Asteroids: Part 3 – Asteroids

Lesson Goals

Here are two useful videos that can show you how to use events in Unity.

https://www.youtube.com/watch?v=70PcP_uPuUc
https://www.youtube.com/watch?v=Z8zlm7kZn30

Lesson Goals

Add collisions to our player, bullets and asteroids.

Give the large asteroids the ability to spawn smaller asteroids.

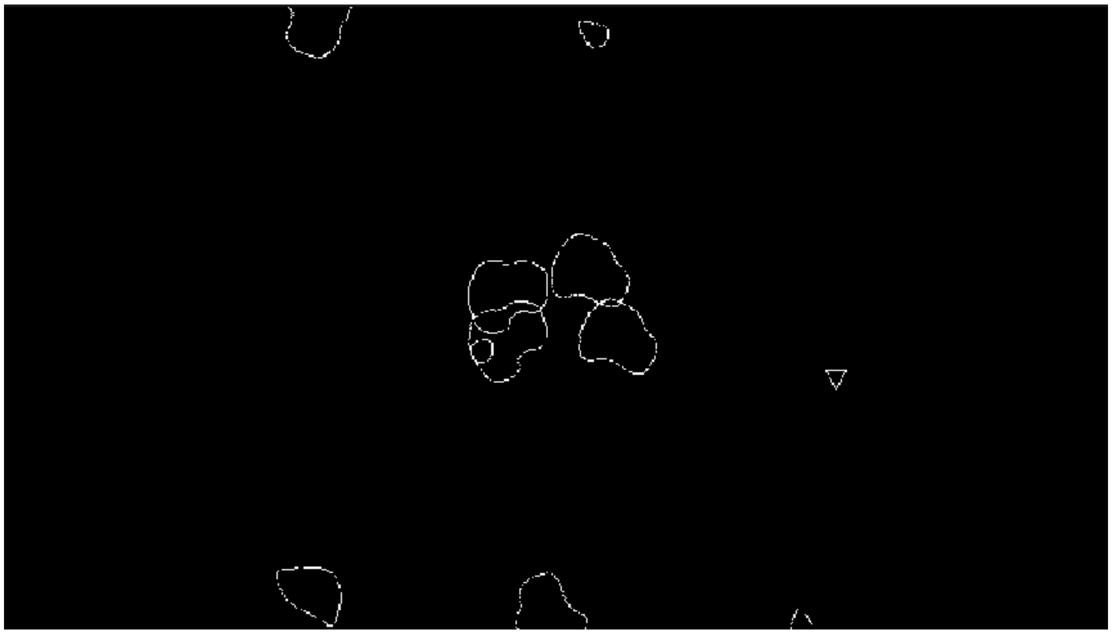
Use Unity's event system and coroutines to spawn in asteroids.

Lesson Intro

This is the final lesson for the asteroids game project and there is a lot of useful game dev know how pack into this lesson. Our primary goal is to finish our game by adding interactions between our game objects. At the end we will take a look at some advanced Unity features that will allow use to properly spawn in asteroids as waves for our game.

Demo

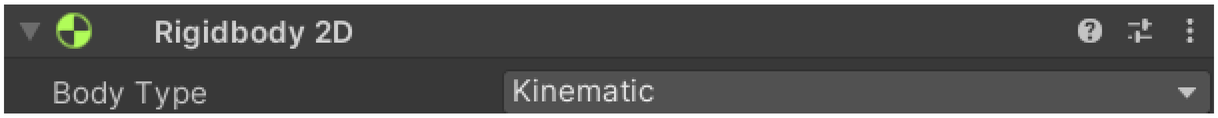
1. By the end of this lesson, we will have something the looks like:



Rigidbodies and Colliders

1. So up until now in our game, there hasn't been much of a threat of danger. The player is able to pass seamlessly through the asteroids. Seeing as how the asteroid is a giant rock hurtling through space this probably shouldn't be the case. So, let's fix this issue by adding collisions!

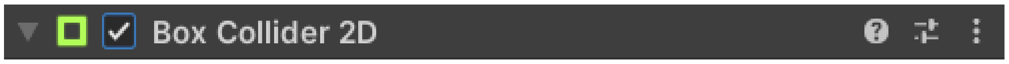
   Select the player object and add a **Rigidbody 2D** component. Set the **Body Type** to **Kinematic**.

   Rigidbody 2D

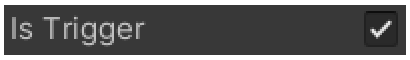
   Body Type Kinematic

   Select the player object and add in a **Box Collider 2D** component.

   Box Collider 2D

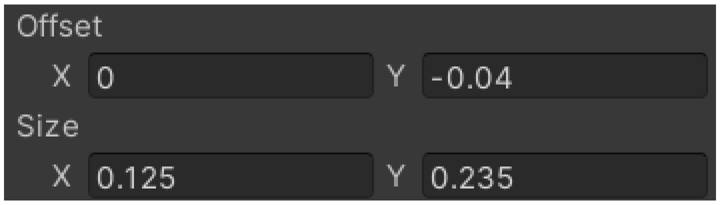
   Turn it into a trigger.

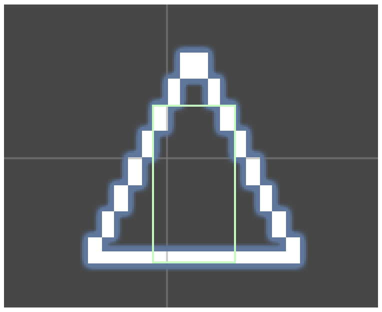
   Is Trigger ✓

   We are making the collider into a trigger because we want to report a collision without actually moving the object in the process.

   Then set the **Offset** and **Size** values to:


Offset
X 0 Y -0.04
Size
X 0.125 Y 0.235

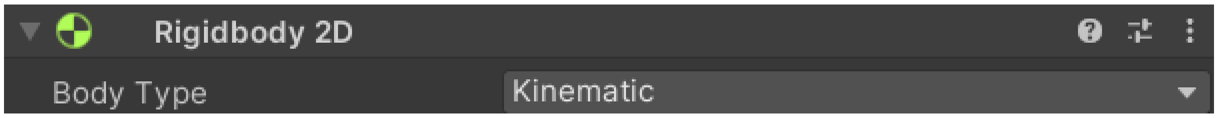
This will provide our ship with the collider.



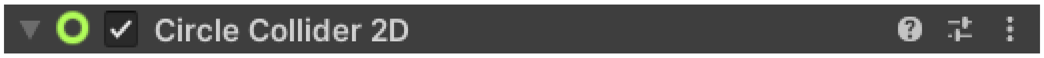
It is okay for the collider to not line up exactly with the ship. Typically, we want to make the hitboxes of our player characters a little smaller than what you would expect. This puts the odds of the game in the player’s favor by a small amount.

2. Let’s go ahead and repeat the process for the asteroid. **Be sure that you are editing the Asteroid prefab, that way any changes made to the prefab will affect all the asteroids in the scene.**

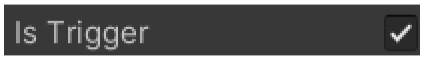
   Give it a **Rigidbody 2D** component.


Rigidbody 2D
Body Type Kinematic

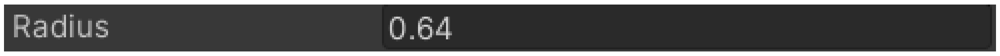
   However, we are going to be using a **Circle Collider 2D** instead of a box collider 2d.

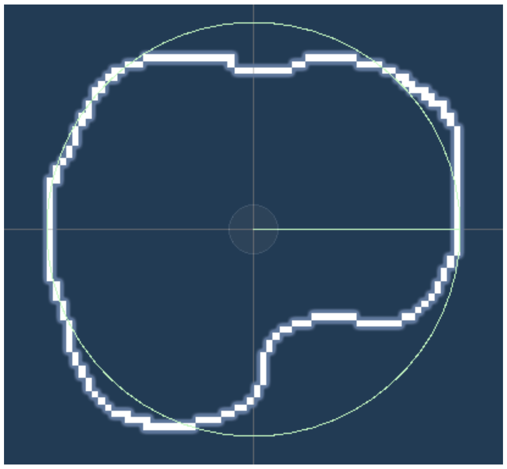

✓ Circle Collider 2D

   Turn it into a trigger.

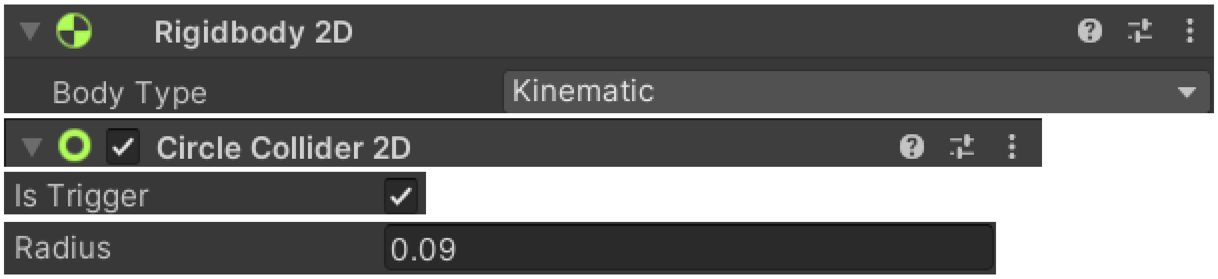

Is Trigger ✓

   The radius of the circle can be 0.64.


Radius 0.64



3. Let’s do this one more time except for the Bullet game object. Again, be sure to be in the bullet prefab scene.



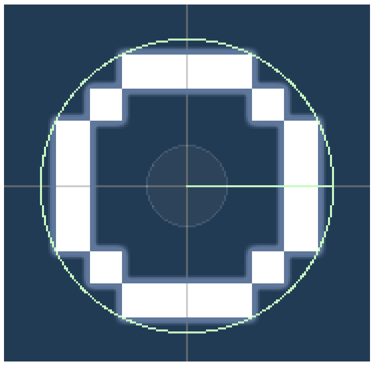
Rigidbody 2D

Body Type: Kinematic

Circle Collider 2D

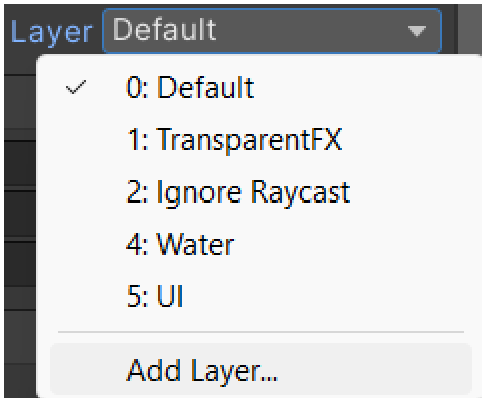
Is Trigger: ✓

Radius: 0.09

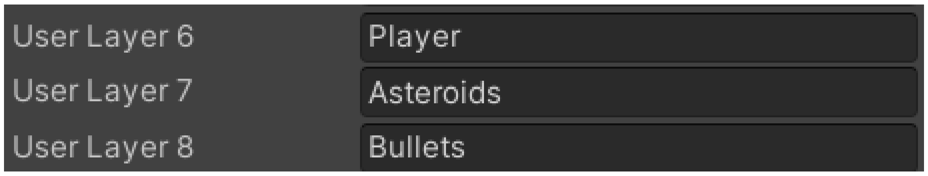


4. Now that each object has their own collider, lets then create, assign and then tell them what layers they should interact with.

   Starting out with the player, at the top-right of the inspector select the **Layer Dropdown** and then in the drop down select **Add Layer...**

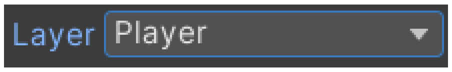


Add the collision layers



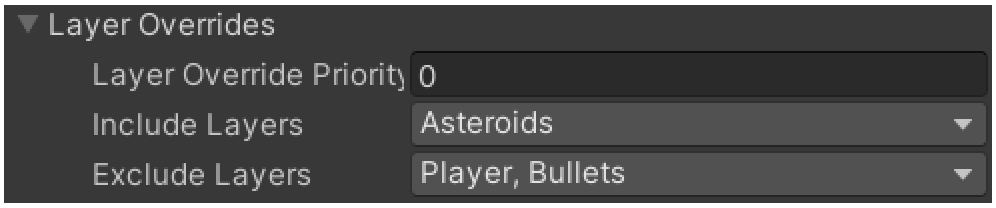
User Layer 6	Player
User Layer 7	Asteroids
User Layer 8	Bullets

Go back to the player's inspector and then assign the player to the **Player** layer. A **popup box** will appear. It is asking if the children of the object should be on the same layer that the parent will be on. For this instance, will click **Yes, change children**.



