. This will allow us to spawn in multiple Rockets at once.



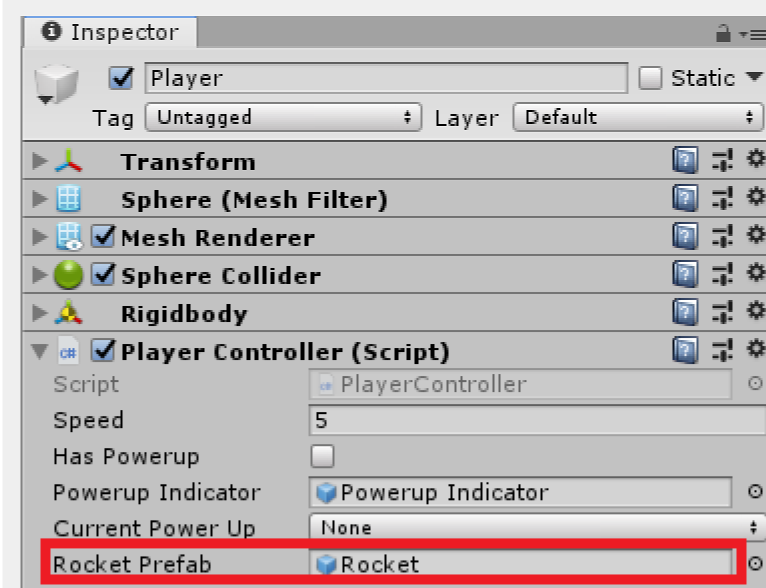
After creating the prefab, right-click the Rocket in the Hierarchy and select **Delete**



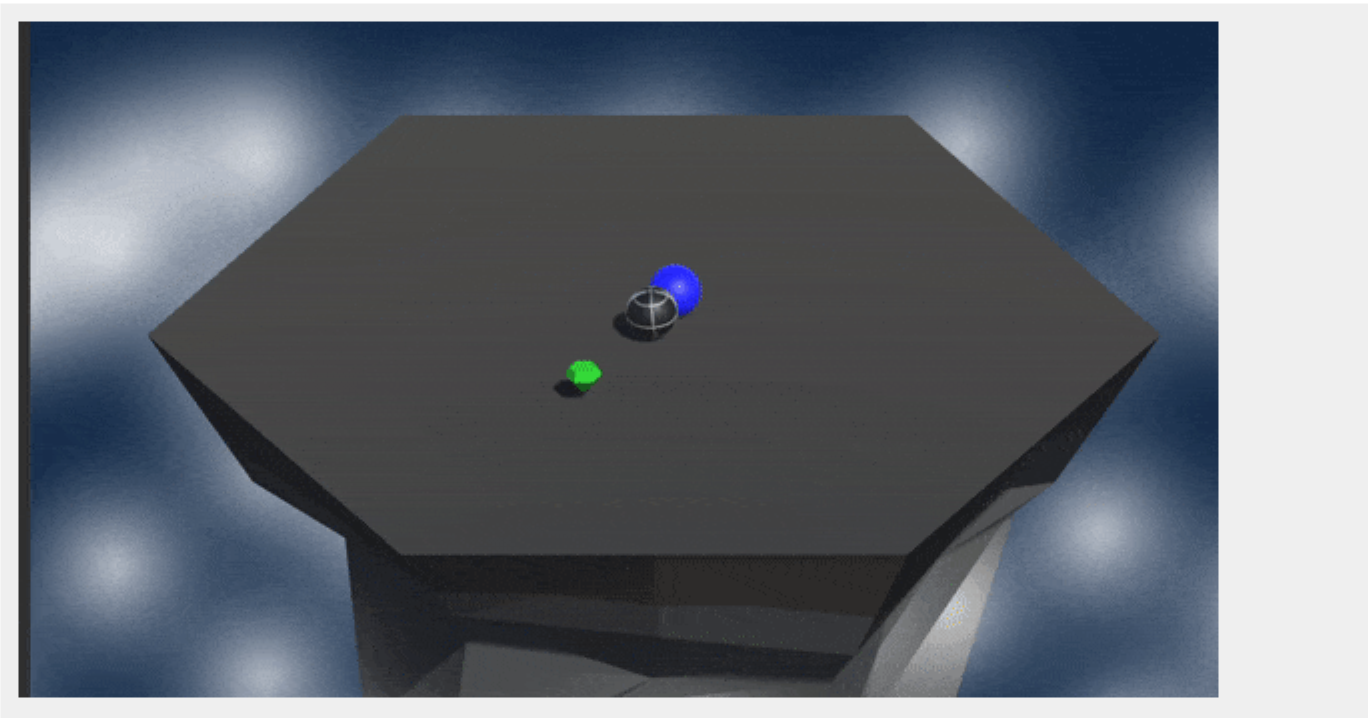
28. In the Hierarchy, select the *Spawn Manager* GameObject. In the Inspector for the Spawn Manager, we now need to set up the Powerup Prefabs parameters. Change the size to **2** and assign the two Powerup prefabs to the fields.



29. In the Hierarchy, select the Player GameObject. We will now set up the Rocket prefab. In the Inspector for the **Player Controller (Script)** component, assign the Rocket from the Prefabs folder to the **Rocket Prefab** Field.



30. Save the scene and press play. Try to pick up the new powerup and then press F. Notice how rockets will spawn and start moving towards all the enemies.



Hard - Smashingly Good

1. Navigate to the Scripts folder and open up the script **Powerup.cs**. Update the PowerUpType enum to look like the following:

```
public enum PowerUpType { None, Pushback, Rockets, Smash }
```

   Save the script and head back to Unity.

2. Open up the **PlayerController.cs** script. Before the **Start** method, add in the following new variables

```
public float hangTime;
public float smashSpeed;
public float explosionForce;
public float explosionRadius;

bool smashing = false;
float floorY;
```

3. Next, let's add in the Smash method. This method will be a coroutine so that we can wait while in the method. We will then launch the player in the air, and then smack them into the ground. When the hit the ground, they will do a shockwave like force to all nearby enemies.

```
IEnumerator Smash()
{
    var enemies = FindObjectsOfType<Enemy>();

    //Store the y position before taking off
    floorY = transform.position.y;

    //Calculate the amount of time we will go up
    float jumpTime = Time.time + hangTime;

    while(Time.time < jumpTime)
    {
        //move the player up while still keeping their x velocity.
        playerRb.velocity = new Vector2(playerRb.velocity.x, smashSpeed);
        yield return null;
    }

    //Now move the player down
    while(transform.position.y > floorY)
    {
        playerRb.velocity = new Vector2(playerRb.velocity.x, -smashSpeed * 2);
        yield return null;
    }
```

```
    //Cycle through all enemies.
    for (int i = 0; i < enemies.Length; i++)
    {
        //Apply an explosion force that originates from our position.
        if(enemies[i] != null)
            enemies[i].GetComponent<Rigidbody>().AddExplosionForce(explosionForce,
transform.position, explosionRadius, 0.0f, ForceMode.Impulse);
    }

    //We are no longer smashing, so set the boolean to false
    smashing = false;
}
```

