

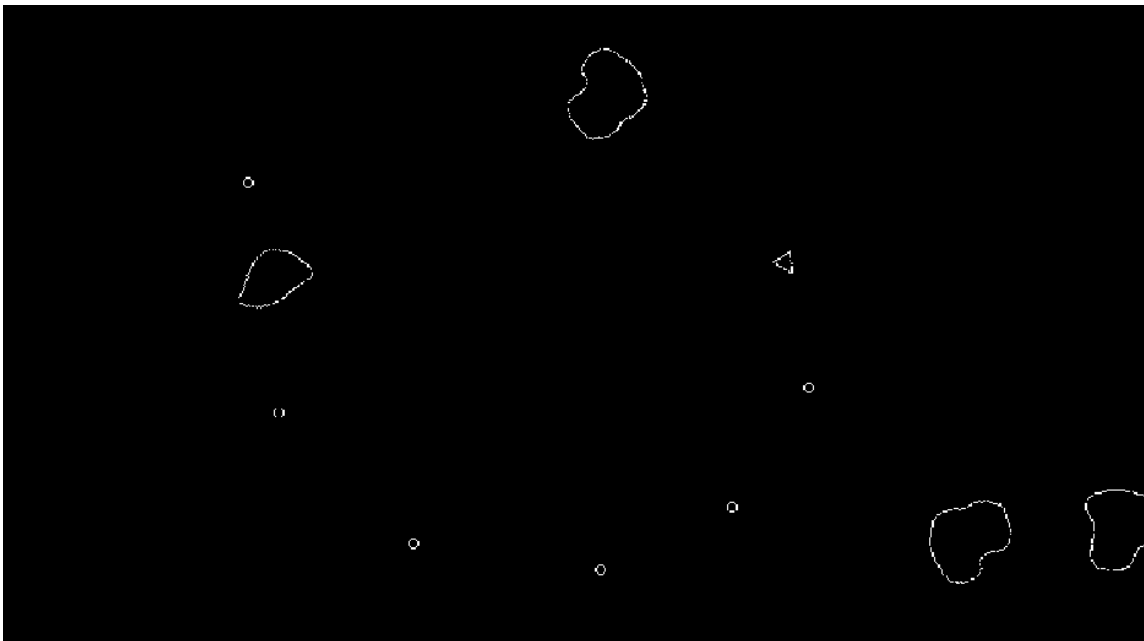
Asteroids: Part 2 – Asteroids

Lesson Goals

1. Create the asteroid game object.
2. Learn how to convert an angle into a direction vector.
3. Learn the inheritance programming pattern and how it allows for the reuse of code.
4. Create a player bullet and allow the player to shoot it.

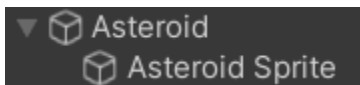
Demo

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

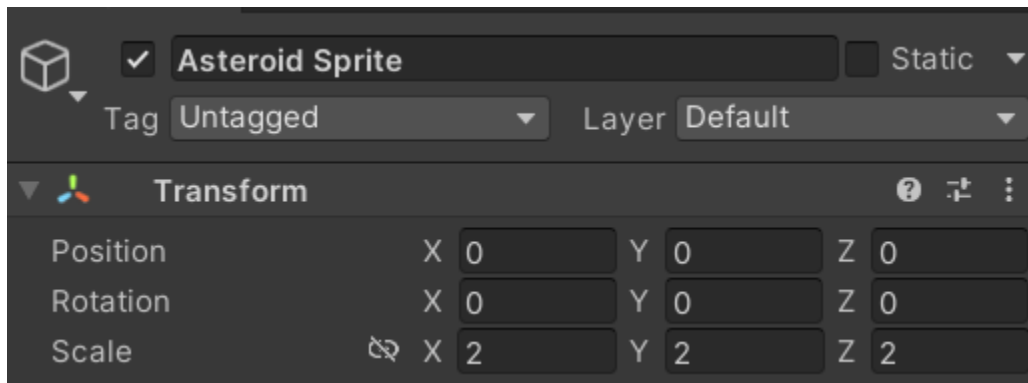


Asteroid Game Object

1. Let's start this lesson off by creating the asteroid game object. In the **hierarchy** create a new **Empty** game object and name it **Asteroid**. Then drag in one of the **large** asteroid sprites and make it a child of the asteroid game object. Name the sprite **Asteroid Sprite**.