Layer: Player

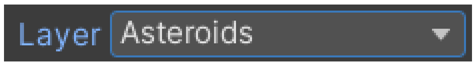
Next find the **Layer Overrides** on the **Box Collider 2D** component and set the **Include** and **Exclude Layers** to:



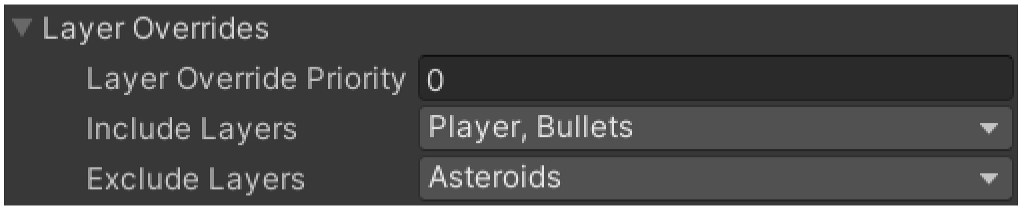
Layer Overrides

Layer Override Priority	0
Include Layers	Asteroids
Exclude Layers	Player, Bullets

5. We can go into the **Asteroid prefab scene** and repeat the process.



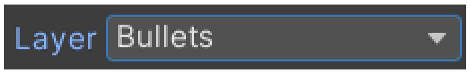
Layer: Asteroids



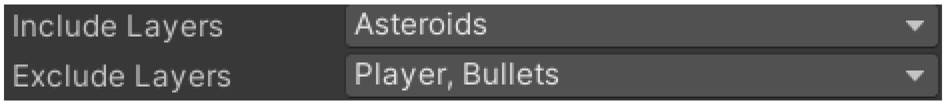
Layer Overrides

Layer Override Priority	0
Include Layers	Player, Bullets
Exclude Layers	Asteroids

6. Lastly, we repeat the process except with the **Bullet prefab scene**.



Layer: Bullets

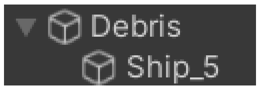


Include Layers	Asteroids
Exclude Layers	Player, Bullets

Player Collisions with Asteroids

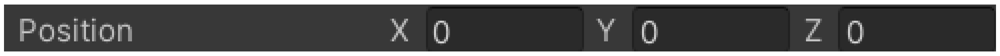
1. Let's make the player explode.

   Create a new **Empty** game object and rename it to **Debris**. As a child object add in the last ship sprite from the ship sprite sheet.

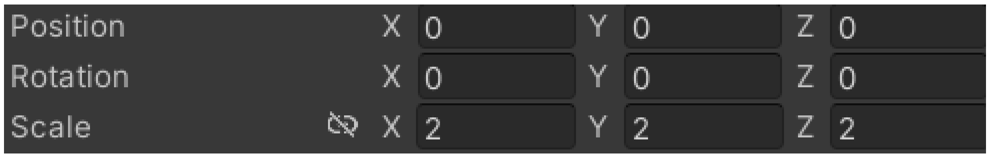
   ```
   ▼ Debris
       Ship_5
   ```

2. Zero out the position of the Debris object.

Position	X	0	Y	0	Z	0

   Then zero out the position of the sprite object. Also set the scale of the sprite to 2 for the X, Y and Z.

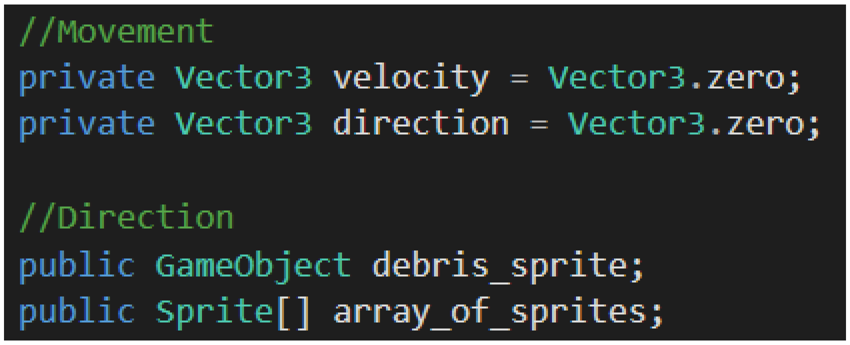
   

Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	2	Y	2	Z	2

3. Create a new script called **Debris** and attach it to the Debris game object.

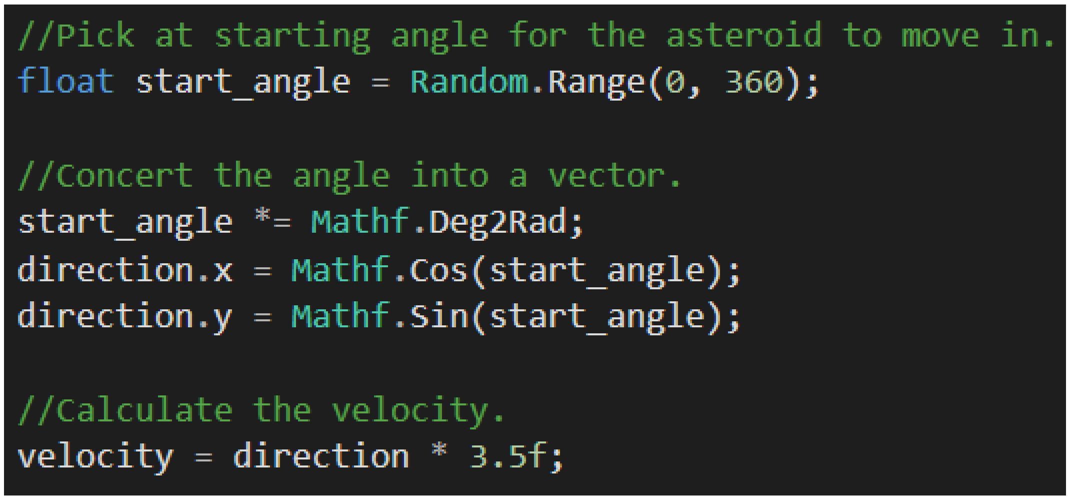
4. Turn the Debris game object into a prefab and then delete the object from the scene.

5. Open the **Debris** script. We are going to be stealing a lot of code from the asteroid script so feel free to copy and paste the code. Just be sure to make the changes in the code below. Let's first add the variables:



```
//Movement
private Vector3 velocity = Vector3.zero;
private Vector3 direction = Vector3.zero;

//Direction
public GameObject debris_sprite;
public Sprite[] array_of_sprites;
```

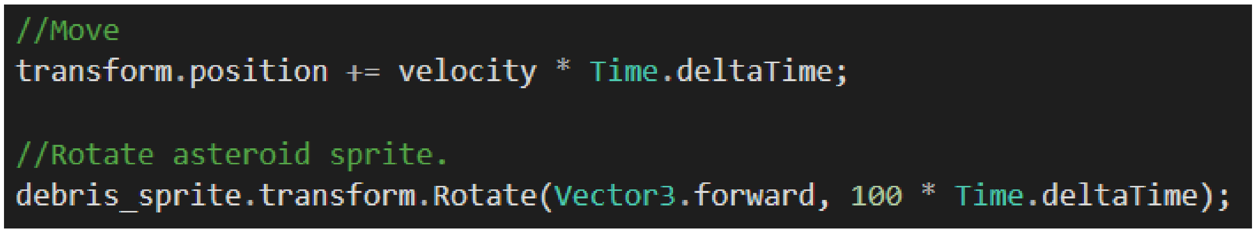
6. Then in the **Start()** function lets add the code:



```
//Pick at starting angle for the asteroid to move in.
float start_angle = Random.Range(0, 360);

//Concert the angle into a vector.
start_angle *= Mathf.Deg2Rad;
direction.x = Mathf.Cos(start_angle);
direction.y = Mathf.Sin(start_angle);

//Calculate the velocity.
velocity = direction * 3.5f;
```

We do not have a max_speed variable so we are hard coding the speed to be at 3.5.

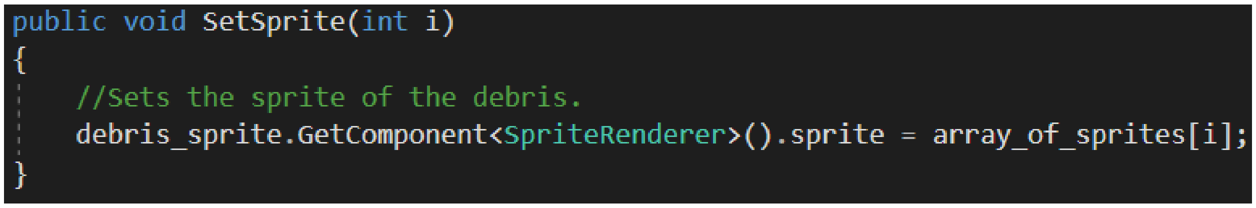
7. Then in the **Update()** function we add the code:



```
//Move
transform.position += velocity * Time.deltaTime;

//Rotate asteroid sprite.
debris_sprite.transform.Rotate(Vector3.forward, 100 * Time.deltaTime);
```

We do not have a spin_speed variable so we hard code the rotation speed to be 100.

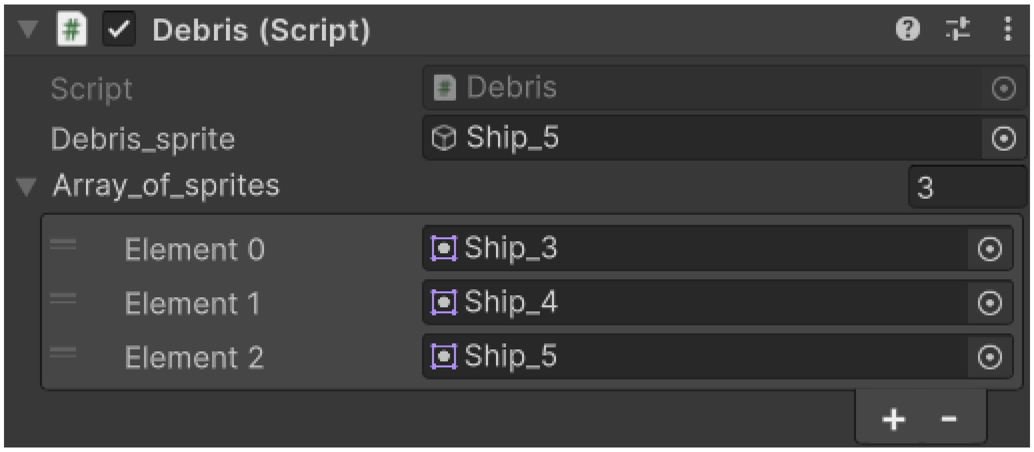
8. We then need to create a new function:



```
public void SetSprite(int i)
{
    //Sets the sprite of the debris.
    debris_sprite.GetComponent<SpriteRenderer>().sprite = array_of_sprites[i];
}
```

Similar to what was done in the asteroids start function, we are just setting the sprite of the debris_sprite game object. However instead of picking a sprite at random we are setting it to the array slot value of i. The value I is passed in to the function.

9. Save the script and go to the **Debris prefab scene**. On the **Debris** object, find the **Debris Script** component. Set the **Debris_Sprite** to the **child sprite object**. **Set** the **size** of the **Array_Of_Sprites** to **5** and then drag in the last three sprites of the ship sprite sheet into the array slots.



