Beetle Mania: Part 2 – Shells

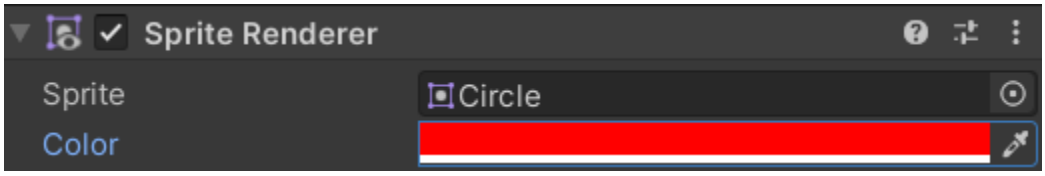
Lesson Goals

1. Create a shell object for our game.
2. Learn how to organize and test collisions with layers and tags.
3. Learn various ways of manipulating a rigidbody to achieve the intended physics of the object.
4. Create an object responsible for spawning shells within the game.
5. Code the relationship between shell and bullet collisions.

Shell Game Object

1. In order to start making the shells for our game, lets first add in a **2D Object > Sprites > Circle** game object into our scene. Let's rename this object to **Shell**.

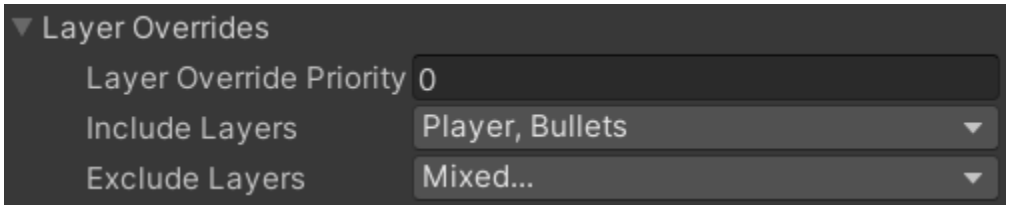
2. With the shell selected, find the **Sprite Renderer** in the inspector and set the color of the sprite to **Red**.



3. Next let's add a few components to our Shell. First, we are going to add **two Circle Collider 2D** components. The first one will be a trigger that is used to detect the player and bullets. The second one will be a normal collider that will be used to detect collisions with the walls and ground.

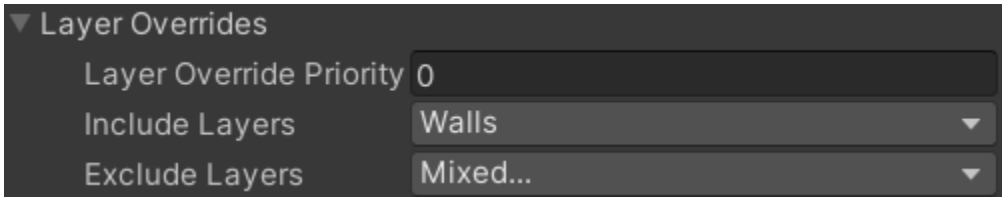
   One the **first** circle collider check the box labeled **IsTrigger**.

   Then in **Layer Overrides**, set the **Include Layers** to **Player** and **Bullets**.
   Then set the **Exclude Layers** to everything **except Player** and **Bullets.**



   On the **second** circle collider, go to the **Layer Overrides** and set the **Include Layers** to

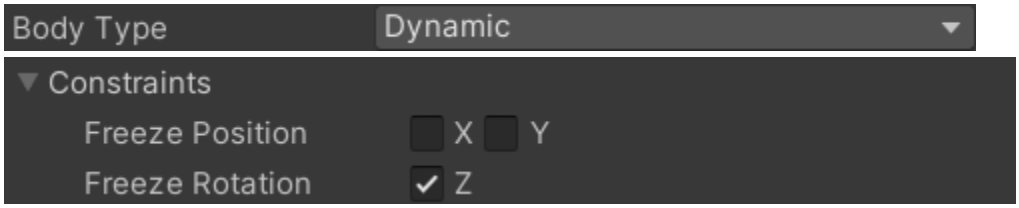
just **Walls**.
Then set the **Exclude Layers** to everything **except Walls**.



This set up may be a little jarring but we are effectively making it so the shell will be able to pass through the player and any bullets while still detecting collisions with them. However, when the shell touches a wall, it should bounce off of it instead of being able to pass through.

4. Let's then add a **Rigidbody 2D** component to our shell object.

Make sure the **Body Type** is set to **Dynamic** and that **Freeze Rotation** is set for the **Z axis**.



5. Let's then create a new script called **Shell** and then attach to the shell game object.

6. With all that done, lets drag the shell game object from the **hierarchy** and drop it into the assets folder to create a **Shell prefab**.