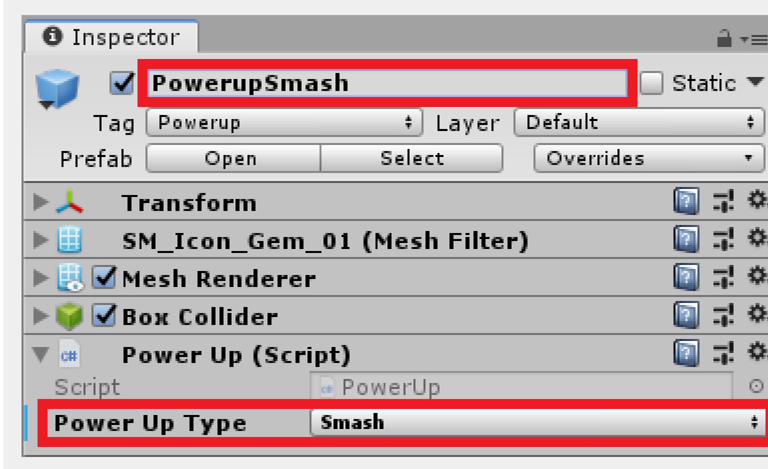
4. The final thing we need to do is add an if statement to the **Update** method. This if statement will check; is the current powerup the smash type, is the space bar pressed, and are we currently not smashing. If all are true, we'll set smashing to true and call out Smash method. The updated Update method looks like this:

```
void Update()
{
    float forwardInput = Input.GetAxis("Vertical");

    playerRb.AddForce(focalPoint.transform.forward * forwardInput * speed);

    powerupIndicator.transform.position = transform.position + new Vector3(0,
-0.5f, 0);

    if(currentPowerUp == PowerUpType.Rockets && Input.GetKeyDown(KeyCode.F))
    {
        LaunchRockets();
    }

    if(currentPowerUp == PowerUpType.Smash && Input.GetKeyDown(KeyCode.Space) &&
!smashing)
    {
        smashing = true;
        StartCoroutine(Smash());
    }
}
```

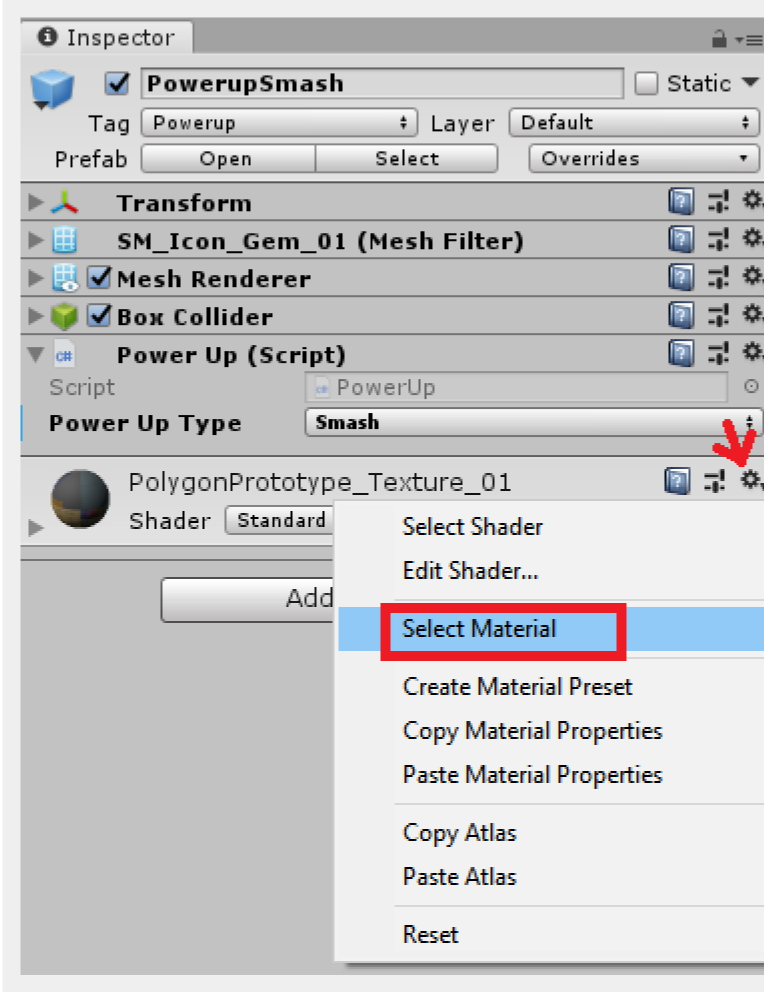
Save the script and head back to Unity.

5. We now need to create the new prefab for the smash powerup. Navigate to the Prefabs folder and drag the Powerup prefab into the Hierarchy.

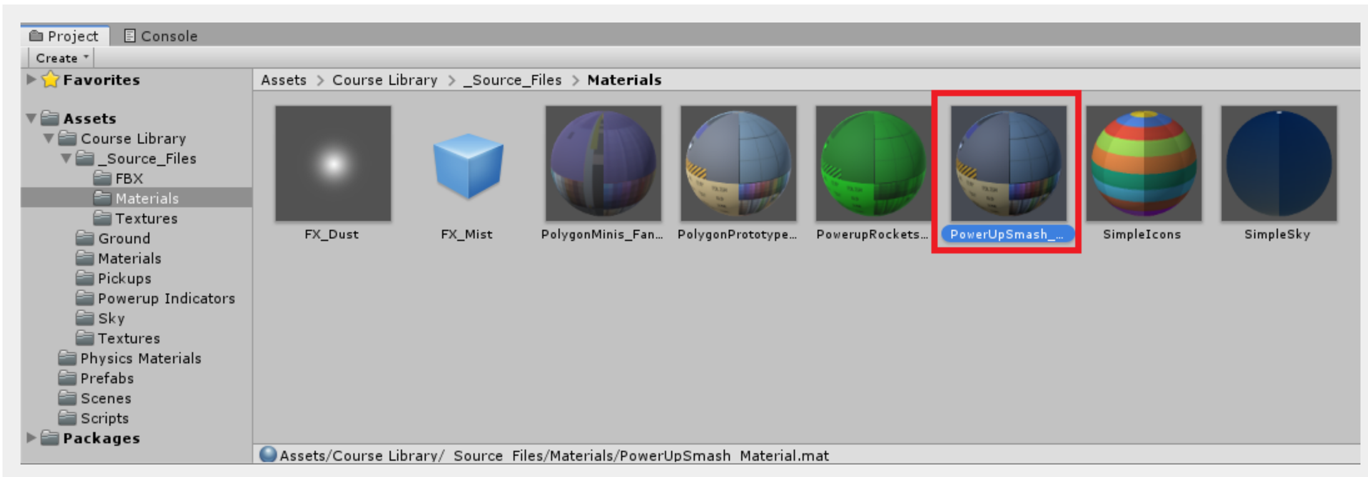
6. In the Hierarchy, select the Powerup GameObject. In its Inspector, change the name to *PowerupSmash* and change the **Power Up Type** to **Smash**.