Debris (Script)

Script: Debris
Debris_sprite: Ship_5
Array_of_sprites: 3
Element 0: Ship_3
Element 1: Ship_4
Element 2: Ship_5

10. Open the **Player** script and add the variable:

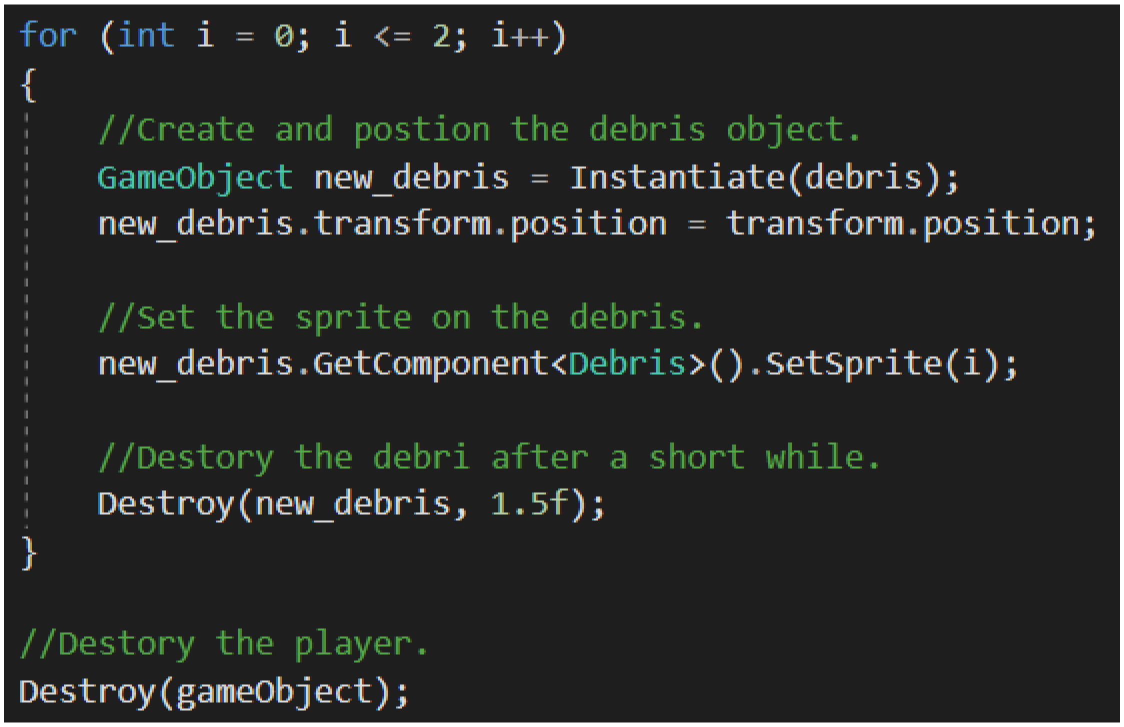
```
//Collision
public GameObject debris;
```

Debris is going to store a reference to the debris prefab object we made.

11. Next add the function:

```
private void OnTriggerEnter2D(Collider2D collision)
{

}
```

Then inside let's write the code:

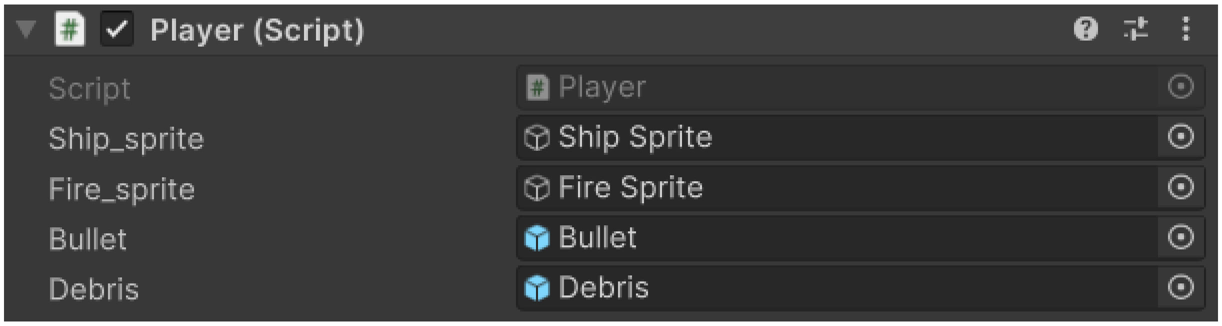


```
for (int i = 0; i <= 2; i++)
{
    //Create and postion the debris object.
    GameObject new_debris = Instantiate(debris);
    new_debris.transform.position = transform.position;

    //Set the sprite on the debris.
    new_debris.GetComponent<Debris>().SetSprite(i);

    //Destory the debri after a short while.
    Destroy(new_debris, 1.5f);
}

//Destory the player.
Destroy(gameObject);
```

We create a for loop to run three times. In the loop, we create a new debris object, set the sprite of the debris and then tell the debris object that it should destroy itself after 1.5 seconds.

Outside of the loop, we tell the player to destroy itself since it hit an asteroid.

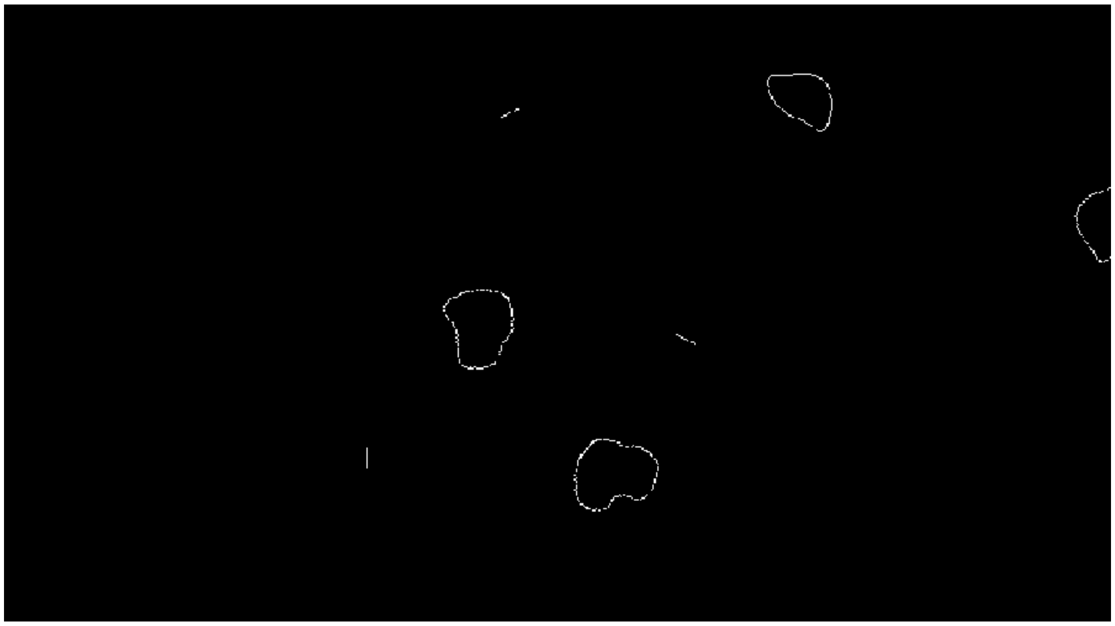
12. Save the scripts and return to Unity. All we need to then do is give the Player script a reference to the Debris prefab object.



Player (Script)

Script	Player
Ship_sprite	Ship Sprite
Fire_sprite	Fire Sprite
Bullet	Bullet
Debris	Debris

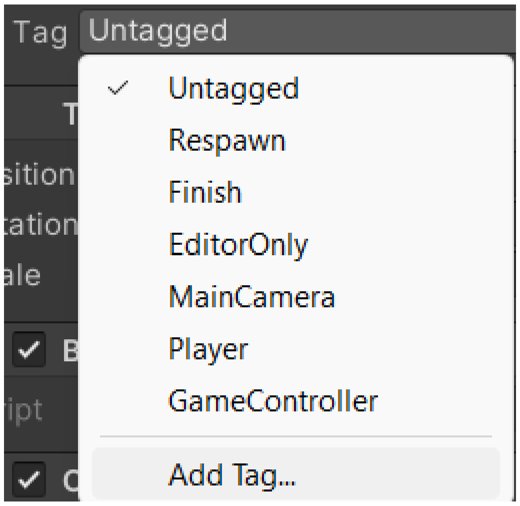
13. If we run the game now and crash into an asteroid the player will explode into three parts.



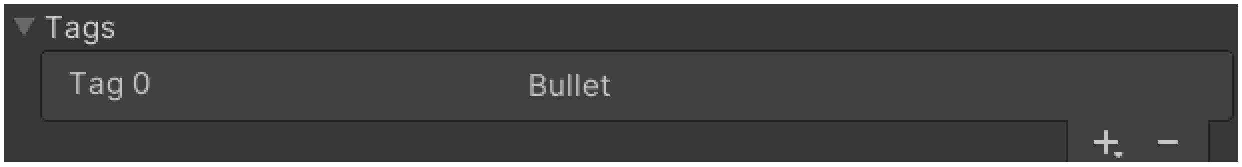
Bullet Collisions with Asteroids

1. Now that the asteroids can harm the player, we should add a way for the player to harm the asteroids. To do so, we will program the collision behavior between a bullet and an asteroid.

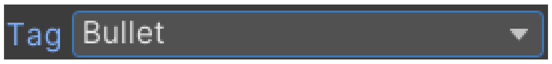
   First, we need to add the **Bullet tag** to our bullet prefab. Click on the **Bullet prefab** and go to its tag. Once there select the tag and then select add tag at the bottom.

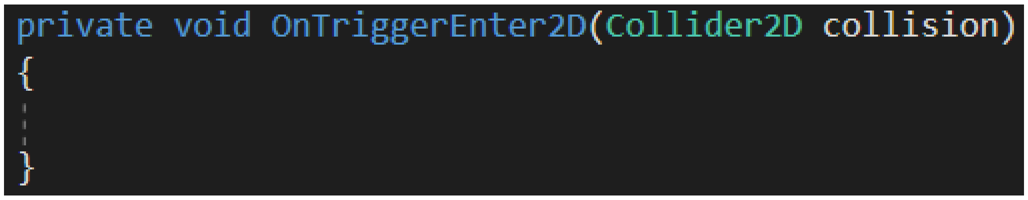
   Once in the tag menu, hit the **plus sign** to add a new tag and then name that tag **Bullet**.



Return to the bullet prefab and then assign it the **Bullet tag**.

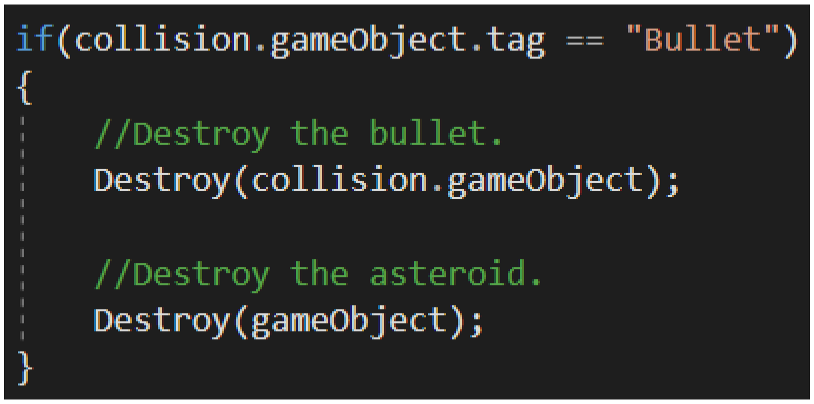


2. Next open up the **Asteroid** script. We then need to add the function:



```
private void OnTriggerEnter2D(Collider2D collision)
{

}
```

3. Then inside we write the code:

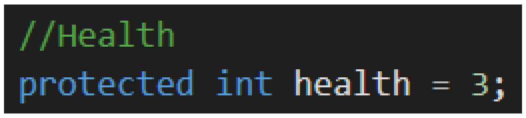


```
if(collision.gameObject.tag == "Bullet")
{
    //Destroy the bullet.
    Destroy(collision.gameObject);

    //Destroy the asteroid.
    Destroy(gameObject);
}
```

First we check to see if the object we collided with is tagged as a bullet.
If so, we destroy the collisions' game object, being the bullet.
We then destroy our game object being the asteroid.