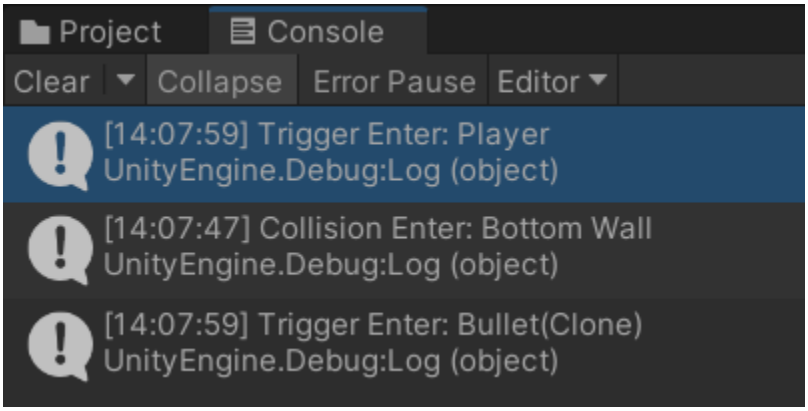
Organizing and Testing Collisions

1. So just a precaution, let's make sure our collisions are set up properly. To do this let's open up the Shell script and add the **OnCollisionEnter2D()** and **OnTriggerEnter2D()** functions.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    Debug.Log("Collision Enter: " + collision.gameObject.name);
}

private void OnTriggerEnter2D(Collider2D collision)
{
    Debug.Log("Trigger Enter: " + collision.gameObject.name);
}
```

Inside each function we are going to be checking name of the game object we collided with.

2. Save the script and run the game. The messages that should be output to the console are:



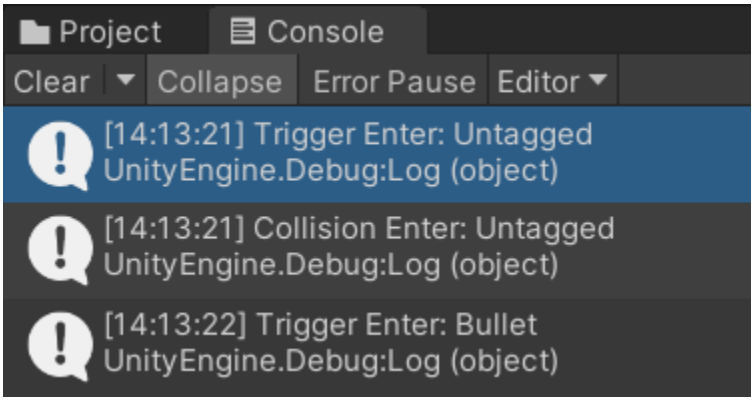
The shell should have just seamlessly passed through the player and the bullet while colliding with just the ground. Our trigger was able to successfully detect Player and the Bullet. While the collider was just able to detect the Bottom Wall.

If you got this result than that means your circle colliders should be good to go.

3. However, let's make a small change to our code to read out the tag of the objects the shell collided with:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    Debug.Log("Collision Enter: " + collision.gameObject.tag);
}

private void OnTriggerEnter2D(Collider2D collision)
{
    Debug.Log("Trigger Enter: " + collision.gameObject.tag);
}
```

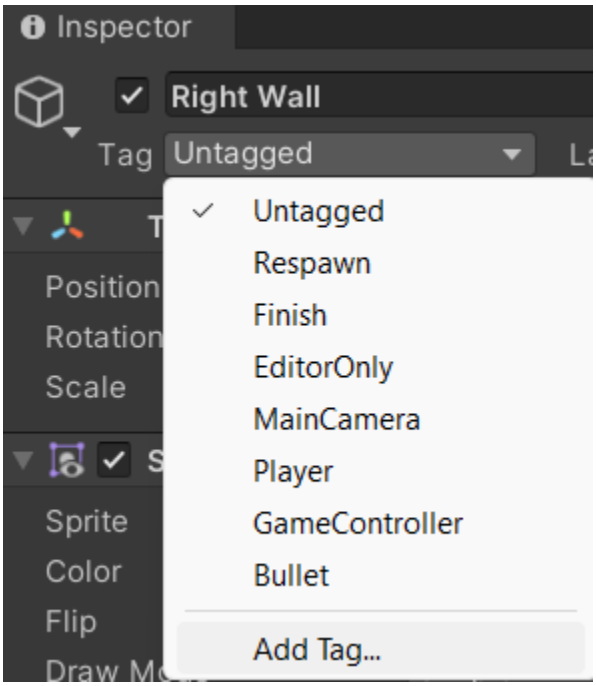
4. Now if we run our game and check the output we should see:



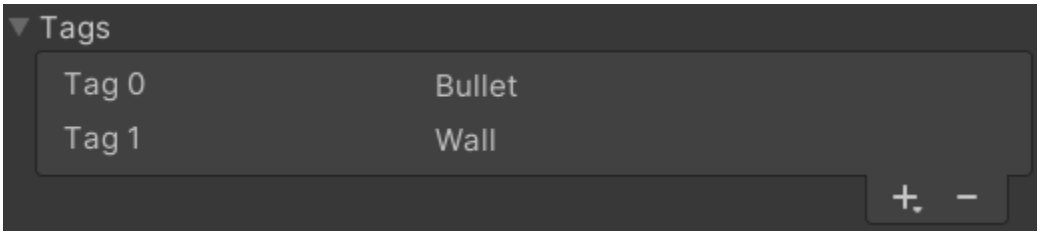
All of our collisions are still working correctly but the tags on our objects are a little unclear. Except for our bullets because we gave them a tag in the last lesson.

As stated in the last lesson, tags are a nice way to group objects together. **This is very handy when dealing with collisions.** For instance, if we did not use tags when checking to see if the shell collided with a wall, we would need to check for the Left, Right and Bottom wall individually. If we use tags, we can give all the walls the same tag and only check for that specific tag. So, let's do that for our project!