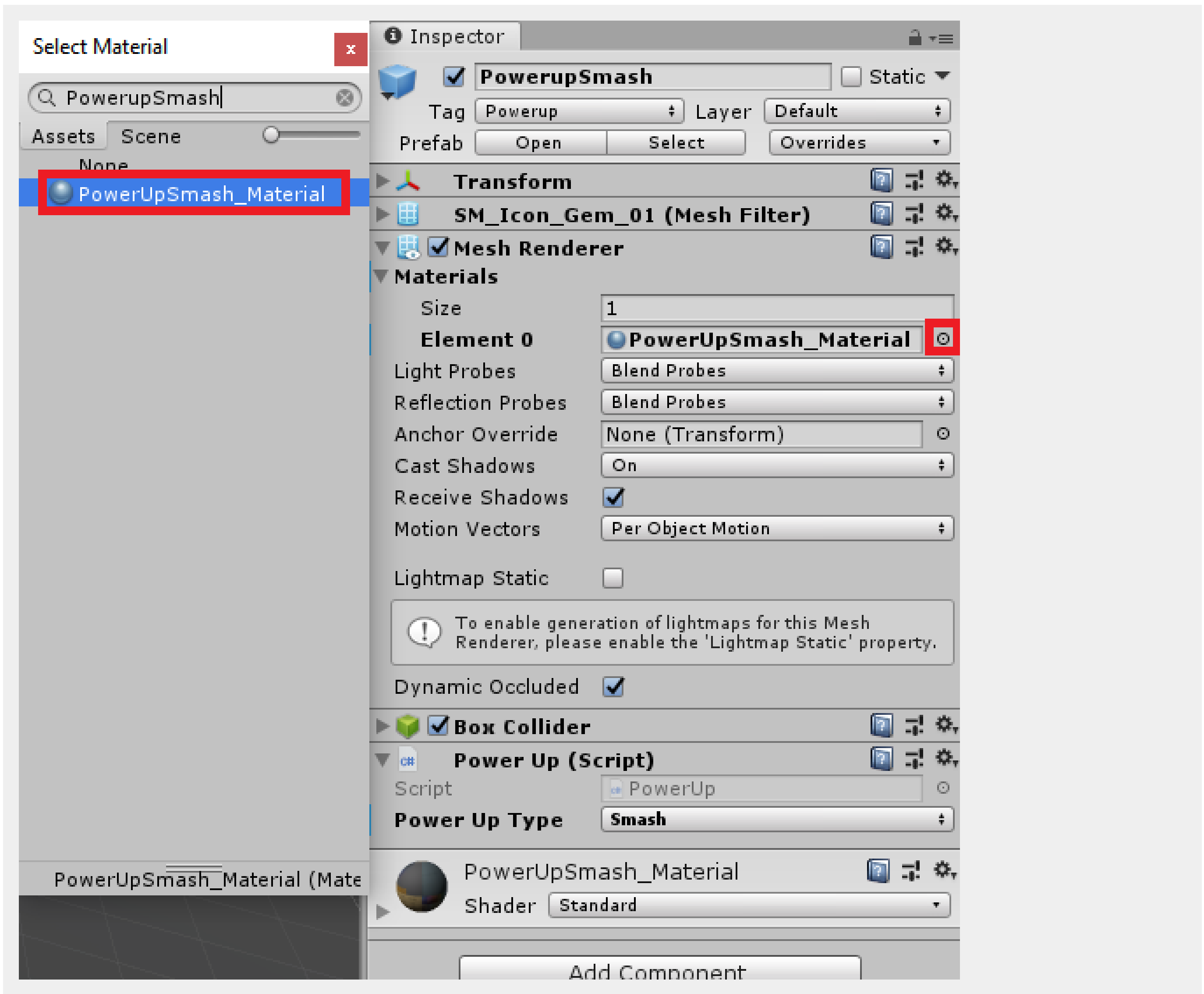
7. In the Inspector, select the Gear icon on the Material of the GameObject and click **Select Material**. This will show you where the material is located in the project.



8. Duplicate the Material by either, selecting the Material and pressing **Ctrl + D** or **Cmd + D**, or going to **Edit** > **Duplicate**. Name the new material *PowerUpSmash_Material*.

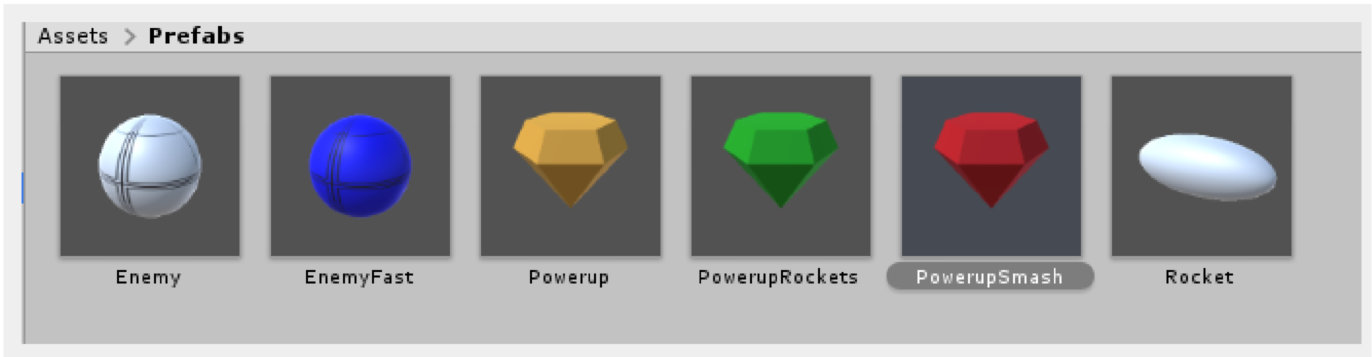


9. Select the PowerupSmash GameObject in the Hierarchy. On the Mesh Renderer component, expand the Materials section and click the selector next to Element 0. Type in *PowerUpSmash* and select the Material we just created.