4. If we run the game now, we are able to shoot and destroy the asteroids. However, since these are the bigger asteroids they should take more than one hit to destroy.

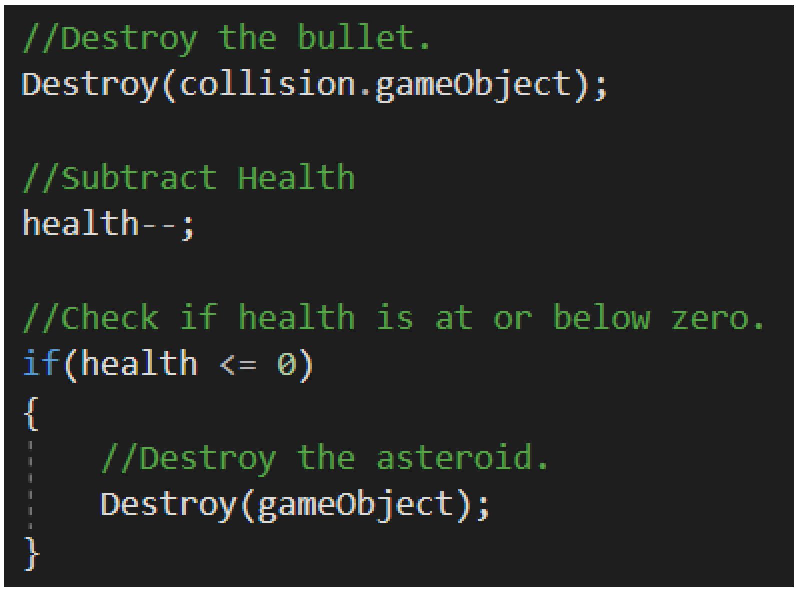
If we go back inside the **Asteroid** script lets add the variable:



```
//Health
protected int health = 3;
```

Health will determine how many hits the asteroid is able to take before it is destroyed.

5. Then back in the **OnTriggerenter2D()** function lets change the code to read:



```
//Destroy the bullet.
Destroy(collision.gameObject);

//Subtract Health
health--;

//Check if health is at or below zero.
if(health <= 0)
{
    //Destroy the asteroid.
    Destroy(gameObject);
}
```

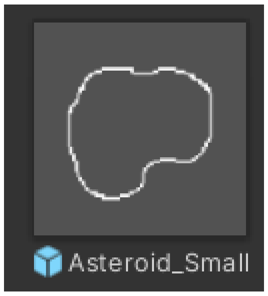
We first add some code to subtract one from the health of the asteroid.
If the health is less than or equal to zero then we destroy the asteroid.

6. Running the game now, we can see that the asteroid will now take three shots to be destroyed.

Small Asteroids

1. One of the things that makes Asteroids challenging as a game is when the player shoots a big asteroid, it breaks off into smaller asteroids, causing more trouble for the player. Let's add this feature to our game next by creating smaller asteroids.

   To do so, select the **Asteroid prefab** and **duplicate** it by pressing **Ctrl + D**. Rename the prefab **Asteroid_Small**.

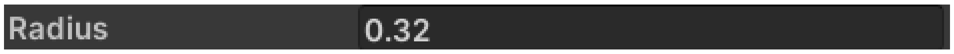


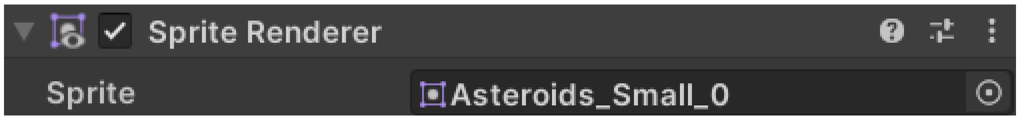
   As a short aside, something we can do with our prefabs is create **Prefab Variants.** To do this we right click the prefab and select **Create > Prefab Variant**.
   A variant creates a duplicate of the prefab however if we change the base prefab, that

change will also appear in the variant. But if we change something in the variant, those changes will not be reflected in the base prefab.
This basically allows use to use inheritance for our prefabs.

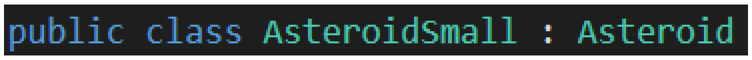
I only wanted to bring this up since it is something we can do to prefabs. However, we are choosing to not make a variant since our smaller asteroids are just different enough.

2. Select the **Asteroid_Small prefab**. In the inspector let's change a few things. First, we are going to be **removing** the **Asteroid script component**.

   Next let's set the **Radius** of the **CircleCollider2D** component to **0.32.**
   
   Radius 0.32

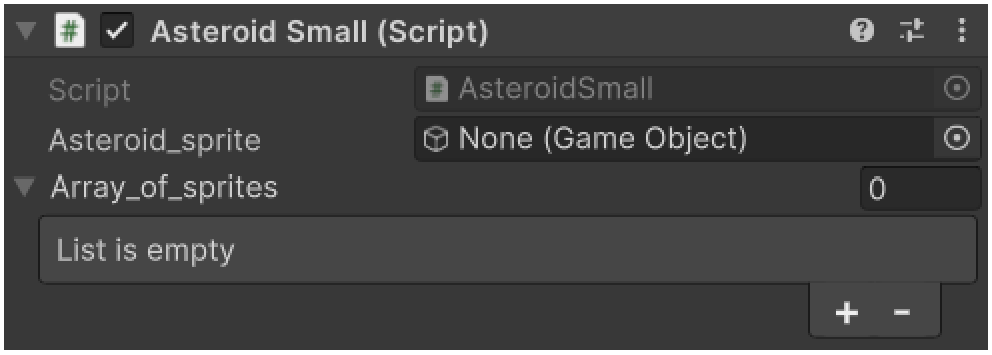
3. Next select the **Asteroid_Sprite** child game object and set its sprite to the first sprite of the **Asteroid_Small sprite sheet.**
   
   Sprite Renderer
   Sprite Asteroids_Small_0

4. Next create a new script called **AsteroidSmall** and then drag it onto the **Asteroid_Small** inspector.

5. Open the script and then have it inherited from **Asteroid**.
   
   ```
   public class AsteroidSmall : Asteroid
   ```
   Since the asteroid script has a bunch of the functionality that our asteroid needs, inheriting from it seems like a good idea.

6. For right now let's **remove** the **Start()** and **Update()** functions. We are only do this so these to functions will not override what the asteroid script's Start() and Update() functions do.

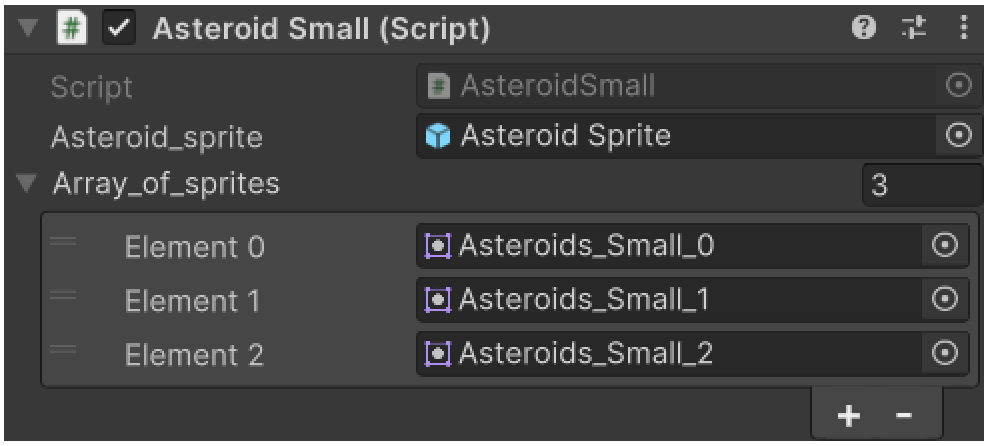
7. Save the script and return to Unity. Since the Asteroid_Small() script inherits from Asteroid script, we will see the same variables in the inspector.


Asteroid Small (Script)

Script	AsteroidSmall
Asteroid_sprite	None (Game Object)
Array_of_sprites	0

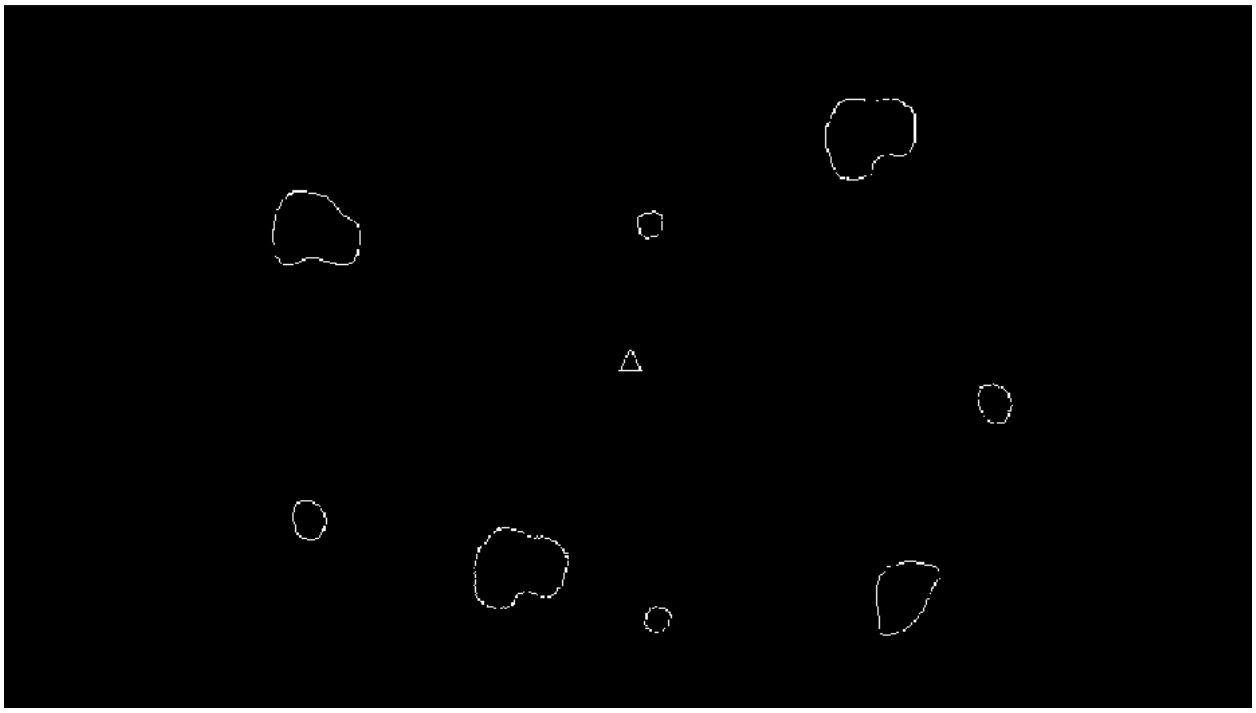
List is empty

Now all we need to do is fill in the properties of the script with the proper game objects.


Asteroid Small (Script)

Script	AsteroidSmall
Asteroid_sprite	Asteroid Sprite
Array_of_sprites	3
Element 0	Asteroids_Small_0
Element 1	Asteroids_Small_1
Element 2	Asteroids_Small_2

8. If we drag a couple of the Asteroid_Small prefabs and place them in our screen, once the game is run, we should see them act just like the bigger asteroids.

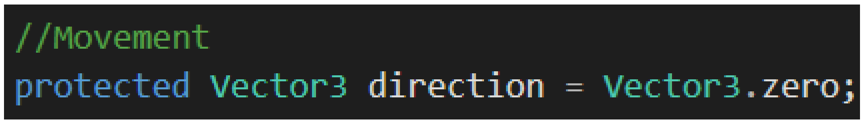


AsteroidSmall Script

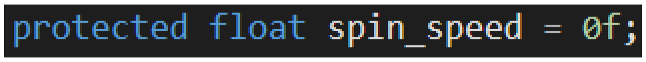
1. Next, we need to make the smaller asteroids act a little bit differently than our larger asteroids. To differentiate them, maybe they can move a little bit fast and take only one shot to destroy. We also need to define a slightly different behavior for when the small asteroids collides with a bullet.