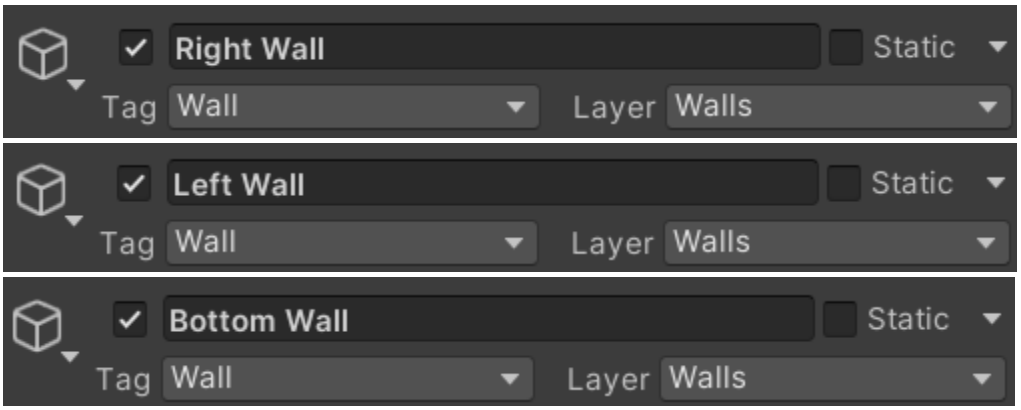
5. Select a wall and in the inspector find the **Tag** drop down. At the bottom select **Add Tag**.



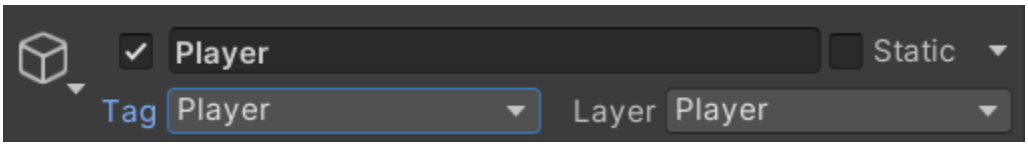
6. Once in the Tags menu, select the **plus sign** and add a new tag called **Wall**.



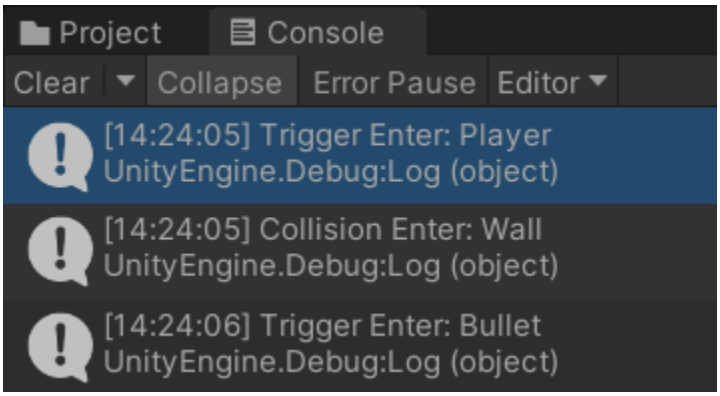
7. Then select a wall and assign it the new Wall tag. **Do this for each of the walls.**



8. We should be sure to give the player the **Player** tag as well.



9. If we run the game now, we should be able to see the proper tags of the objects that were collided with.



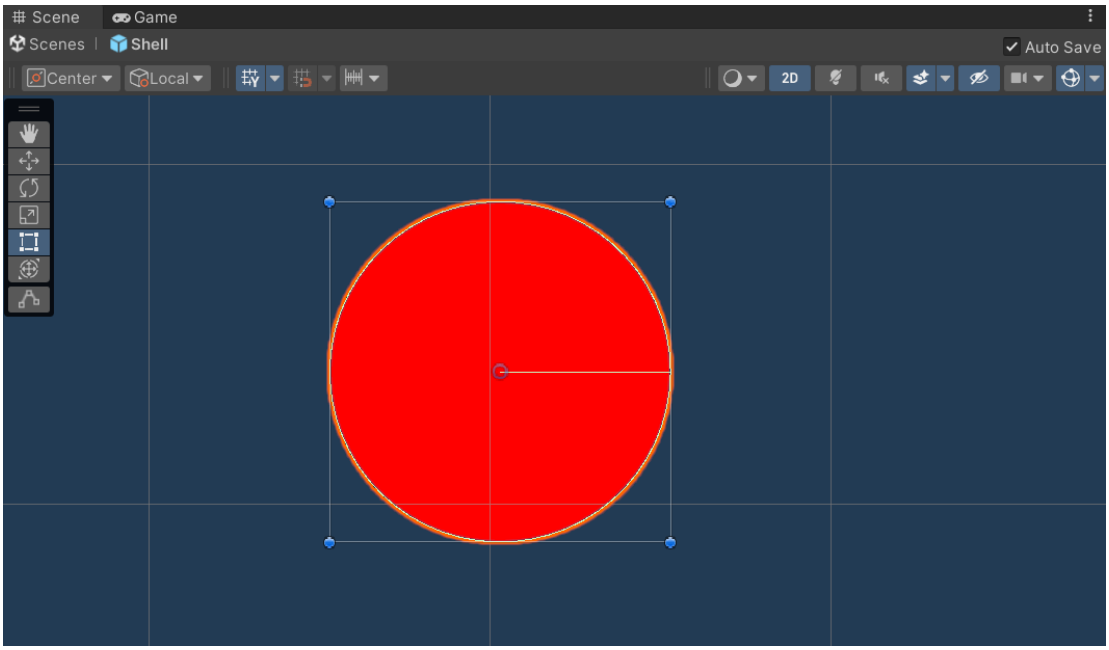
Typically in games you will use a combination of checking tags along with object names when writing code. For this project we will be using a combination of the two.