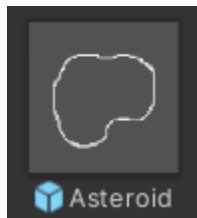
Set the asteroid sprite's **position** to zero for the **X**, **Y** and **Z**. then set the **scale** of the object to **2** for the **X**, **Y** and **Z**.



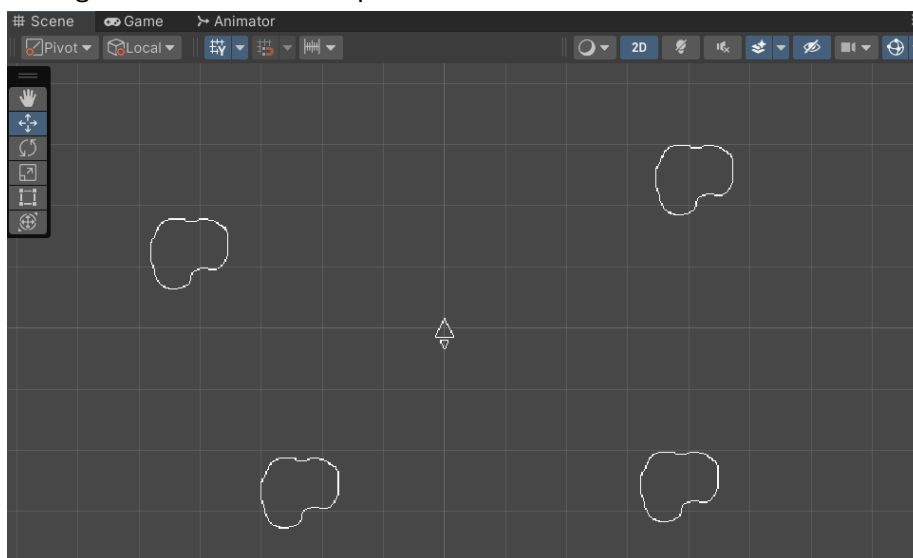
- Next create a new **Script** called **Asteroid**, then attach it to the asteroid game object. **Make sure the script is not attached to the sprite.**



- Lastly, drag the asteroid into the assets fold to turn it into a prefab.



Then go ahead and add a couple asteroids into the scene.



Asteroid Movement

1. In the **Asteroid** script, lets create the variables:

```
//Movement
private Vector3 direction = Vector3.zero;
private float max_speed = 0f;
private Vector3 velocity = Vector3.zero;
```

Direction is the direction that the asteroid will travel in.

Max_Speed is how fast the asteroid will travel.

Velocity will be used to tell the asteroid to travel in a direction at a certain speed.

These variables shouldn't be any surprise. They are the same ones we are using for the ship. Keep the fact that these are the same variables in mind.

2. Next in the **Start()** function let's write the code:

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

We first start off by setting the max_speed of the asteroid to a random value between 0.65 and 1.1. This will just add some small variance between each asteroid that is in our game.

3. Underneath let's write the code:

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

We start by declaring a variable called start_angle. This variable is going to determine the direction that our asteroid will travel in. We pick random angle between 0 and 360.

As a quick aside let's visualize how angles work in unity. We will use the z axis to rotate along and we will use the player to show how rotation affect it.

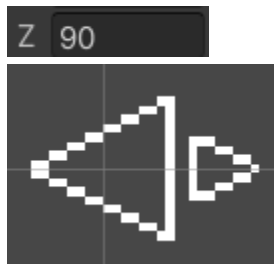
By default, **0** points an object **up**.



180 points an object **down**.



90 to the **left**.



270 to the **right**.



It is really easy for humans to understand and visualize the relationship between angles and directions. However, we have been using vectors to represent directions which are a little less clear for us to visualize.

Luckily, we can use some trigonometry to turn an **angle** into the components for a direction **vector**.