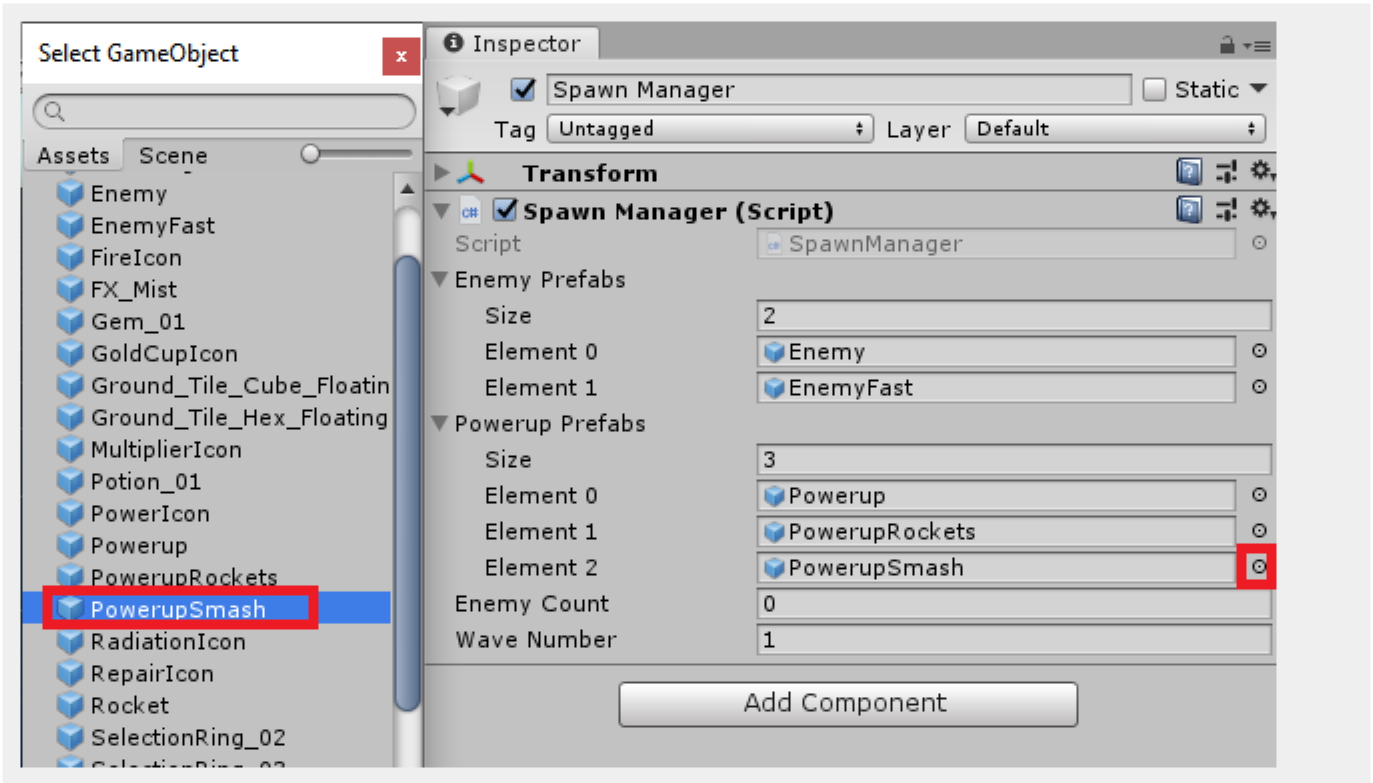


10. Expand the Material at the bottom of the Inspector and change the albedo value. We put ours as red.

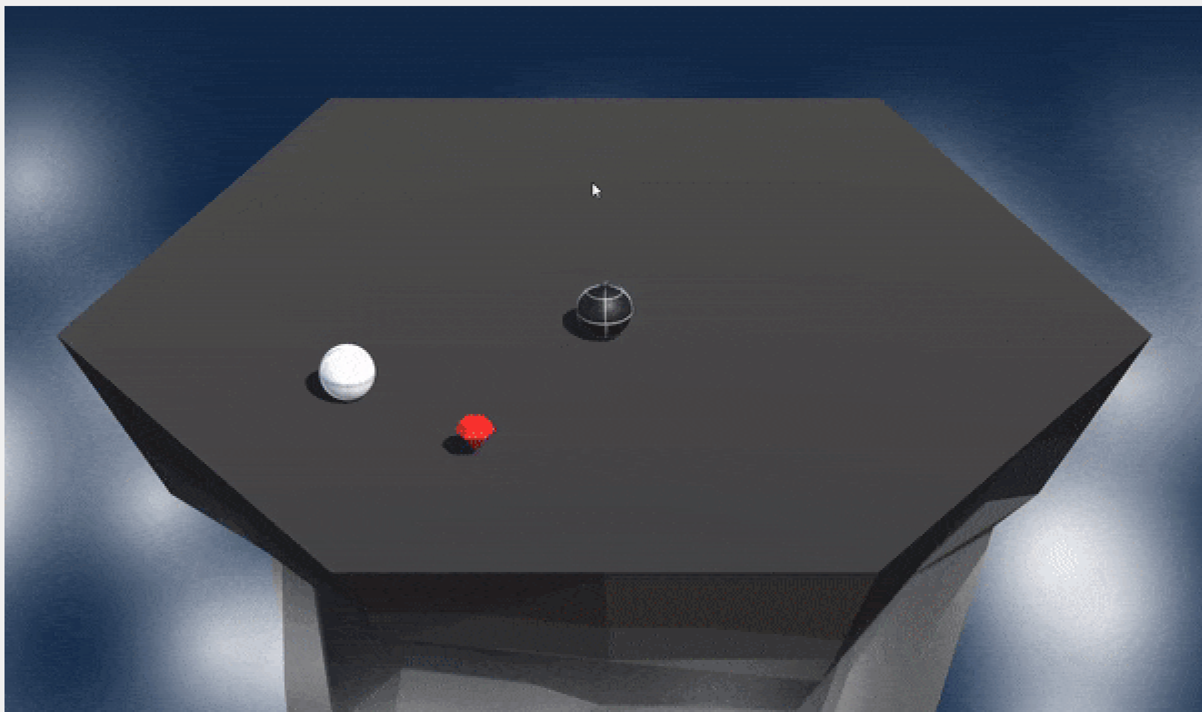
11. Navigate to the Prefabs folder, and drag the PowerupSmash GameObject from the Hierarchy into the Project view. When the Create Prefab pop-up appears, click **Original Prefab**. After you have turned it into a prefab, delete the PowerupSmash GameObject from the Hierarchy.



12. In the Hierarchy, select the SpawnManager GameObject. In the Inspector, on the Spawn Manager component, change the Powerup Prefabs size to 3. Click the selector button and select the PowerupSmash Prefab.



13. Save the scene and press play. Pick-up a red powerup and press space to use the smash attack



14.

15.

16.

Expert - Boss Battle

After a certain number of waves, program a mini “boss battle,” where the boss has some completely new abilities. For example, maybe the boss can fire projectiles at you, maybe it is extremely agile, or maybe it occasionally generates little minions that come after you.

1. Navigate to the Scripts folder and open up the **Enemy** script. Below the current variable declarations, add the following:

```
public bool isBoss = false;

public float spawnInterval;
private float nextSpawn;

public int miniEnemySpawnCount;

private SpawnManager spawnManager;
```

2. Inside the **Start** method, let’s check if the enemy is a boss. If it is, we will set up the spawnManager variable. The update Start method should look like this:

```
void Start()
{
    enemyRb = GetComponent<Rigidbody>();
    player = GameObject.Find("Player");

    if (isBoss)
    {
        spawnManager = FindObjectOfType<SpawnManager>();
    }
}
```

3. Next let’s update the **Update** method to call a method we’ll make shortly on the SpawnManager. The updated method should look like this:

```
void Update()
{
    Vector3 lookDirection = (player.transform.position - 
transform.position).normalized;

    enemyRb.AddForce(lookDirection * speed);

    if(isBoss)
    {
        if(Time.time > nextSpawn)
        {
            nextSpawn = Time.time + spawnInterval;
```

```
            spawnManager.SpawnMiniEnemy(miniEnemySpawnCount);
        }
    }

    if(transform.position.y < -10)
    {
        Destroy(gameObject);
    }
}
```

4. Save the script and open up the **SpawnManager** script. Below the current variable decelerations, add the following:

```
public GameObject bossPrefab;
public GameObject[] miniEnemyPrefabs;
public int bossRound;
```

5. Below the **GenerateSpawnPosition** method, we will need to create two new methods. The first will handle spawning the boss. The second will handle spawning the mini enemies. Lets start with the boss method. Add the following:

```
void SpawnBossWave(int currentRound)
{
    int miniEnemysToSpawn;
    //We dont want to divide by 0!
    if (bossRound != 0)
    {
        miniEnemysToSpawn = currentRound / bossRound;
    }
    else
    {
        miniEnemysToSpawn = 1;
    }

    var boss = Instantiate(bossPrefab, GenerateSpawnPosition(),
bossPrefab.transform.rotation);
    boss.GetComponent<Enemy>().miniEnemySpawnCount = miniEnemysToSpawn;
}
```

Let's break this down. First we are dividing the current round by the specified boss round and then storing it into a variable for future use. Then we spawn in the boss and set the amount of mini enemies they will spawn to the variable we just created.