Shell Physics

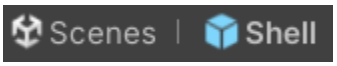
1. Now that we have our shell prefab made and collisions squared away, let's get to work on making the shell act like how it does in the game.

   If you remember from the gameplay video, the shell will spawn in from the top of the screen and shoot off in a random direction. When it hits a wall or the floor the shell bounces off that surface. The shell will not slow down and will continue to bounce until it is destroyed by the player.

2. Firstly, we need to make sure that we are in the **Shell Prefab scene**. All we need to do to access this scene is **double click on the shell prefab** that is in the assets folder of our project. Once we do, we will be in the appropriate scene.

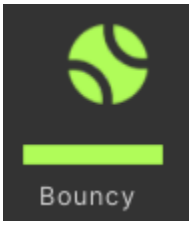


At the top left you should be able to see the name of the current scene. We can see that we are in the Shell scene. If we want to return to the main scene, we just click on the word **Scenes**.

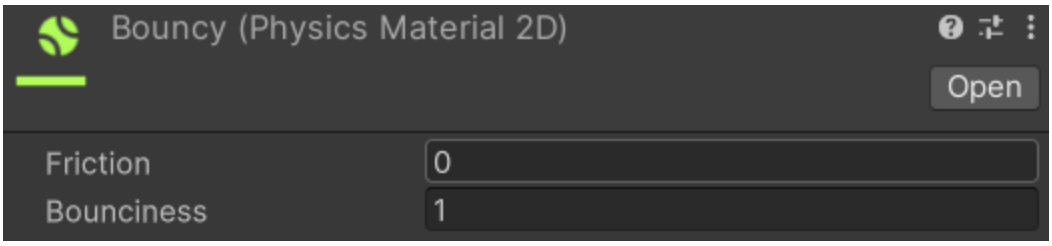


We want to be in the prefab scene to ensure that all instance of the shell will have the changes that we add to the object.

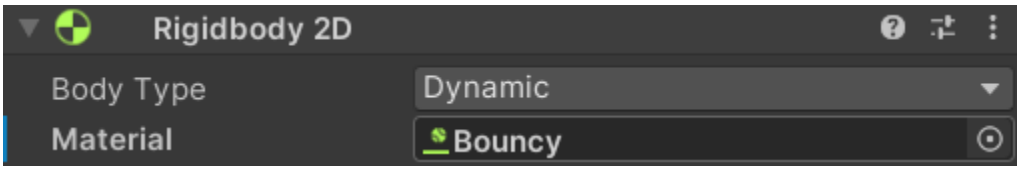
3. In order to make our shell bounce, we are going to add a Physics Material 2D to its rigidbody. Let's first create the material by right clicking the Assets folder and selecting **Create > 2D > Physics Material 2D**. Once created name the material **Bouncy**.



4. With the material selected we can see two properties, **Friction** and **Bounciness**.
Friction determines how fast the rigidbody will slow down when it hits something. Since we don't want our shell to slow down, we will leave this at a value of zero.
Bounciness controls how much energy is maintained when the object bounces off another object. Since we want our shell to bounce forever, we will set this to a value of one to ensure that the shell will always retain the same energy after each bounce.



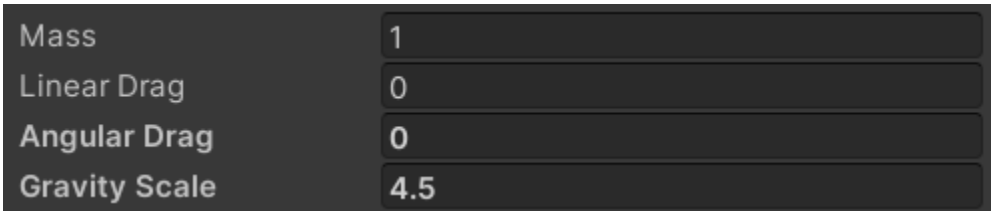
5. If we now go to the Shell's rigidbody component, we should be able to drag in the Physics Material and place it in the Material slot.



6. If we run the game now, we should be able to see the shell bounce in place. No matter what happens the shell will not slow down.

   We should notice to that the shell is moving pretty slowly when compared to its game counterpart. So, let's adjust some variables that will make the shell move a bit faster.

7. Still in the shell's rigidbody component, let's set the following properties to:



   Mass controls how heavy an body is.
   Linear Drag controls how fast on body will slow down when it is in the air or on the ground.
   Angular Drag is similar to line drag except it affects the rotational speed of the body.
   Gravity Scale is a multiplier that effects how strong gravity is for the body.

   We don't really need to change the preset options to much. We only set the angular drag to zero and the gravity scale to 4.5.

8. If we run the game now, we can see that the shell will fall much faster. However, there are a few issues. Depending on where the shell was dropped from, the shell will always bounce back to that height. Also, if we were to let the shell continue to bounce it would bounce higher and higher each time.

   The shells in Beetle Manion will always bounce to a specific height on the screen. So, to try and emulate this effect we are going to be adjusting the physics of the shell through code.

Faking Physics with Code

1. Let's open up the **Shell** script and add the variable:

```
private Rigidbody2D rb;
```

RB will hold a reference to the rigidbody on the shell.