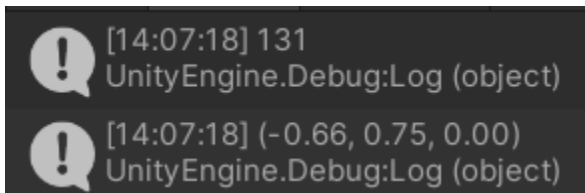
4. Back in the **Start()** function lets write the code:

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

In order to convert an angle into the components for the direction vector we first need to take the angle and convert it into radians. We can do this by multiplying an angle by **Mathf.Deg2Rad**. If we ever needed to go from a radian to a degree, we can use **Mathf.Rad2Deg**.

Then in order to get the **X** component of the direction value we just take the **Cosine** of the angle. For the **Y** component we take the **Sine**. I'm not going to go into the math behind these conversions, quite frankly I don't know them myself. So, for right now, it just works.

5. If we were to output angle and the direction vector, we might get a value like:



```
[14:07:18] 131
UnityEngine.Debug:Log (object)
[14:07:18] (-0.66, 0.75, 0.00)
UnityEngine.Debug:Log (object)
```

The first number shows the starting angle in degrees.

Right below it we can see that angle converted into a direction vector.

The nice thing too is that our direction vector is already normalized!

6. Alright, with that aside, the last piece of code we write in our **Start()** function is:

```
//Calculate the velocity.
velocity = direction * max_speed;
```

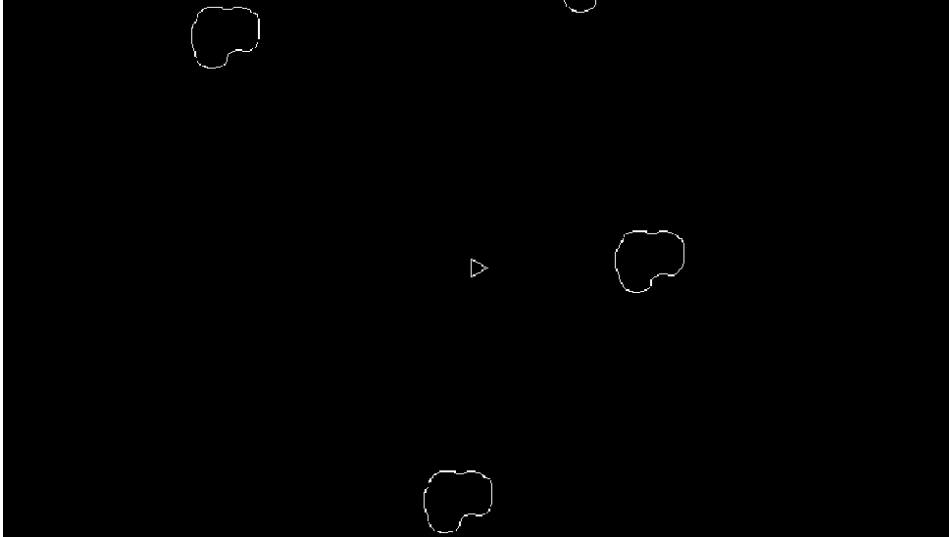
We simply just set the velocity.

7. Lastly, in the **Update** function, let's write the code:

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

All we are doing is moving the player.

8. Save the script and return to the game. If the game is run, we can see each of the asteroids move in a different direction.



One thing you may be wondering too, why we are not using the up direction of the asteroid to move. And truthfully, we totally could if we wanted to. We could rotate the asteroid, like we do with the player, and then use that as the direction. But in this example, I chose to separate the direction the asteroid is facing with the direction that it is traveling. There are multiple ways to approach a problem, each way has its own pros and cons.

Asteroid Graphics

1. Our asteroids can move, but they all look the same, plus it is pretty boring for them to be a still image moving. So, let's fix this by allowing the asteroid to choose what asteroid sprite it will be as well as make the asteroid sprite spin in place.

In the **Asteroids** script lets add the variables:

```
//Graphics
public GameObject asteroid_sprite;
public Sprite[] array_of_sprites;
private float spin_speed = 0f;
```

Asteroid_Sprite will hold a reference to the child sprite game object that the asteroid has.