   First lets open the **AsteroidSmall** script and add the **Start()** function back in.

2. Then let's go into the **Asteroid** script. We need to change the protection level of a few variables so our AsteroidSmall script will be able to access them.

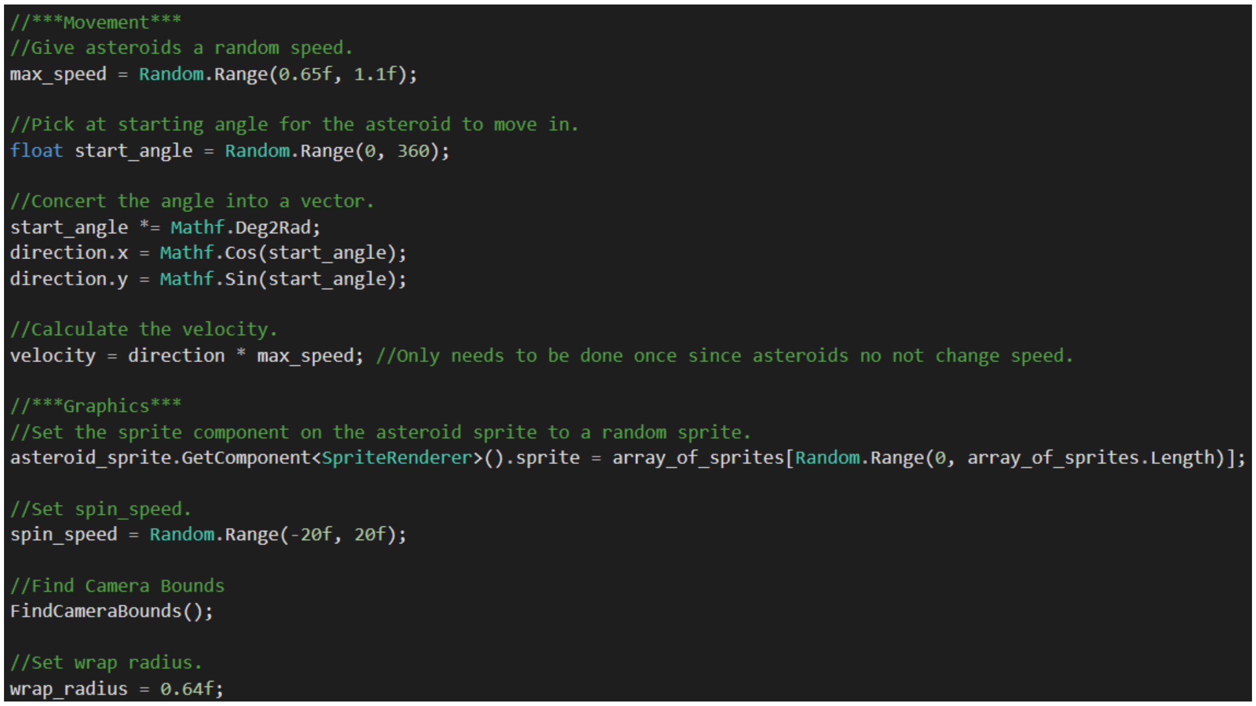


```
//Movement
protected Vector3 direction = Vector3.zero;
```



```
protected float spin_speed = 0f;
```

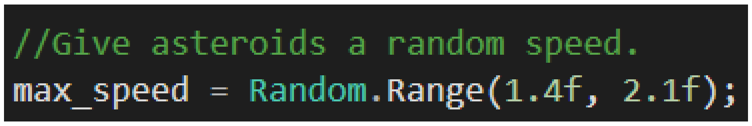
3. Then copy the contents of the **Start()** function. Head back to the **AsteroidSmall** script and paste the contents inside that scripts **Start()** function.



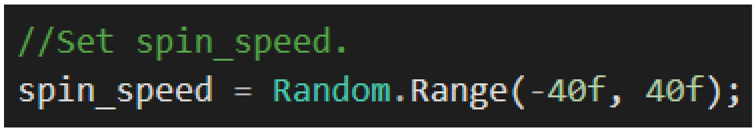
```
//***Movement***
//Give asteroids a random speed.
max_speed = Random.Range(0.65f, 1.1f);

//Pick at starting angle for the asteroid to move in.
float start_angle = Random.Range(0, 360);

//Concert the angle into a vector.
start_angle *= Mathf.Deg2Rad;
direction.x = Mathf.Cos(start_angle);
direction.y = Mathf.Sin(start_angle);

//Calculate the velocity.
velocity = direction * max_speed; //Only needs to be done once since asteroids no not change speed.

//***Graphics***
//Set the sprite component on the asteroid sprite to a random sprite.
asteroid_sprite.GetComponent<SpriteRenderer>().sprite = array_of_sprites[Random.Range(0, array_of_sprites.Length)];

//Set spin_speed.
spin_speed = Random.Range(-20f, 20f);

//Find Camera Bounds
FindCameraBounds();

//Set wrap radius.
wrap_radius = 0.64f;
```

   In truth we could probably break up the code into a couple functions and have those functions be in the asteroid script for us to call. But we can just have things be a little bit messy for now.

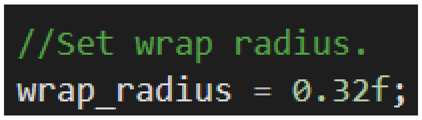
4. Alright, lets change a few variables:



```
//Give asteroids a random speed.
max_speed = Random.Range(1.4f, 2.1f);
```



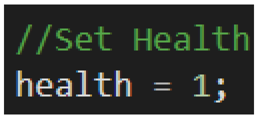
```
//Set spin_speed.
spin_speed = Random.Range(-40f, 40f);
```



```
//Set wrap radius.
wrap_radius = 0.32f;
```

All we are really doing is just speeding up our smaller asteroid.
We also change the wrap_radius to make screen wrapping a little bit more seamless.

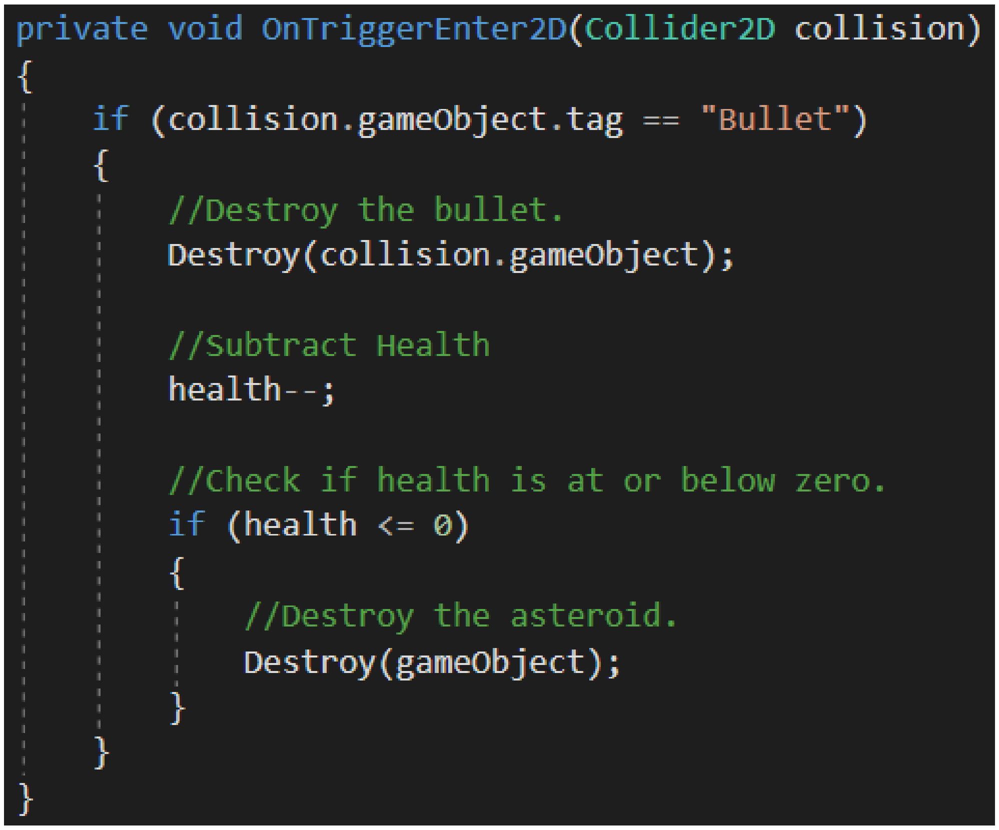
5. Let's also write the code:



```
//Set Health
health = 1;
```

We change the health of the asteroid to one so that smaller asteroids will be easier to destroy.

6. Then head back to the **Asteroid** script, **copy** the **OnTriggerEnter2D()** function and then paste it into the **AsteroidSmall** script.



```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.tag == "Bullet")
    {
        //Destroy the bullet.
        Destroy(collision.gameObject);

        //Subtract Health
        health--;

        //Check if health is at or below zero.
        if (health <= 0)
        {
            //Destroy the asteroid.
            Destroy(gameObject);
        }
    }
}
```

We are going to leave this the same for right now but we will be adding changes to both the OnTriggerEnter2D functions for the large and small asteroid.

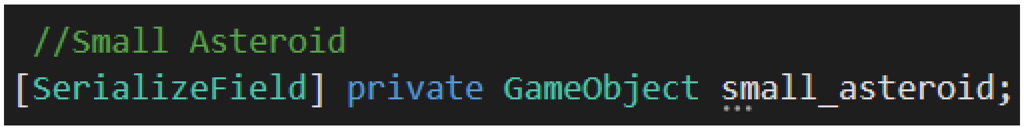
Note: Since we are calling the OnTriggerEnter2D() function in this script, it will override the OnTriggerEnter2D() function from the inherited script.

7. If we save the script and run the game. We should see that the smaller asteroids reflect the changes that we made in the script.

Small Asteroid Spawning

1. So now that our small asteroids are functional, we need them to spawn in when a bigger asteroid is destroyed.

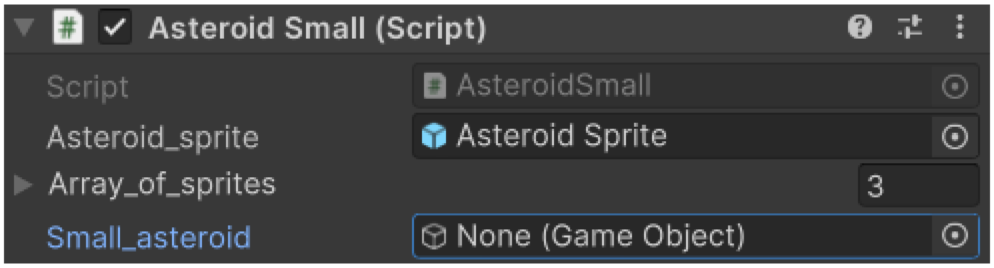
   To do this, in the **Asteroid** script, lets add the variable:



```
 //Small Asteroid
[SerializeField] private GameObject small_asteroid;
```

   Small_Asteroid will hold a reference to the small asteroid prefab we made.
   [SerializeField] is a way to make a private variable appear in the inspector.
   We want this variable to be private since the AsteroidSmall script doesn't really need to have access to it since small asteroids should not be able to spawn themselves when destroyed.

   Note: Since we are making this private variable appear in the inspector and since the AsteroidSmall script inherits from Asteroid, this variable will also appear in its inspector as well.



```
Asteroid Small (Script)
Script              AsteroidSmall
Asteroid_sprite     Asteroid Sprite
Array_of_sprites    3
Small_asteroid      None (Game Object)
```

   Which can be a little bit jarring since, that script will not have access to it.

   However, it we wanted to avoid having this variable appear it seems like we could use **this method** outlined in a Unity discussion forum reply. We would load the prefab though the code rather than the inspector. Sometimes it might be worth taking the extra steps just to ensure things are nice and clean.