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

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();

    rb.velocity = Vector2.right * Random.Range(-1f, 1f) * 20;
}
```

We first get the rigidbody component on the shell and store a reference to it in the rb variable.
We then determine the angle that the shell is thrown in. We do this by setting the velocity of the rigidbody to Vector2.right and multiplying it by a value between -1 and 1.
Vector2.right is the vector (1,0) so when we multiply it, we will get a vector between (-1,0) and (1,0). This determines if the shell will shoot to the left or right at the start of the game.
We then multiply it by 20 to add some force to shell to affect its starting angle. Making the starting vector of our shell's velocity somewhere between (-20, 0) and (20,0).

2. Next let's hop down to the **OnCollisionEnter2D()** function and write the code:

```
//Cap velocity when touching the ground.
if (collision.gameObject.name == "Bottom Wall")
{
    //Perserve x velocity but cap y velocity at 27.
    rb.velocity = new Vector3(rb.velocity.x, 27, 0);
}
```

We first check to see if the shell collided with the bottom wall. We have to make sure to **spell the name of the wall correctly,** or else the code inside would not trigger since the name is not match the string that we are comparing against.
If so, we then set the velocity of the shell to a new Vector3. We set the X value of the new vector to the rigidbody's existing x velocity. This will keep the shell moving in the same direction.
Then we set the Y value of the new vector to 27. This effectively replaces the rigidbody's y velocity. This ensures that the shell will always have the same y velocity when it bounces off the ground which will then make it always bounce to the same height.

We set the Z value of the new vector to zero. The shell does not need any z velocity for this project.

You may also be wondering why we didn't multiply the speed by Time.deltaTime or Time.fixedDeltaTime. We did this because we are not setting the velocity in the Update() nor the FixedUpdate() functions. This essentially makes the force we are setting the rigidbody to move or an impulse rather than a constant force over time.

3. Let's wrap up this section by writing a few more lines of code. Still in the **OnCollisionEnter2D()** function let's write the code:

```
if(collision.gameObject.name == "Left Wall" || collision.gameObject.name == "Right Wall")
{
    if (Mathf.Abs(rb.velocity.x) < 0.5f)
    {
        rb.velocity = new Vector2(-Mathf.Sign(rb.velocity.x) * 2, rb.velocity.y);
    }
}
```

This code is responsible for making the shell not get stuck at the edges of the screen. When building and testing the game for this lesson, sometimes the shell would hit a wall and then get stuck bouncing at the edge of the screen.

So what the code does is check to see if the shell hit the left or right wall. If so, it checks the absolute value of the velocity's x value and sees if it is less than 0.5. We are just checking to see if the shell hit the wall at a very slow horizontal speed. If that is the case, we give the shell a new horizontal speed of 2 or -2 depending on if the shell hit either the left or right wall.

4. Save the script and run the game. The shell should start off in a random direction. When the shell hits the bottom wall, it will then bounce up to a specific height with little variation.

Spawning

1. Next let's make it so more than one shell can appear in our game. To do this we are going to create a game logic object that will be responsible for keeping track of important game aspects such as score or number of turtles shells spawned.

2. First **remove** any shell prefab that is currently in our scene.

3. Next, in the hierarchy create a new **Empty** object and rename it to **Game Logic**.

4. Create a new script called **ShellSpawner** and then **attach** it to the **Game Logic** object.

5. In the script let's write the variables:

```
//Shell Spawning
public GameObject shell;
private float shell_spawn_time = 0.35f;
private float shell_spawn_timer = 0f;
private int max_shell_spawns = 30;
[HideInInspector] public int number_of_shells = 0;
```

Shell is the prefab shell that will be spawned into the game.
Shell_Spawn_Time is the time it takes for a new shell to spawn into the game.
Shell_Spawn_Timer keeps track of when a new shell will spawn in.
Max_Shell_Count is the maximum number of shells that can be in our game at once.
Number_Of_Shells is the current number of shells that are in the game.

In the **Start()** function we write the code:

```
void Start()
{
    //Wait a short amount of time before spawning shells.
    shell_spawn_timer = 1f;
}
```

We just set the timer to 1 to add a short delay before shells can start spawning into the game.

In the **Update()** function we write:

```
void Update()
{
    //Check if a shell can spawn.
    if (number_of_shells < max_shell_spawns)
    {
        //If so count down the time.
        if (shell_spawn_timer > 0)
        {
            shell_spawn_timer -= Time.deltaTime;
        }
        else //When time is counted down, spawn a shell.
        {
            SpawnShell();
        }
    }
}
```

We first check to see if the number of shells is less than the max number of shells.
If so, we then check if the timer is above zero, if it is we subtract deltaTime from the timer.
If not we then spawn a shell.

6. Then lets also create the **SpawnShell()** function so we do not get any errors.