6. Next let's create the mini enemy spawning method. Below the **SpawnBossWave** method, add the following:

```
public void SpawnMiniEnemy(int amount)
{
```

```
    for(int i = 0; i < amount; i++)
    {
        int randomMini = Random.Range(0, miniEnemyPrefabs.Length);
        Instantiate(miniEnemyPrefabs[randomMini], GenerateSpawnPosition(),
miniEnemyPrefabs[randomMini].transform.rotation);
    }
}
```

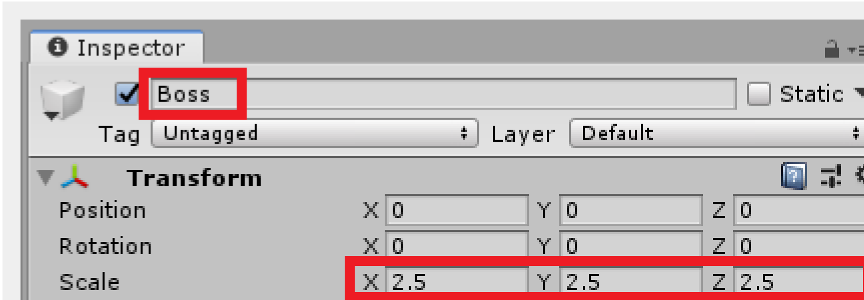
7. The last thing we need to do in the **SpawnManager** script is to adjust the Update method. The updated **Update** method should look like this:

```
void Update()
{
    enemyCount = FindObjectsOfType<Enemy>().Length;

    if(enemyCount == 0)
    {
        waveNumber++;
        //Spawn a boss every x number of waves
        if (waveNumber % bossRound == 0)
        {
            SpawnBossWave(waveNumber);
        }
        else
        {
            SpawnEnemyWave(waveNumber);
        }

        //Updated to select a random powerup prefab for the Medium Challenge
        int randomPowerup = Random.Range(0, powerupPrefabs.Length);
        Instantiate(powerupPrefabs[randomPowerup], GenerateSpawnPosition(),
powerupPrefabs[randomPowerup].transform.rotation);
    }
}
```

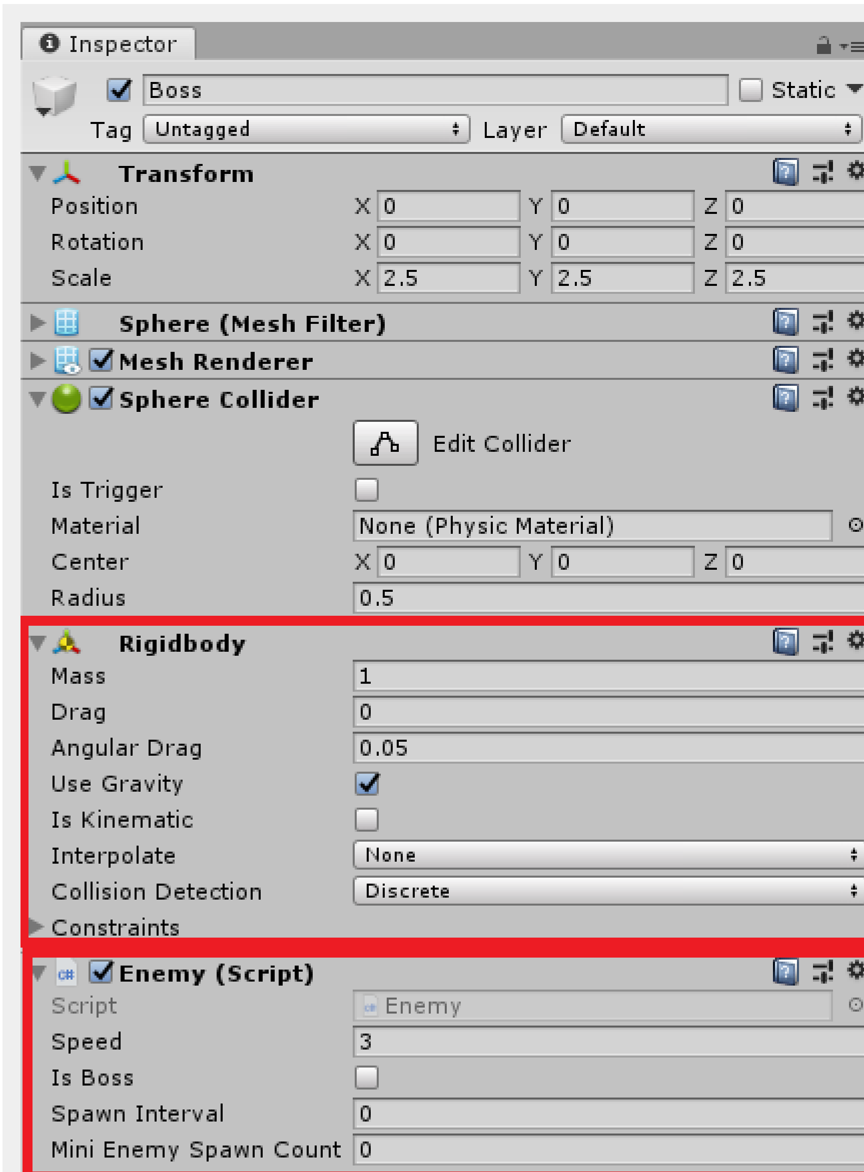
The code we added, uses a modulus operator to check if we are currently on a boss round. For example, if bossRound is set to 3 and the waveNumber was set to 6 the modulus operator would return 0 as the can be divided without a remainder.

Save the script and return to Unity.