2. Next, we can create the function:

```
private void spawn_small_asteroid()
{

}
```

Then inside we can write the code for instancing the small asteroids.

```
//Instance small asteroid.
int num = Random.Range(2, 5);
for(var i = 0; i < num; i++)
{
    //Instance asteroid.
    GameObject new_small_asteroid = Instantiate(small_asteroid);

    //Position the asteroid.
    new_small_asteroid.transform.position = transform.position;
}
```

We first declare a local variable called **num** that will store a random value between **2** and **4**.
Note: For whatever reason, when the Random.Range function uses integers, the max number is exclusive meaning it will not be used. So, the max number is one less than what was put.
We then write a for loop that will loop the number of times as the num variable.
Inside we instantiate the small asteroid and then move it to the position of the asteroid that spawned.

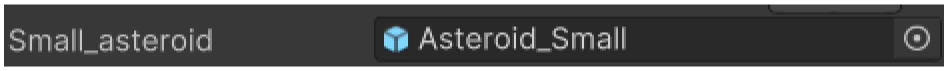
3. Let's then head up to the **OnTriggerEnter2D()** function. In the if statement that check to see if our health is less than zero, we write the code:

```
spawn_small_asteroid();
```

Here we just perform a function call to spawn in our small asteroids.

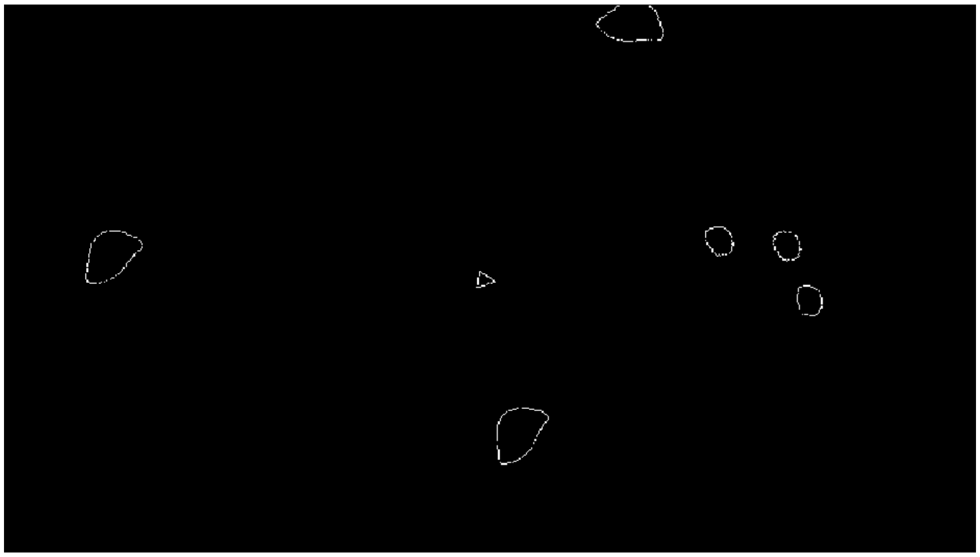
Note: Since we added this line, the Asteroid and the AsteroidSmall script's OnTriggerEnter2D() functions are now different. The Asteroid script will spawn smaller asteroids while the AsteroidSmall script will not. This is one of the reasons why we wanted to two separate OnTriggerEnter2D() functions for each type of asteroid.

4. If we save the script and return to Unity. We need to give the Asteroid prefab a reference to the Asteroid_Small prefab.


Small_asteroid Asteroid_Small

Make sure you are setting this property in the Asteroid prefab and not the prefabs that are in the scene.

5. If we run the game, when a large asteroid is destroyed it will spawn two to four small asteroids.



Using Events to Spawn Asteroids

1. Create a new **Empty** game object and name it **Game Logic**.

Game Logic

2. Then create two new scripts. One called **Events** and the other called **AsteroidSpawner**.

Attach both of the script to the GameLogic object.

Events will keep track of the events we need to use through out our game. AsteroidSpawner will have the task of spawning asteroids once all the asteroids in the game are gone.

3. Open the **Events** script and include the library:

   ```
   using System;
   ```

   This library will allow us to have access to a C# class called **Actions** that we will use to make our events.

4. Inside of our class lets add the variable:

   ```
   //Events for our game.
   public static Action Asteroid_Was_Destroyed;
   ```

   Asteroid_Was_Destroyed is the event we will invoke to tell the spawner that it should create more asteroids.

5. Let's then open the **AsteroidSpawner** script and at the top, lets add the variable:

   ```
   //Asteroid to spawn.
   public GameObject asteroid;
   ```

   Asteroid will hold a reference to the asteroid prefab that is in our game.

6. Next lets create a new function called **SpawnAsteroid()**.

   ```
   private void SpawnAsteroid()
   {

   }
   ```

   Inside we can write the code:

   ```
   int num = Random.Range(2, 5);
   for (var i = 0; i < num; i++)
   {
       //Instance asteroid.
       GameObject new_asteroid = Instantiate(asteroid);

       //Position the asteroid.
       new_asteroid.transform.position = Vector3.zero;
   }
   ```

   This is the same code we used when spawning small asteroids. The only change we made was to set the position of the asteroid to **Vector3.zero**. This will spawn asteroids at the center of the screen. We will change this later.

7. Then in the **Start()** function lets write the code:

```
//Subscribe to the event.
Events.Asteroid_Was_Destroyed += SpawnAsteroid;
```

What this line does it add the SpawnASteroid() function to the list of functions that are subscribed to the Asteroid_Was_Destoryed event.

What this means is that when the Asteroid_Was_Destoryed event is invoked, the SpawnAsteroid() function will be called.

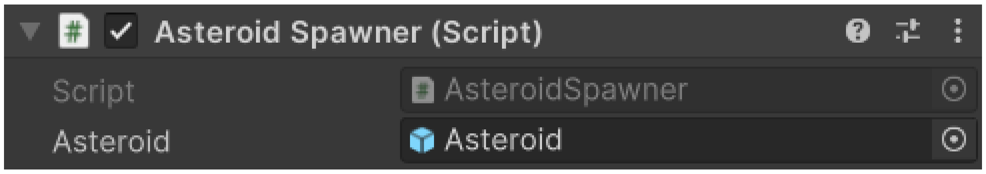
8. So now, the last thing we need to do is invoke the event. To do this, go to the **AsteroidSmall** script and find the **OnTriggerEnter2D()** function. Find the if statement where we check to see if its health is less than zero. Inside, add the line below underneath where we destroy the game object.

```
//Invoke the event.
Events.Asteroid_Was_Destroyed?.Invoke();
```

The if statement will look like:

```
//Check if health is at or below zero.
if (health <= 0)
{
    //Destroy the asteroid.
    Destroy(gameObject);

    //Invoke the event.
    Events.Asteroid_Was_Destroyed?.Invoke();
}
```

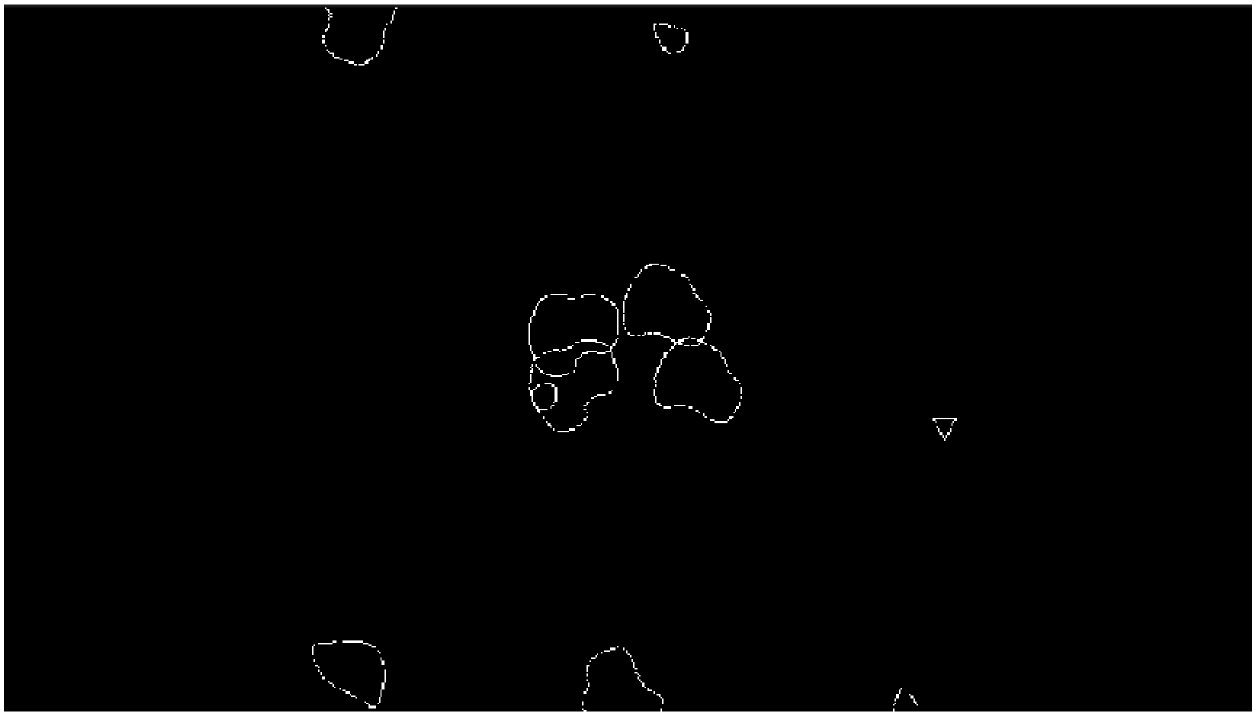
9. Save all the script and return to Unity. Give the **AsteroidSpawner** script component a reference to the **Asteroid prefab.**



Asteroid Spawner (Script)

Script	AsteroidSpawner
Asteroid	Asteroid

If we run the game, when a small asteroid id destroyed, two to four large asteroids will spawn.



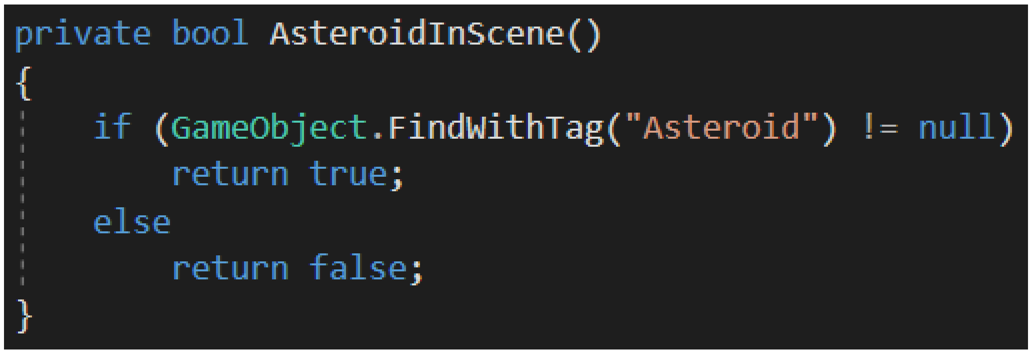
Fixing Asteroid Spawns

So right now, we have two problems that we are encountering with spawning in our asteroids. The first is that every time a small asteroid is destroyed, more asteroids will spawn in. The second is that they spawn in at the center of the screen.

To fix the first issue, we are going to add a check to the SpawnAsteroid function to see if any asteroids are left in the game. To fix the second issue, we will write a function for the asteroid that will allow them to spawn on the outside of the screen.

Checking for Asteroids in the Scene

1. To start, let's open the **AsteroidSpawner** script and add the function:

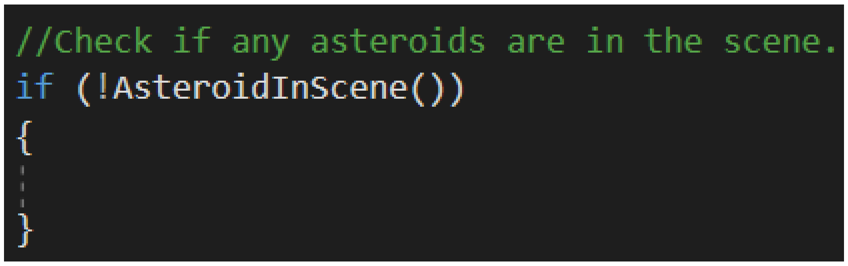


```
private bool AsteroidInScene()
{
    if (GameObject.FindWithTag("Asteroid") != null)
        return true;
    else
        return false;
}
```

All this function will do is return true if there is an asteroid in the scene and false if there is

not.
We are using the **FindWithTag()** function from the GameObject class. This is similar to the Find() function except instead of checking a name it checks the tag of the object.