```
private void SpawnShell()
{

}
```

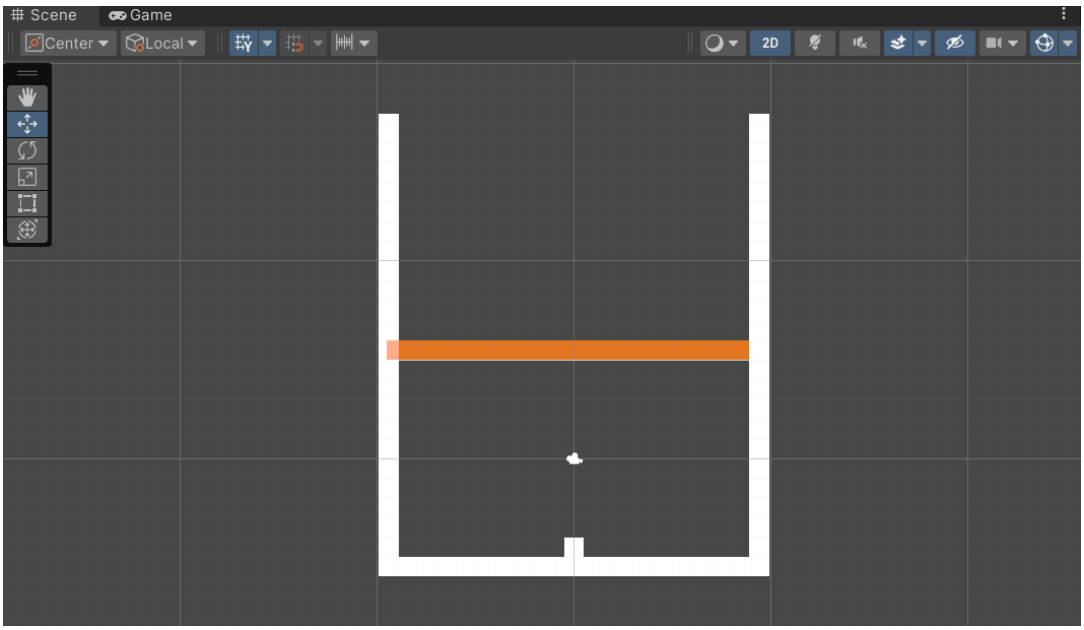
Then inside the function, let's write the code:

We then create the **SpawnShell()** function and write the code:

```
private void SpawnShell()
{
    //Instance a new shell.
    GameObject new_shell = Instantiate(shell);

    //Set position of shell to a random location.
    new_shell.transform.position = new Vector3(Random.Range(-7f, 7f), 6.5f);

    //Increase the shell count.
    number_of_shells += 1;

    //Reset Spawn Timer
    shell_spawn_timer = shell_spawn_time;
}
```

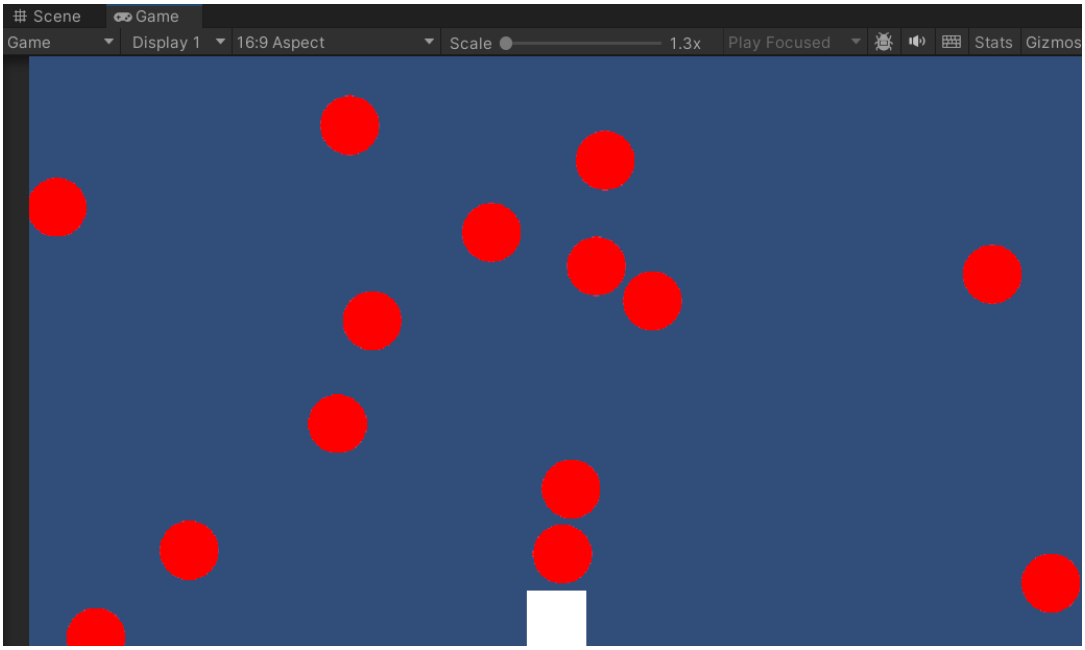
The code above should look very similar to the code we used for spawning a player bullet. The only thing that is really different is that we are setting the location of the shell to a random position. The horizontal position will be between -7 and 7 while the vertical position will be at 6.5. All this does is move the instantiated shell to above the view of the camera.

7. Save the script and return to unity. As a precaution for when the shells spawn in, let's raise the walls of our level.



We do this to ensure that any shells that get spawned in will be trapped inside the level.

8. If we run the game now, we should see that the shells will spawn in quick succession at various points at the top of the screen.



However there will be one issue. When the shell spawner spawns the maximum number of shells, it will no longer spawn any more even if the player has shot and destroyed a shell.

Bullet Collisions

1. So now that multiple shells can spawn into our game, we next need to make it so the player is able to shoot and destroy them. As well as have them be subtracted from the overall shell count in the game.

   Let's go into the **Shell** script and add the variable:

```
//Give reference to the spawner.
[HideInInspector] public ShellSpawner spawner;
```

   Spawner will hold a reference to the shell spawner script on the Game Logic object.

2. Then in the **OnTriggerEnter2D()** function lets write the code:

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

    //Destory ourselves.
    Destroy(gameObject);
}
```

We check to see if the object we collided with has the tag of bullet. **Be sure to spell the tag correctly** or else the if statements code will not run.
If it does, we destroy the object we collided with, the **collision.gameObject** being our bullet instance. Then we destroy the shell, that's the **gameObject** that the script is attached to.