Array_Of_Sprites will hold all of the sprites that the asteroid can choose from.

Spin_Speed is how fast the asteroid sprite will be able to rotate.

2. In the **Start()** function, lets add the code:

```
//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)];
```

First we get the sprite renderer component on the asteroid sprite and then access its sprite property. We then set the property equal to one of the sprites in the array.

Remember that an array has multiple values that can be accessed. To access these values, we write the name of the array followed by the position of the value we wish to get. In our example we are getting a random position in the array by saying [Random.Range(0, array_of_sprites.Length - 1)].

Hopefully that all makes sense but in total we are just getting a random sprite from our array.

3. Underneath lets write the code:

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

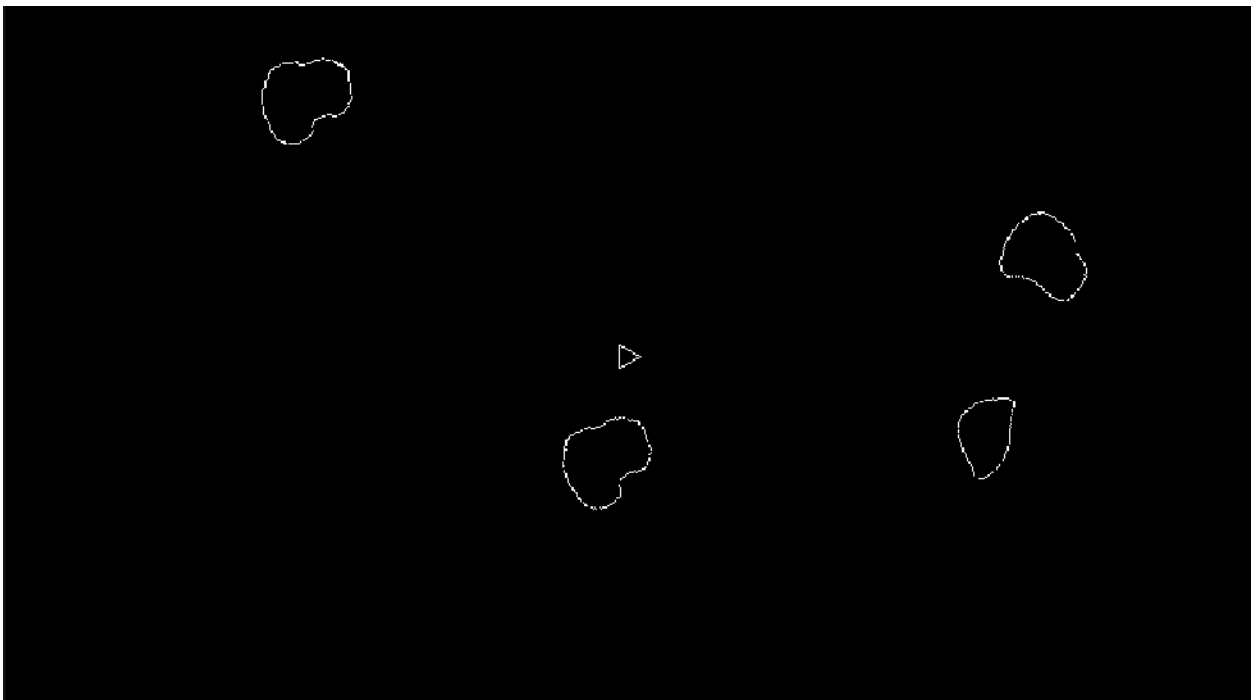
Here we just set the spin speed to a random value between -20 and 20.

4. Lastly in the **Update()** function lets write the code:

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

Similar to how we rotate the player, we rotate the asteroid along the forward vector by the spin_speed * Time.deltaTime.

5. If we save the script and run the game, the asteroids will now pick a random sprite and will spin.



Screen Wrapping

1. Our asteroids are looking pretty good by the are able to leave the bounds of the level, so let's add screen wrapping to them.