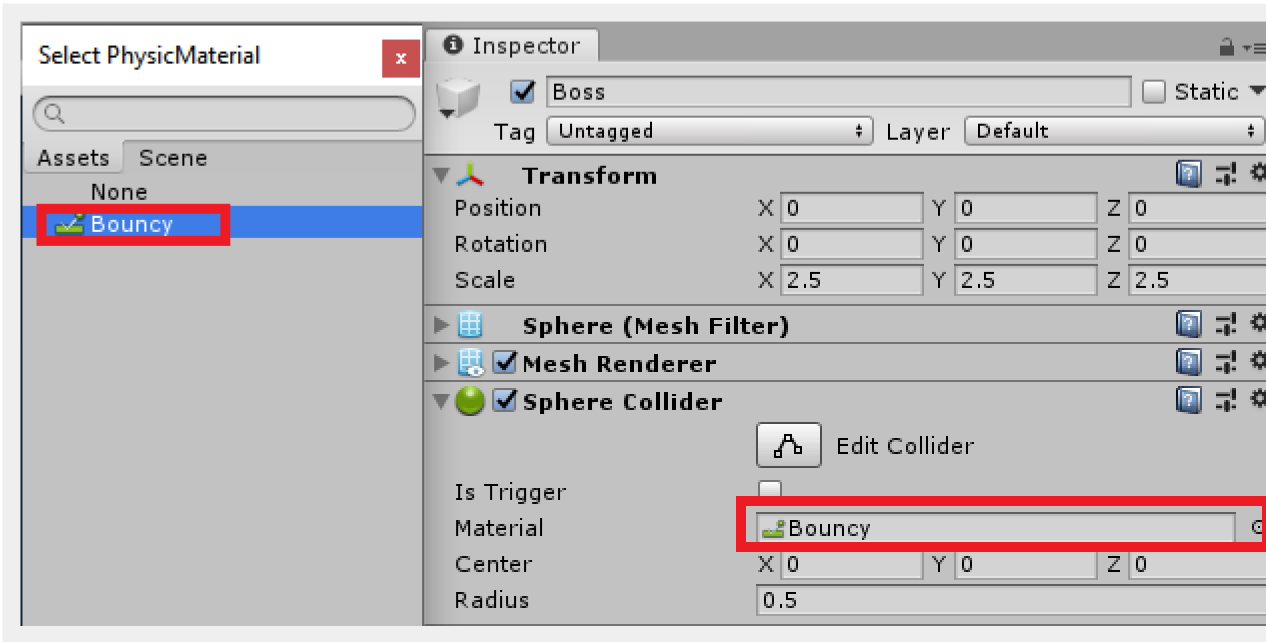
8. Back in the editor, we will need to create the new prefabs for the boss and mini enemies. Let's start with the boss. Right-click in the Hierarchy and select **3D** > **Sphere**. Rename the Sphere to Boss and adjust the scale to 2.5 on all axes.



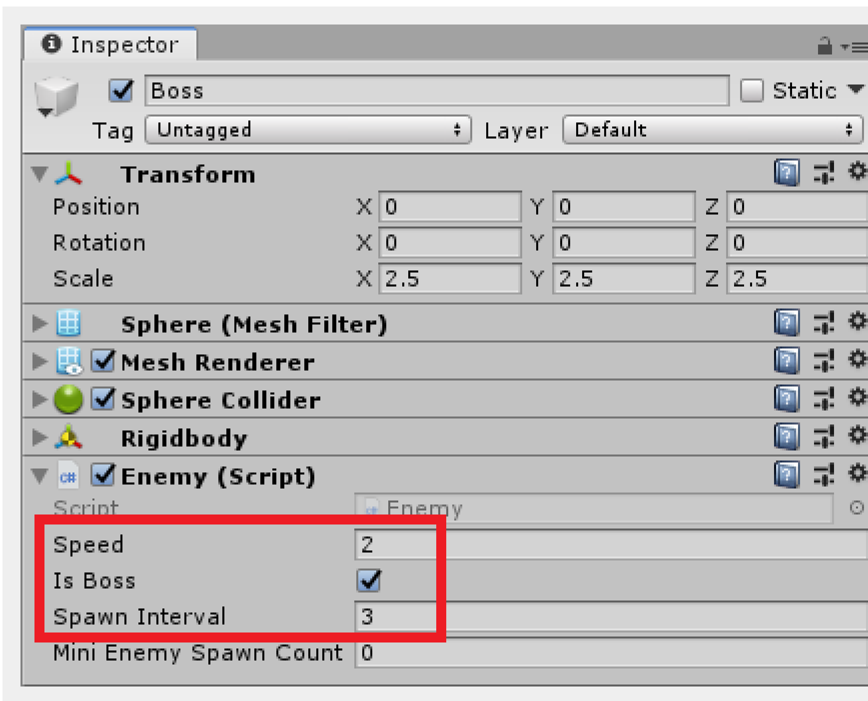
9. Next, add two components to the Boss GameObject. The first is a Rigidbody and the second is the Enemy script.



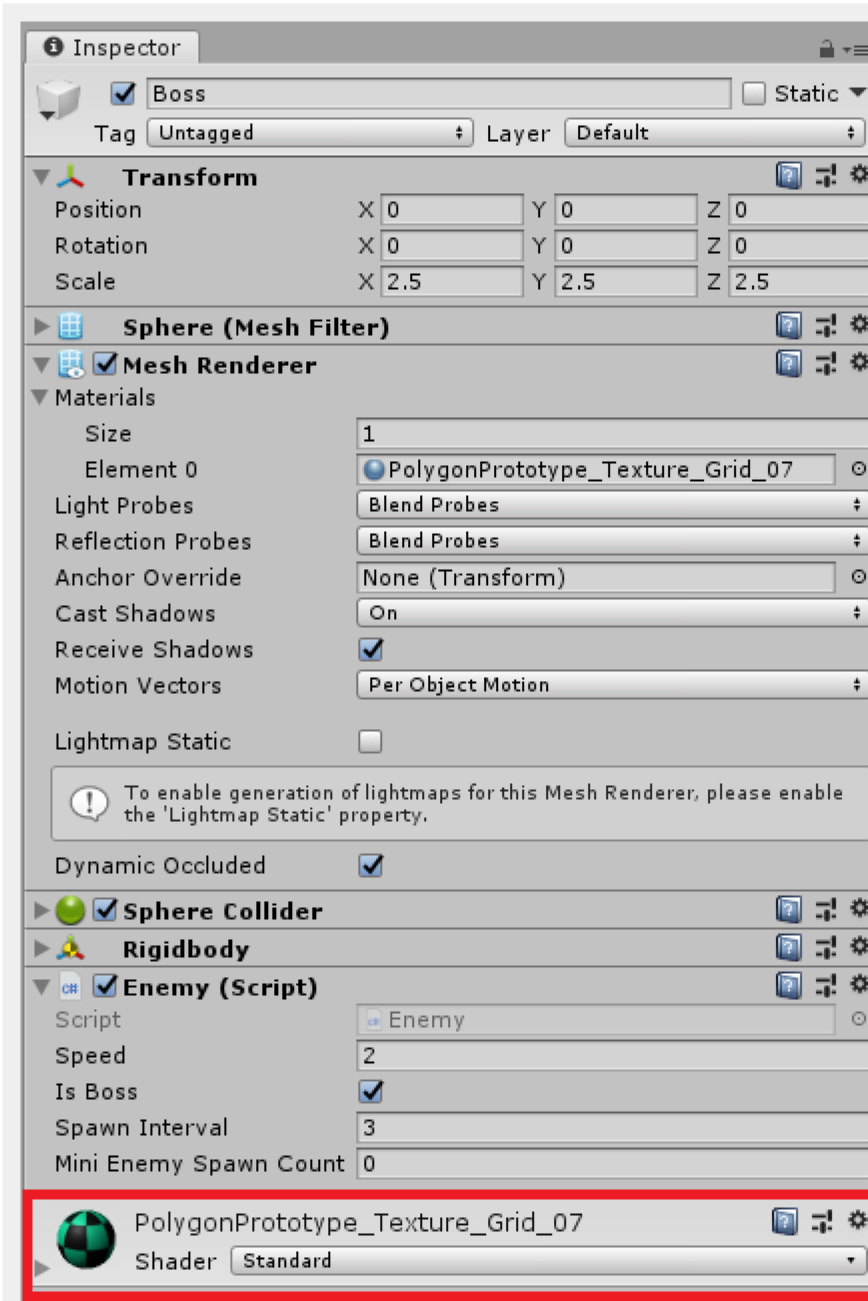
10. On the **Sphere Collider** click the selector icon to the right of the Material field. Look for the Bouncy Material and select it. This will ensure the boss has the same behaviour as the enemies when colliding with GameObjects.