3. Lastly lets add the **OnDestroy()** function and then write the code:

```
private void OnDestroy()
{
    spawner.number_of_shells -= 1;
}
```

When the shell is destroyed it will tell the spawner to reduce the number of shells. This is similar to how bullets work for the player.

4. Lastly, switch to the **ShellSpawner** script, and at the bottom of the **SpawnShell()** function add the line:

```
//Give reference to spawner.
new_shell.GetComponent<Shell>().spawner = this;
```

When the spawner spawns in a shell, that shell will now have a reference to the spawner.

5. If we save the script and run our game, when a shell collides with a bullet, both the shell and the bullet will disappear from the game.

   If the maximum number of shells has been reached and the player destroys a shell, the shell spawner should be able to start spawning shells again.

   However, we still need to make it so that when the collision occurs between the two objects, the shell will spawn five additional bullets that will shoot out in a random direction.

Spawning Bullets from Shells

1. Firstly, lets add a couple variables to our **Shell** script:

```
//Flag that the shell has been hit.
private bool was_hit = false;
```

```
//Bullet
public GameObject bullet;
```

Was_Hit is a Boolean that we will use to tell the shell that it has been hit by a bullet. We are going to use this variable to avoid the shell spawning multiple bullets if the shell happed to collided with more than one bullet on the same frame.
Bullet is the bullet game object that the shell will spawn five of when hit.

2. Let's go back to the **OnTriggerEnter2D()** function and change the if statement to:

```
if (collision.gameObject.tag == "Bullet" && !was_hit)
{
```

Lets also write one line of code inside and at the top of the if statement that will set the was_hit variable to true.

```
was_hit = true;
```

Now whenever the shell collides with a bullet, it will know that it was hit and will not respond to any more collisions with any other bullets.

3. Before we write the code for spawning the five bullets, let's first think back to how our bullets work. Looking at the start function in the **Bullet** script we see:

```
void Start()
{
    //Get Components
    rb = GetComponent<Rigidbody2D>();

    //Set starting velocity to make the bullet shoot upwards.
    rb.velocity = Vector3.up * 500 * Time.fixedDeltaTime;
}
```

This has been working for us so far, but if the shell were to instantiate a bullet and give it a random direction to travel in, the code in the start function would run one frame later. This would then overwrite the direction of the bullet to Vector3.up. This is a good example of

Unity's **execution order** coming into play.

So instead, let's make a function that will set the bullet up and tell the bullet what direction to travel in.

In the **Bullet** script, we can copy the code in the **Start()** function and the delete it to leave the function empty.

```
void Start()
{

}
```

Then let's create a new function called **SetUp()**. We can paste the code from the start function inside and then change to match the code below:

```
public void SetUp(Vector3 dir)
{
    //Sets up the bullet for use in the game.

    //Dir - Initial direction that the bullet will travel in.

    //Give reference to the Rigidbody2D.
    rb = GetComponent<Rigidbody2D>();

    //Set the velocity of the bullet.
    rb.velocity = dir * 500 * Time.fixedDeltaTime;
}
```

The function SetUp() takes in one argument. This is the direction that the bullet will travel in. Like before in the Start() function, we give the bullet a reference to the bullet's rigidbody. We then set its velocity. We make sure to **replace** the **Vector3.up with dir** since this will be the direction we want the bullet to travel in.

I've also left a couple comments in function. The first is to note what the functions purpose is and the second is to describe what each argument of the function is used for.

4. One quick side note, since we are in the bullet script we should change the **OnDestroy()** function's code to:

```
private void OnDestroy()
{
    if(player != null)
    {
        player.bullet_count -= 1;
    }
}
```

Since the shell object will now be able to set the player property on the bullet when it spawns the five bullets, the player value will be null. So, we are going to add a check to see if the player variable is not null. This will ensure that only the bullets fired by the player will reduce the player's bullet count when destroyed. Since shells can now fire bullets, we do not want them reducing the bullet count variable on the player.

5. Let's get back to the **OnTriggerEnter2D()** function in the **Shell** script and underneath the code where we are destroying the bullet, lets write:

```
//Spawn five bullets.
for (var i = 0; i < 5; i++)
{
    //Instance a new bullet.
    GameObject new_bullet = Instantiate(bullet);

    //Move the bullet to shells position.
    new_bullet.transform.position = transform.position;

    //Set up the bullet.
    Vector3 bullet_dir = new Vector3(Random.Range(-1f, 1f), Random.Range(-1f, 1f), 0).normalized;
    new_bullet.GetComponent<Bullet>().SetUp(bullet_dir);

    //Destroy the bullet after a short period of time.
    Destroy(new_bullet.gameObject, 0.25f);
}
```

We start by writing a for loop that will only loop through five times.
We then instantiate a bullet game object.
We set it's position to the position of the shell.
We create a variable that will store the direction vector of the bullet. The vector will have a random number between -1 and 1 for the X and Y value. Z is 0 since we do not want the bullet to travel in that axis.
We then call the bullet's set up function and pass in the direction.
Lastly, we well the bullet to destroy itself when 0.25 seconds have passed.