In the **Asteroid** script lets add the variables:

```
//Screen Wrapping
public Camera cam;
private Vector2 cam_bottom_left = Vector2.zero;
private Vector2 cam_top_right = Vector2.zero;
private float wrap_radius = 0.64f;
```

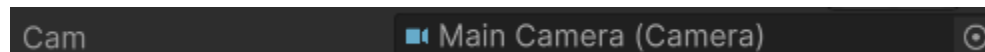
These are all the same variables that the player uses. Again, keep this fact in mind.

2. In our **Start()** function lets write the code:

```
//Find the camera in the scene.
cam = GameObject.Find("Main Camera").GetComponent<Camera>();
```

Instead of dragging in the camera and putting it in the cam slot on each of the asteroids, we are opting to have the script find the camera for us. We use the GameObject class's **Find** function to search for the Main Camera in our game. We then just grab the camera component from the Main Camera object.

Once our code is done and we run the game we should be able to see that the camera was properly found if we look at the inspector.



3. Next, we are going to head over to the **Player** script, grab all the code for calculating the camera's bounds and paste it in to the **Start()** function of the **Asteroid** script.

```
//Set camera vectors to screen space. (Pixels)
cam_top_right = new Vector3(cam.scaledPixelWidth, cam.scaledPixelHeight, 0);

//Convert from screen space in world space.
cam_bottom_left = cam.ScreenToWorldPoint(cam_bottom_left);
cam_top_right = cam.ScreenToWorldPoint(cam_top_right);
```

Again more code from the player.

4. Again, we can head over to the **Player** script and then copy the code for screen wrapping and then paste it at the bottom of the **Update()** function.


```
//Screen Wrap
//Right Side
if (transform.position.x - wrap_radius > cam_top_right.x)
    transform.position = new Vector3(cam_bottom_left.x - wrap_radius + 0.01f, transform.position.y, 0);

//Left Side
if (transform.position.x + wrap_radius < cam_bottom_left.x)
    transform.position = new Vector3(cam_top_right.x + wrap_radius - 0.01f, transform.position.y, 0);

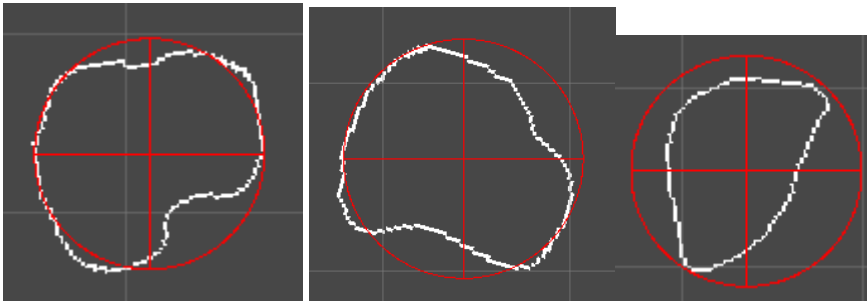
//Top Side
if (transform.position.y - wrap_radius > cam_top_right.y)
    transform.position = new Vector3(transform.position.x, cam_bottom_left.y - wrap_radius + 0.01f, 0);

//Bottom Side
if (transform.position.y + wrap_radius < cam_bottom_left.y)
    transform.position = new Vector3(transform.position.x, cam_top_right.y + wrap_radius - 0.01f, 0);
```

This is getting out of hand!

5. If we save the script and run the game, the asteroids should now be able to wrap around the screen.

One thing to keep in mind is that the screen wrap for each asteroid is a one size fits all solution. This radius might not be the best to represent each asteroid.