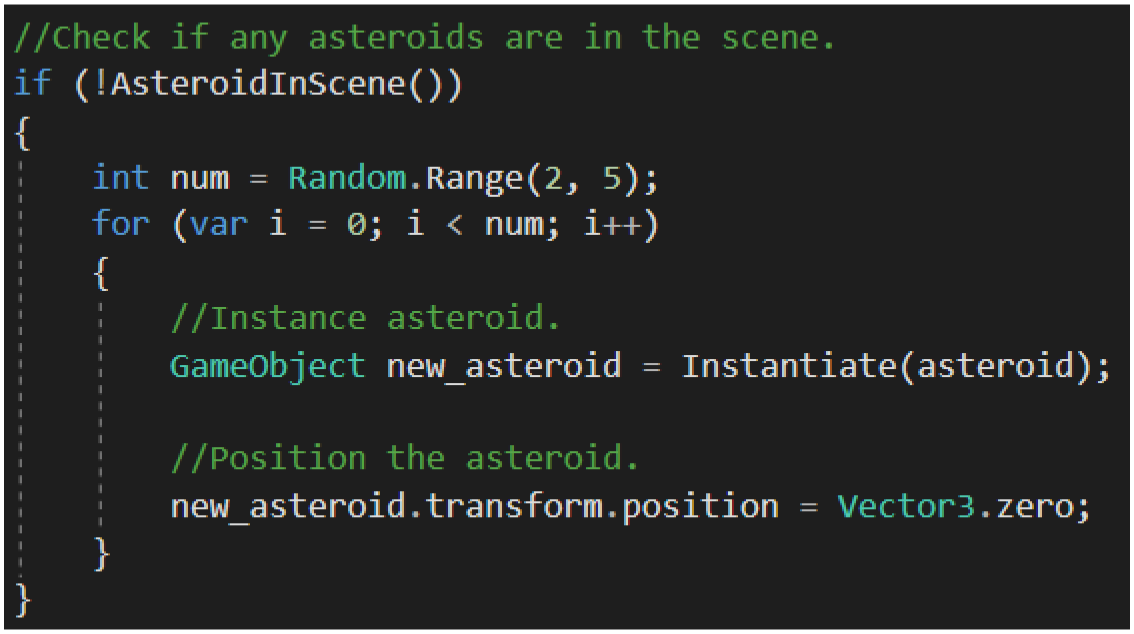
2. Then in the **SpawnAsteroid()** function lets add the if statement:



```
//Check if any asteroids are in the scene.
if (!AsteroidInScene())
{

}
```

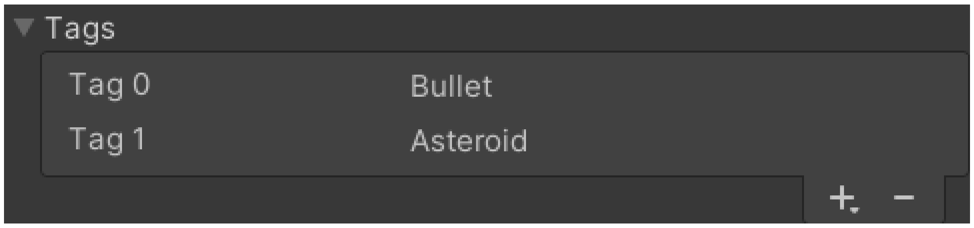
We get to use the function we just created as a condition for the statement. Since the function returns true if there is an asteroid in the scene, we invert what it returns by adding an exclamation mark sign.

3. We can then cut and paste the code outside the if statement and put it inside.



```
//Check if any asteroids are in the scene.
if (!AsteroidInScene())
{
    int num = Random.Range(2, 5);
    for (var i = 0; i < num; i++)
    {
        //Instance asteroid.
        GameObject new_asteroid = Instantiate(asteroid);

        //Position the asteroid.
        new_asteroid.transform.position = Vector3.zero;
    }
}
```

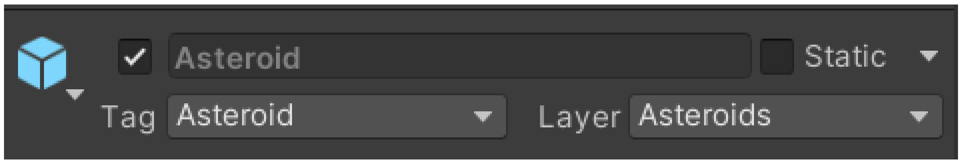
4. Save the script and return to Unity. We now just need to tag both the asteroid prefabs as asteroids. We create the **Asteroid tag**:

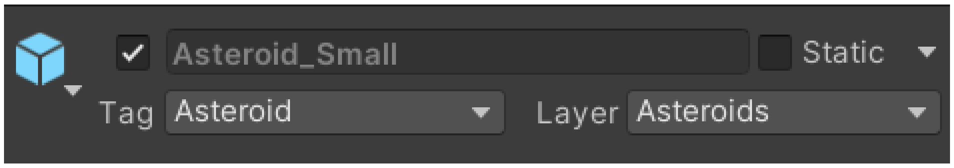


Tags

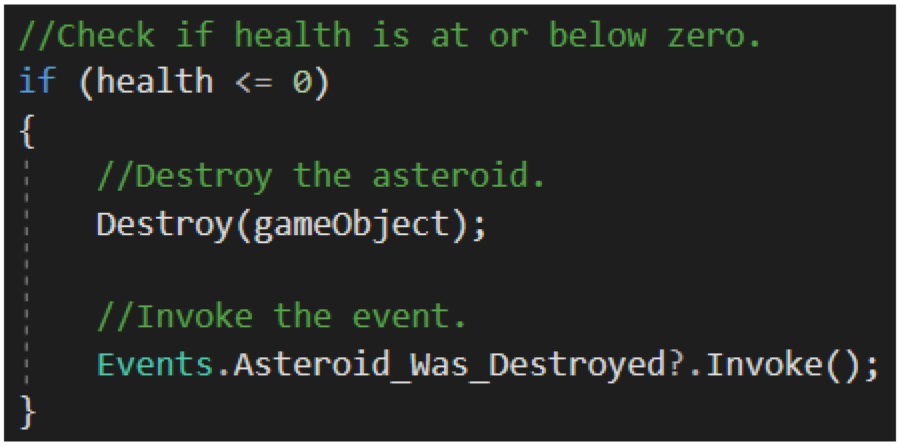
Tag 0	Bullet
Tag 1	Asteroid

Then we tag both the asteroids as such.


Asteroid
Static
Tag Asteroid
Layer Asteroids


Asteroid_Small
Static
Tag Asteroid
Layer Asteroids

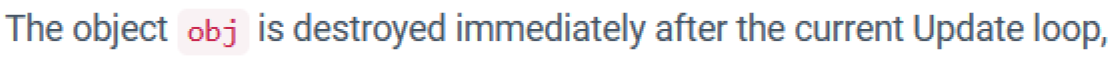
5. If we run the game, we are still going to experience a problem. Despite clearing out all of the asteroids, the game will not spawn any more. Let's look back at our code responsible for destroying the small asteroid.



```
//Check if health is at or below zero.
if (health <= 0)
{
    //Destroy the asteroid.
    Destroy(gameObject);

    //Invoke the event.
    Events.Asteroid_Was_Destroyed?.Invoke();
}
```

We destroy the object first and then invoke the event so we can tell the spawner to spawn more asteroids. We are doing things in the right order, so what gives?

Well, it is our pesky problem of execution order again. The issue is that the game object isn't fully destroyed until the end of update loop.


The object `obj` is destroyed immediately after the current Update loop,

It even says so in the Unity Documentation.

So as far as Unity is concern, our game object hasn't been destroyed yet, so when we go to check to see if an asteroid is in the scene, Unity says that there is one.

So in order to get around this issue, we need to add some sort of delay.

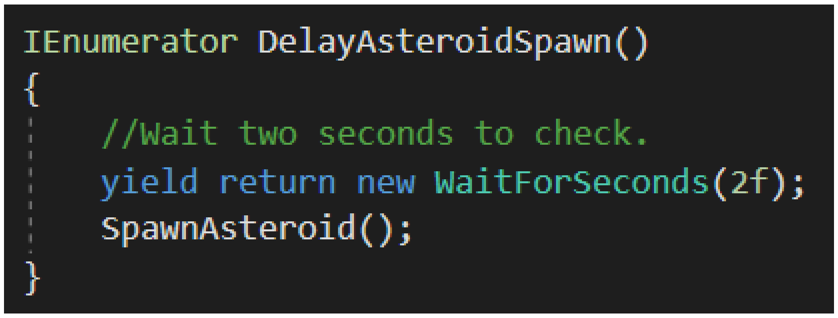
Coroutines

First and foremost, I need to apologize. I am clearly tacking on way more new things to learn in this lesson than I originally intended. I just wanted to cover Events since it helps objects in our game communicate with each other but then while making the game I sort of realized that we need to use coroutines. This lesson plan is already 25 pages at this point, I need to stop. It's time to stop.

Coroutines allow us to spread out a task in code across several frames. We could delay a function from running by a couple seconds or we could have another function run at the end of a frame.

With coroutines we can now delay the check needed to see if asteroids are currently in the scene.

1. To create our coroutine, lets add the function below to the **AsteroidSpawner** script.

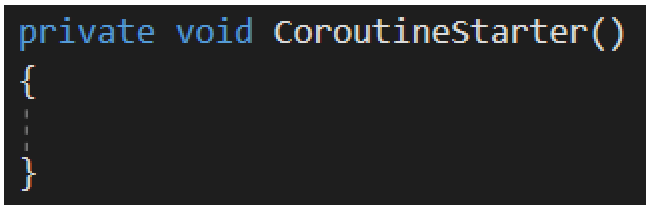
   ```
   IEnumerator DelayAsteroidSpawn()
   {
       //Wait two seconds to check.
       yield return new WaitForSeconds(2f);
       SpawnAsteroid();
   }
   ```

   Coroutine methods are declared with the **IEnumerator** variable type.
   The need to also have a **yield return** statement included in them. For our statement we are using the **WaitForSeconds()** function. This function can only be used with a yield return for coroutines. We pass in 2f to wait for two seconds.
   We then call the SpawnAsteroid() function.

   Effectively when this code is run, the coroutine will say that Unity needs to wait for two seconds before it will resume looking at the code. Once the two second are up, the SpawnAsteroid() function will run.

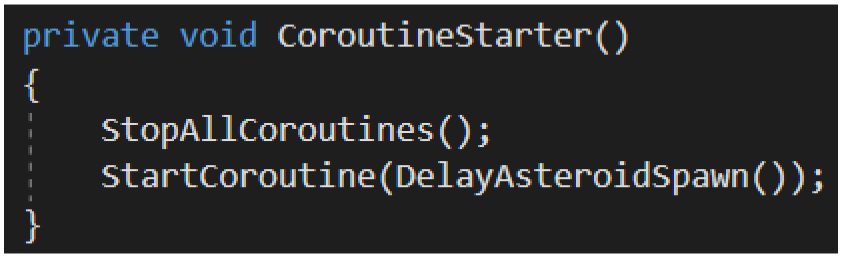
   So that will be our coroutine.

2. Next, we just need a way to **start** the coroutine. To do that lets create the function

   ```
   private void CoroutineStarter()
   {

   }
   ```

   Then inside, lets add the code:

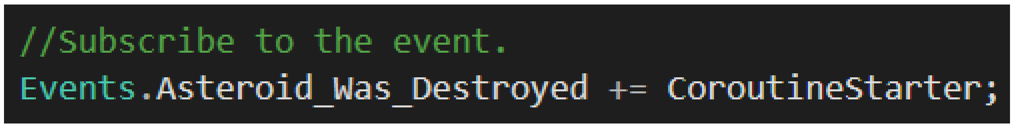
   

   ```
   private void CoroutineStarter()
   {
       StopAllCoroutines();
       StartCoroutine(DelayAsteroidSpawn());
   }
   ```

First, we tell any running coroutines to stop. We do this to ensure that there will always be a two second delay between the last asteroid being destroyed and the new one's spawning. We then just tell our DelayAsteroidSpawn coroutine to run.