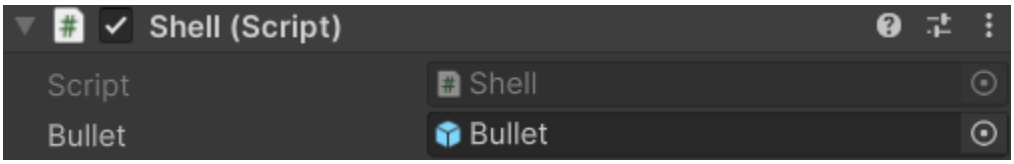
6. Let's next hop over to the **Player** script, in the **SpawnBullet()** function, under where we instantiate the new bullet, we write the code:

```
//Set Up bullet.
new_bullet.GetComponent<Bullet>().SetUp(Vector3.up);
```

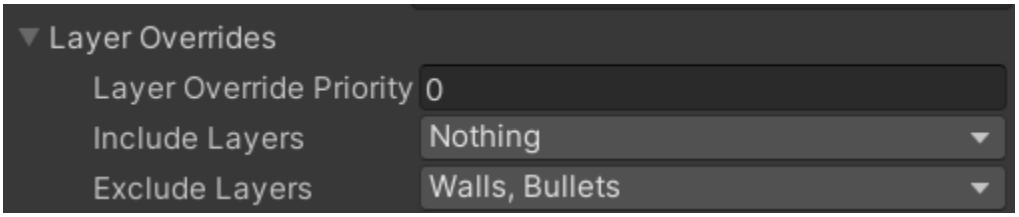
Since the bullet does not fire up by default, we need to make sure to tell the player to set the direction of the bullet to up when it is first fired.

7. Save all the script and return to Unity. We then just need to make a few changes to a couple objects.

   First, we give the shell a reference to the bullet object so it can instantiate them,

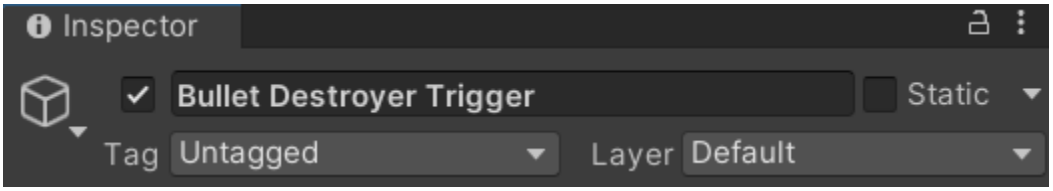


   Next on the **Bullet Prefab** we change the layer overrides on the **Circle collider 2D** to:



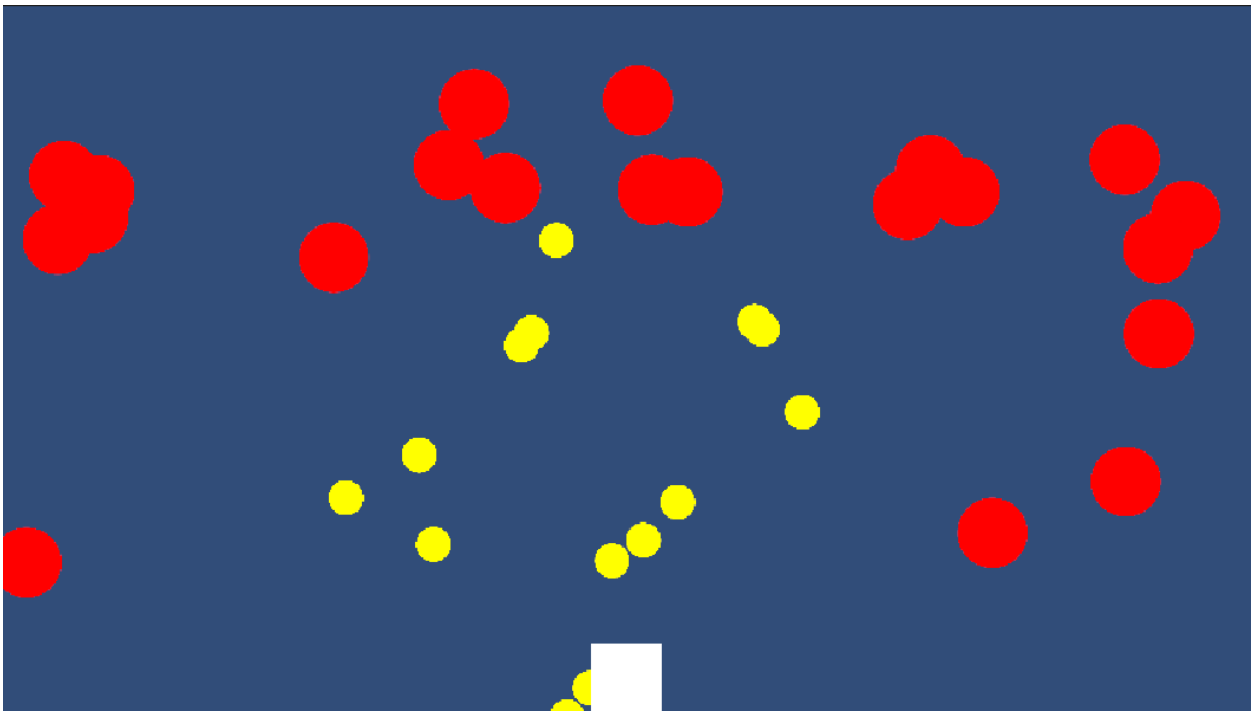
   This ensures that the bullet will be able to pass through walls and other bullets.

   We also need to make sure that the bullet destroyer trigger is on the **Default** layer. We do not want it to be on the wall layer since bullets now ignore walls.



8. With that all done, if we run the game, when a bullet collides with a shell it will spawn five bullets. If these bullets collide with other shells, they will also spawn bullets. You should be

able to chain hitting multiple shells this way.



GitHub

1. Push the project to the GitHub repository.