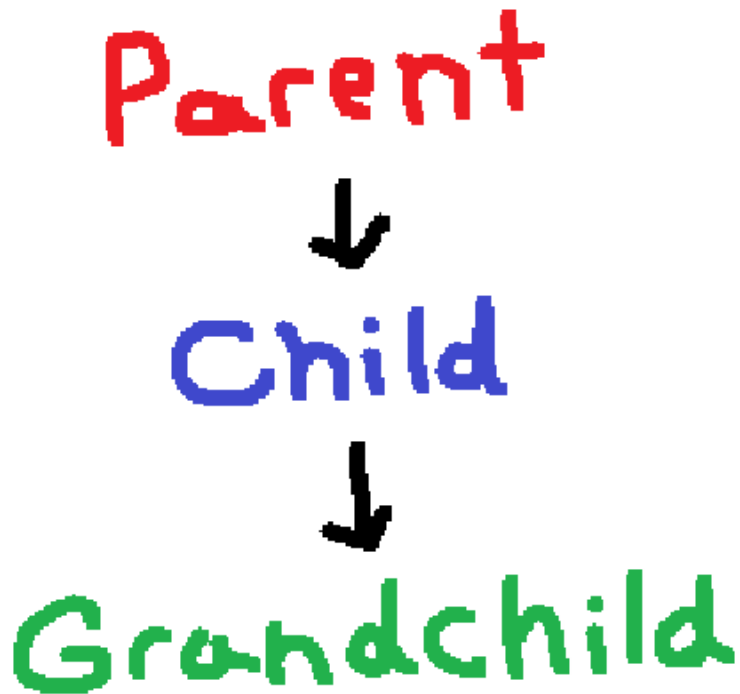
Again we can use the **OnDrawGizmos()** function to visualize the radius around are asteroid.

If we wanted to make our asteroids have a more accurate radius, we could set up a switch statement that looks at which sprite the asteroids has. And then based on that a proper radius could be set.

Inheritance

1. So, you should hopefully be realizing that we are using quite a few bits of the player's code for the asteroid. It's pretty annoying to copy and paste the same code over and over again. We should be able to write a piece of code once and then use that piece of code for multiple things. One way we can achieve this is to use of **Inheritance**.

To put it simply, **Inheritance** allows us to reuse code from a base class for subclasses. A base class is called a **Parent** and a subclass is called a **Child**.



The image above shows the flow of code from parent to child. The parent class at the top gives code to the child and that child class gives it code to its child grandchild.

2. To make better sense of this let's look at a more practical example.

Animal
-Species

Dog
-Species - Name
- Bark()

Puppy
-Species - Name - Toy
- Bark()

Our parent class is called Animal, and it has the variable species.

We then have a child class called Dog. It **inherits** the species variable from the parent class but also has a variable called name and a function for barking.

Then we have the grandchild class called Puppy. It has the species variable from the base parent class, it has the name variable and bark function from its parent class and then it has a variable of its own called Toy.

So in total the Puppy class has all the functionality from the Animal class and the Dog class as well as whatever functionality the Puppy class would have. The Animal class however only has the functionality of the Animal class. It does not have the functionality of its child or grandchild classes.

3. Hopefully with the previous example you can start to see how we may apply inheritance to our game. There is a lot more we can do with inheritance, but we will just cover the basics for right now.

Space Object Class

1. Back in our game, lets create a new class called **SpaceObject**.



This will be the parent class that both the Player and Asteroid classes will inherit from.

2. Let's open the script, and then let's copy all the **Moving** and **Screen Wrapping Variables** from either the Player or Asteroid script, and then paste them into the SpaceObject script.

```
//Moving
private float max_speed = 5.5f;
private float acceleration = 13f;
private Vector3 velocity = Vector3.zero;

//Screen Wrapping
public Camera cam;
private Vector2 cam_bottom_left = Vector2.zero;
private Vector2 cam_top_right = Vector2.zero;
private float wrap_radius = 0.16f;
```

I've gone ahead and taken the variables from the Player script.

Let's then remove a couple variables:

```
//Moving
private float max_speed = 5.5f;
private Vector3 velocity = Vector3.zero;

//Screen Wrapping
private Vector2 cam_bottom_left = Vector2.zero;
private Vector2 cam_top_right = Vector2.zero;
private float wrap_radius = 0.16f;
```

First, I removed the **acceleration** variable. It is a variable that only the player would need. If you copied from the asteroid script, be sure to **remove the direction variable**.

Second, I removed the **cam** variable. While writing this lesson I realized that this variable only gets used in a specific place in our script. So instead of having the script have full access to this variable, we are going to change it so that only one function will have access to it. When we do something like this the variable becomes something called a **Local Variable** since it is local to just that function.