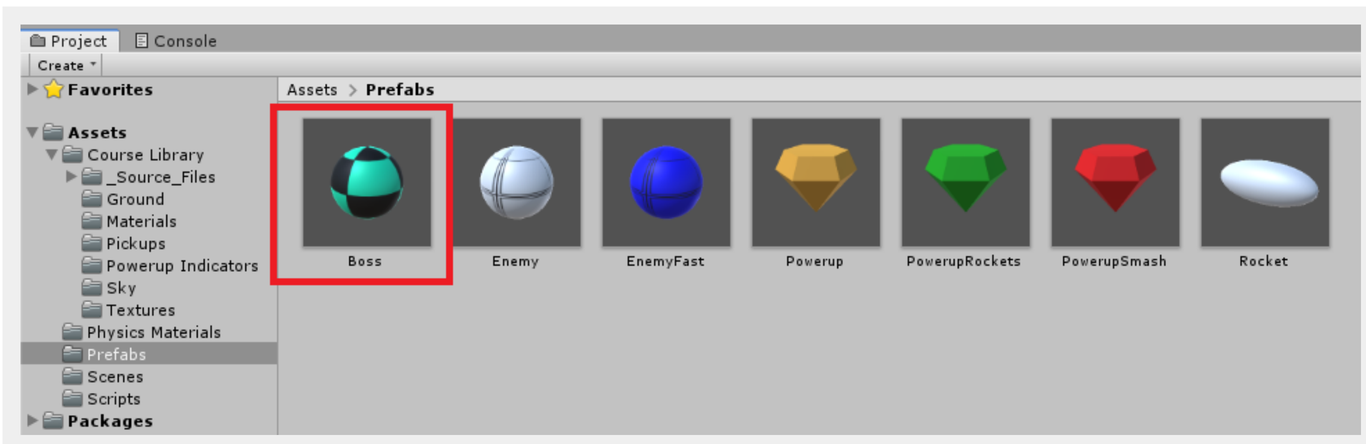
11. On the **Enemy** component, we need to set up some values. Change the **Speed** value to **2**, check the **Is Boss** checkbox, and Change Spawn Interval to **3**.



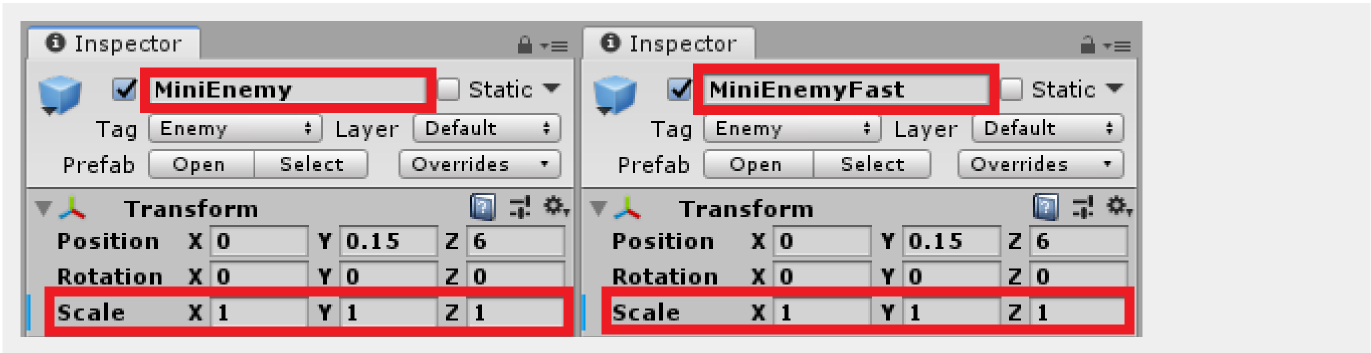
12. On the **Mesh Renderer** component, we will set a material. Navigate to the folder *Course Library > Materials***.** We selected *PolygonPrototype_Texture_Grid_07* and adjusted the albedo value to give a turquoise color. Drag this onto the Boss GameObject.



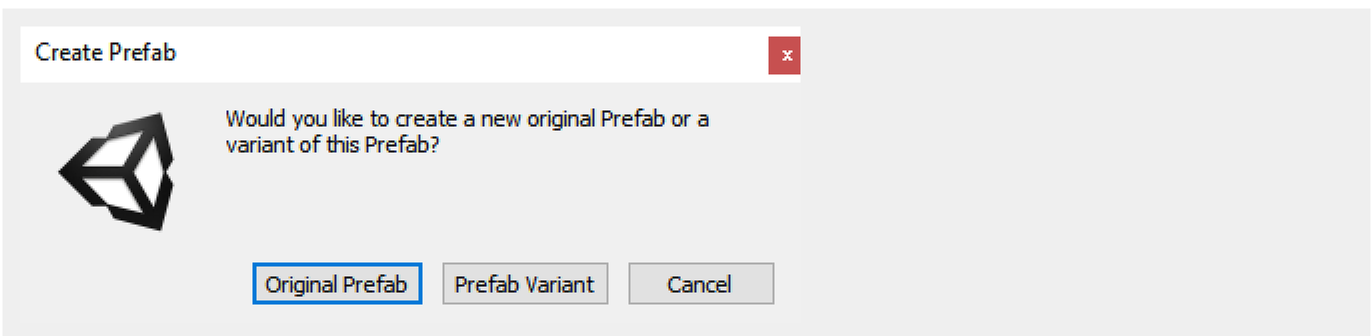
13. Drag the Boss GameObject from the Hierarchy into the Prefabs folder to turn it into a prefab. After turning it into a prefab, delete it from the Hierarchy.