Deep Explanation as to why we stop all coroutines. If we didn't have this there could be a scenario where the player could destroy two asteroids in quick succession. The first asteroids would start the coroutine, then after two seconds it would see that there are no more asteroids in the scene. However, since there was a little gap of time that passed when the last asteroid was destroyed, the game didn't wait two seconds after the last asteroid was destroyed to spawn new ones in. We wanted to do this to give the player a sense of consistency when playing the game. the player should almost instinctually know that the game waits a few seconds before new asteroids spawn even if we do not directly tell that information to the player.

3. Lastly, in the **Start()** function we are going to change what function subscribes to the event.

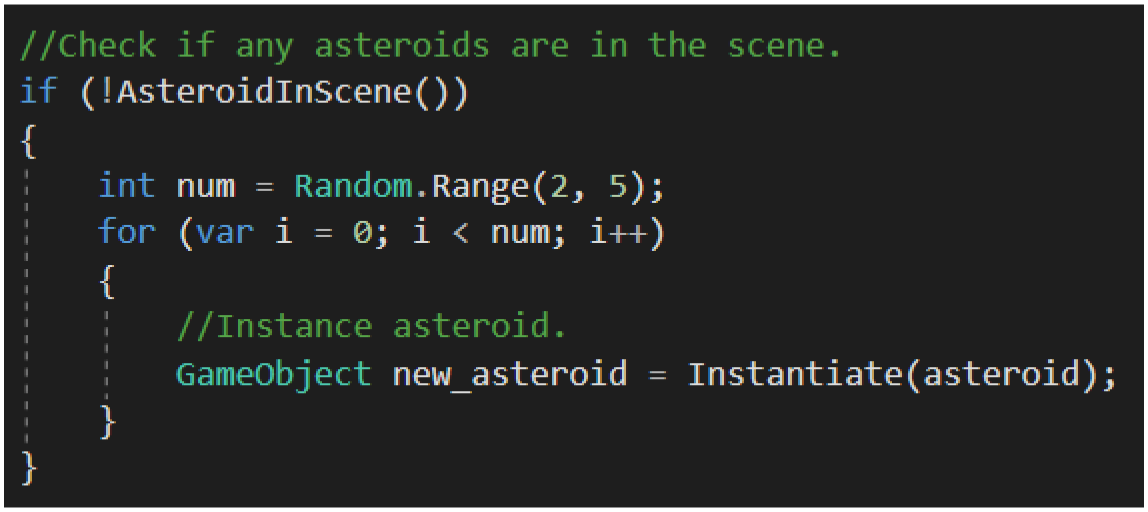


```
//Subscribe to the event.
Events.Asteroid_Was_Destroyed += CoroutineStarter;
```

4. So now if we play the game, the asteroids will take two seconds to spawn in after the last asteroid has been destroyed.

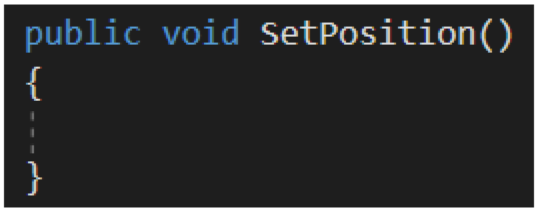
Spawning Off Screen

1. First, in the **SpawnAsteroid()** function remove the code for setting the position of the asteroid.



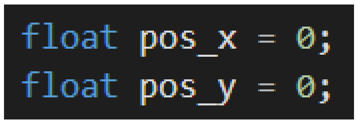
```
//Check if any asteroids are in the scene.
if (!AsteroidInScene())
{
    int num = Random.Range(2, 5);
    for (var i = 0; i < num; i++)
    {
        //Instance asteroid.
        GameObject new_asteroid = Instantiate(asteroid);
    }
}
```

2. Then lets go to the **Asteroid** script and add function:



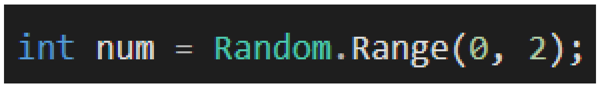
```
public void SetPosition()
{

}
```

3. At the top of the function we declare variables for holding the position of the asteroid.



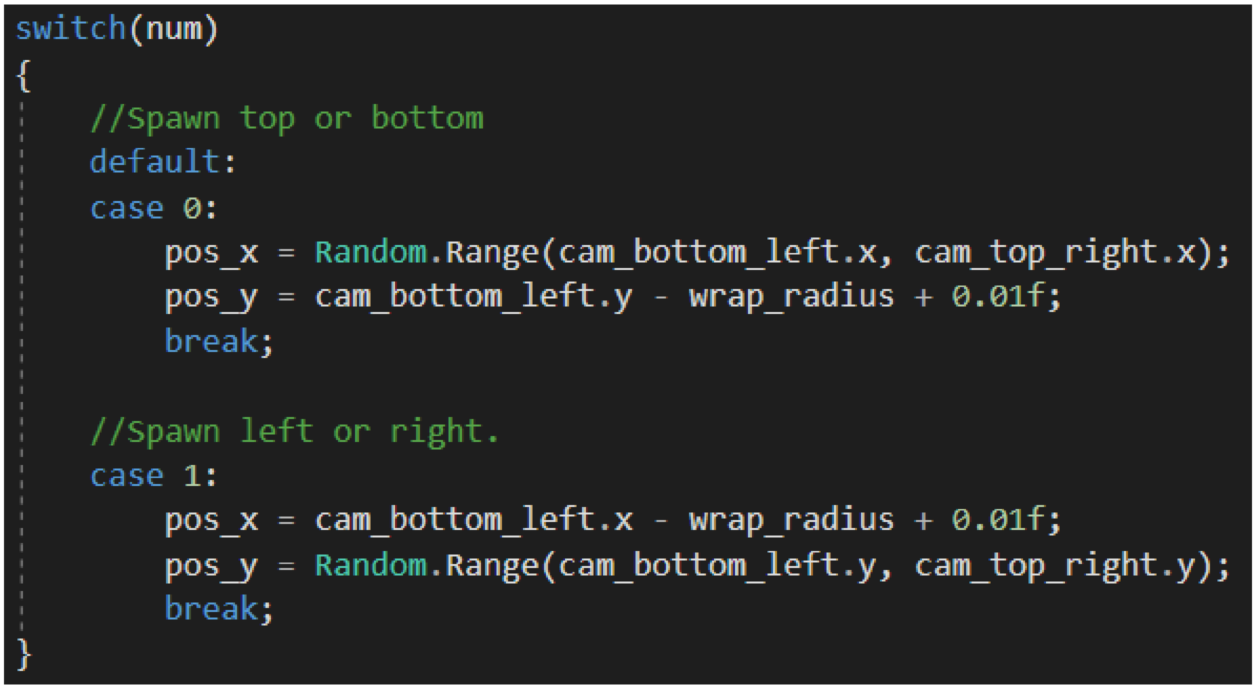
```
float pos_x = 0;
float pos_y = 0;
```

We also add a variable that will pick be assigned a value between 0 and 1.



```
int num = Random.Range(0, 2);
```

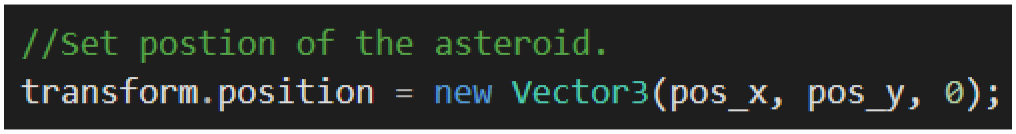
4. Underneath we write the switch statement:



```
switch(num)
{
    //Spawn top or bottom
    default:
    case 0:
        pos_x = Random.Range(cam_bottom_left.x, cam_top_right.x);
        pos_y = cam_bottom_left.y - wrap_radius + 0.01f;
        break;

    //Spawn left or right.
    case 1:
        pos_x = cam_bottom_left.x - wrap_radius + 0.01f;
        pos_y = Random.Range(cam_bottom_left.y, cam_top_right.y);
        break;
}
```

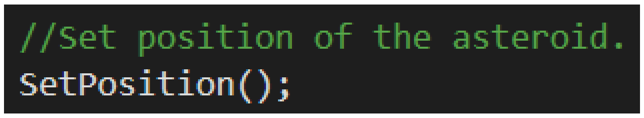
All we are doing is setting the pos_x and pos_y variable to place the asteroid just off screen.

5. Under the switch statement we write the code to position the asteroid.



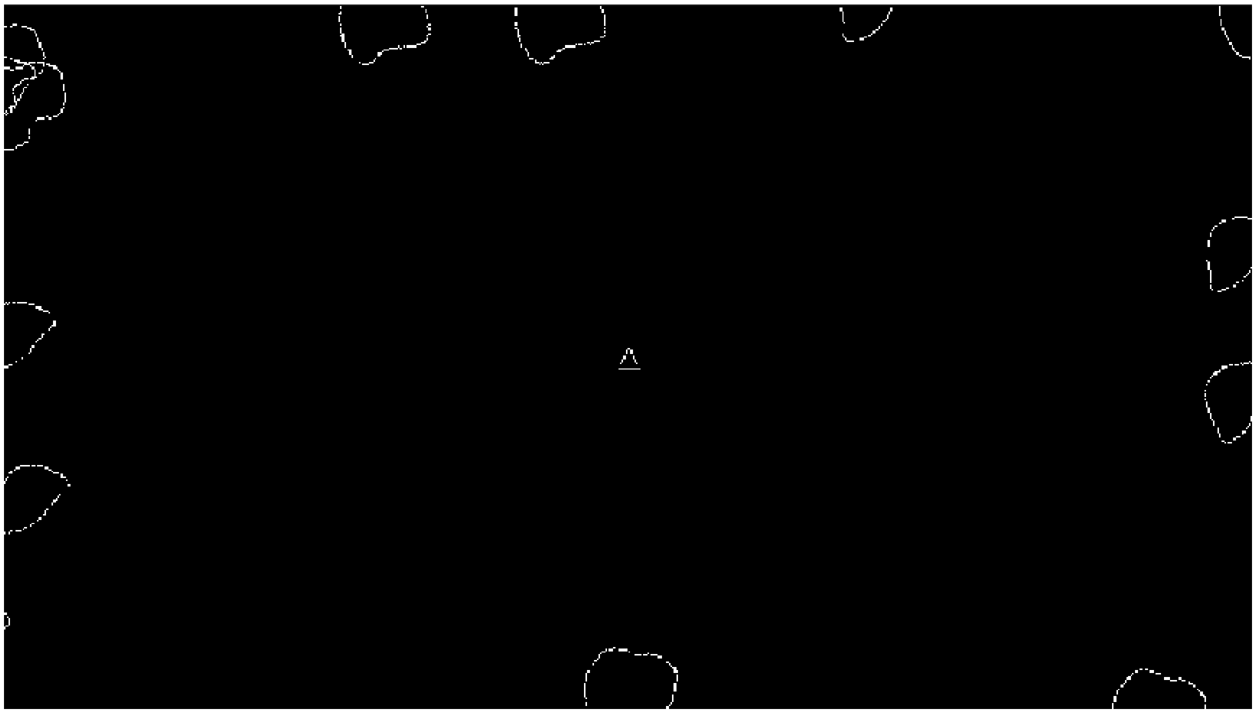
```
//Set postion of the asteroid.
transform.position = new Vector3(pos_x, pos_y, 0);
```

6. At the bottom of the start function we just need to call the SetPostion() function we wrote:



```
//Set position of the asteroid.
SetPosition();
```

7. Saving the script and returning to Unity we can see that when all the asteroids are destroyed new ones will spawn just off screen.


Keep in mind that since the asteroids are now setting their own position, any asteroids at the start of the game will also set their position.

GitHub

1. Push the project to the GitHub repository.