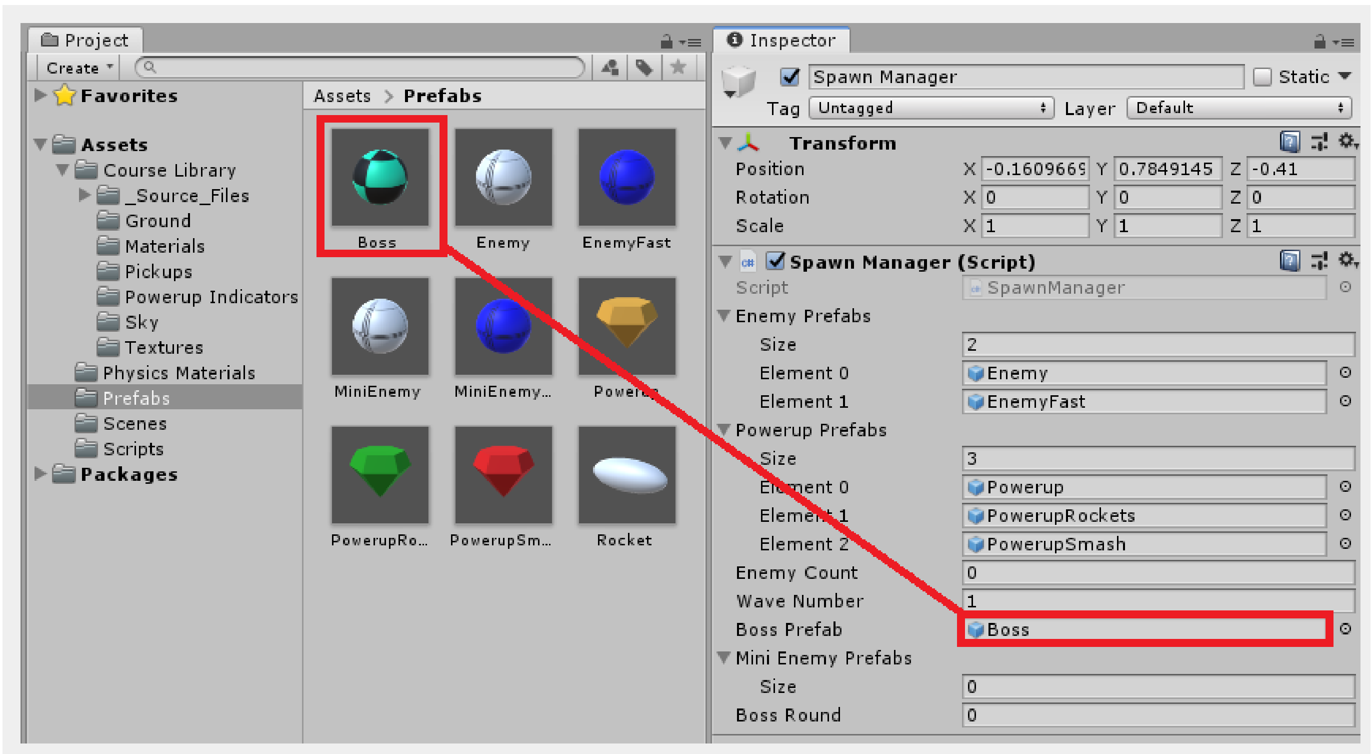
14. Drag the Enemy and EnemyFast prefabs into the Hierarchy. We will use these as a base for our mini enemies. Select them both in the Hierarchy and adjust the scale to 1. Add the prefix of *Mini* to both GameObjects names.



15. Drag the GameObjects from the Hierarchy to the Prefabs folder. When prompted with a window, select Original Prefab. After creating the prefabs, delete the GameObjects from the Hierarchy.



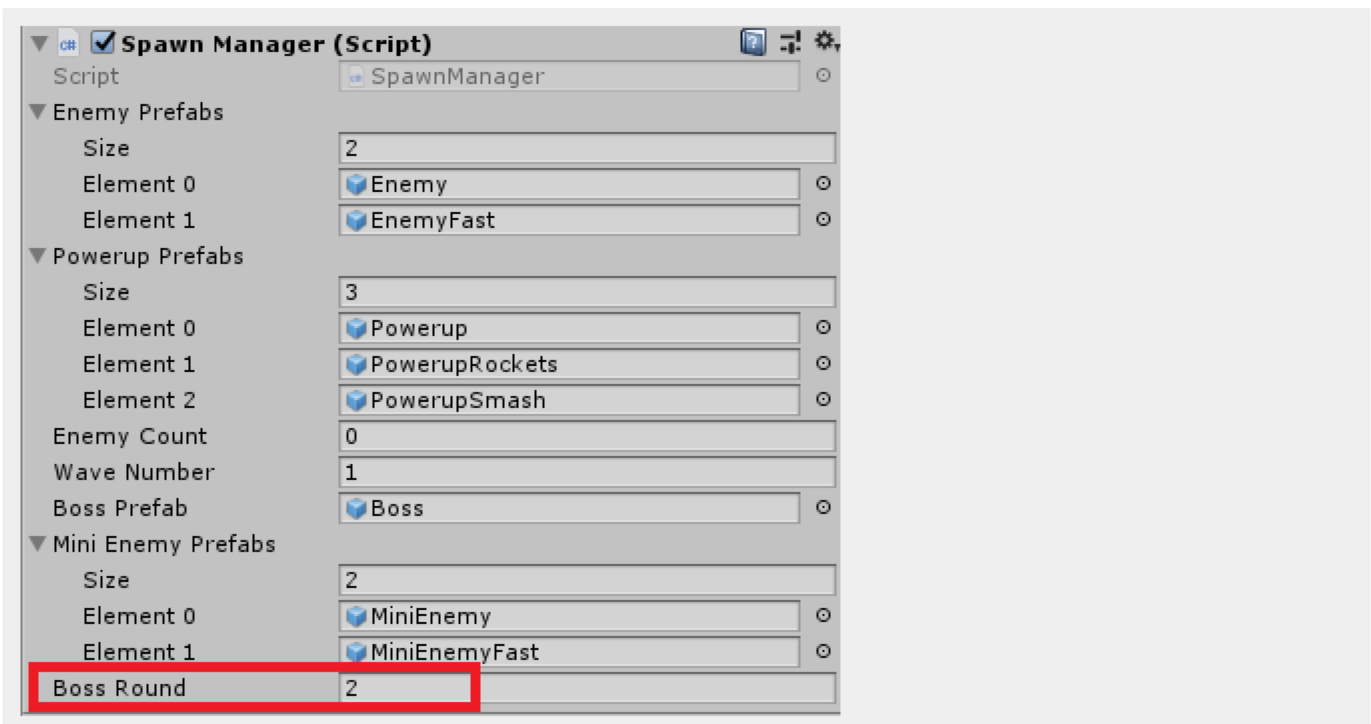
16. The final thing we need to do is to set up the **Spawn Manager** variables. First drag the Boss prefab from the Project view into the **Boss Prefab** field.



17. Next, drag the two MiniEnemy's into the **Mini Enemy Prefabs** field.

Inspector
Assets > Prefabs
Boss
Enemy
EnemyFast
MiniEnemy
MiniEnemy...
Powerup
PowerupRo...
PowerupSm...
Rocket
Spawn Manager
Static
Tag Untagged
Layer Default
Transform
Position X -0.1609669 Y 0.7849145 Z -0.41
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Spawn Manager (Script)
Script SpawnManager
Enemy Prefabs
Size 2
Element 0 Enemy
Element 1 EnemyFast
Powerup Prefabs
Size 3
Element 0 Powerup
Element 1 PowerupRockets
Element 2 PowerupSmash
Enemy Count 0
Wave Number 1
Boss Prefab Boss
Mini Enemy Prefabs
Size 0
Boss Round 0
Add Component

18. The last thing we need to do is change the **Boss Round**. This will be used to determine which round will be the boss one. If you put 5, after 4 normal rounds a boss round would happen. To make it a bit quicker to show the boss, we put our **Boss Round** value to **2**.



19. Save the scene and press play. Every 2 rounds a boss should spawn. The boss will keep spawning mini enemies until you knock them off the platform.