3. Next since these variables are going to be used for the Player and The Asteroid scripts, we should give them generic values. For this case, I am just going to assign each value to zero.

```
//Moving
private float max_speed = 0f;
private Vector3 velocity = Vector3.zero;

//Screen Wrapping
private Vector2 cam_bottom_left = Vector2.zero;
private Vector2 cam_top_right = Vector2.zero;
private float wrap_radius = 0.0f;
```

4. Next, we are going to change the protection level of some the variables to **Protected**.

```
//Moving
protected float max_speed = 0f;
protected Vector3 velocity = Vector3.zero;

//Screen Wrapping
protected Vector2 cam_bottom_left = Vector2.zero;
protected Vector2 cam_top_right = Vector2.zero;
protected float wrap_radius = 0.0f;
```

As a quick recap, protection levels determine the accessibility level of a variable or function. Public means the data can be accessed by other scripts. Private means the data can **only** be accessed by the script that defines the data.

Protected can be thought up as a mix of the two. Other scripts cannot access the protected data, however scripts that inherit from the class can.

5. Next let's create a new function called **FindCameraBounds()**.

```
protected void FindCameraBounds()
{
    //Find the left, right, top and bottom side of the camera in world space.
}
```

6. Then let's go to the **Asteroid** script and copy the code that sets up the variables for screen wrapping and then paste it in the **FindCameraBounds()** function.

```
/**Screen Wrapping**
//Find the camera in the scene.
cam = GameObject.Find("Main Camera").GetComponent<Camera>();

//Set camera vectors to screen space. (Pixels)
cam_top_right = new Vector3(cam.scaledPixelWidth, cam.scaledPixelHeight, 0);

//Convert from screen space in world space.
cam_bottom_left = cam.ScreenToWorldPoint(cam_bottom_left);
cam_top_right = cam.ScreenToWorldPoint(cam_top_right);
```

I wanted to use the code in the asteroid since we are using that nifty find function.

We are going to get an error since we don't have the cam variable anymore. To fix this we just need to change the top line to:

```
Camera cam = GameObject.Find("Main Camera").GetComponent<Camera>();
```

Again we are turning the cam variable into a **Local** variable since it is only need inside of this

function.

7. Let's go ahead and create another function called **ScreenWrap()**.

```
protected void ScreenWrap()  
{  
    ...  
}
```

Then we can just copy the screen wrap code from either the player or asteroid script and paste it inside.

```
//Right Side  
if (transform.position.x - wrap_radius > cam_top_right.x)  
    transform.position = new Vector3(cam_bottom_left.x - wrap_radius + 0.01f, transform.position.y, 0);  
  
//Left Side  
if (transform.position.x + wrap_radius < cam_bottom_left.x)  
    transform.position = new Vector3(cam_top_right.x + wrap_radius - 0.01f, transform.position.y, 0);  
  
//Top Side  
if (transform.position.y - wrap_radius > cam_top_right.y)  
    transform.position = new Vector3(transform.position.x, cam_bottom_left.y - wrap_radius + 0.01f, 0);  
  
//Bottom Side  
if (transform.position.y + wrap_radius < cam_bottom_left.y)  
    transform.position = new Vector3(transform.position.x, cam_top_right.y + wrap_radius - 0.01f, 0);
```

8. Lastly, just save the script.

Player and Asteroid Clean Up

1. Now that we have our SpaceObject class made, we need to have both our Player and Asteroid classes inherit from them. Starting with the **PlayerScript** lets change the class definition code to:

```
public class Player : SpaceObject
```

Instead of inheriting from MonoBehaviour, we no inherit from SpaceObject. Keep in mind, since space object still inherits from MonoBehaviour the player script does too. This is why we can still use the Start() and Update() functions.

2. Once we inherited from SpaceObject, you should have noticed that a few of the variables now have green underlines on them.

```
//Moving
private float max_speed = 5.5f;
private float acceleration = 13f;
private Vector3 velocity = Vector3.zero;

//Screen Wrapping
public Camera cam;
private Vector2 cam_bottom_left = Vector2.zero;
private Vector2 cam_top_right = Vector2.zero;
private float wrap_radius = 0.16f;
```

These are all the variables that are in the SpaceObject class. So, let's go ahead and remove them. We also need to be sure to remove all of the variables for screen wrapping.

We should only have the acceleration variable by the end.

```
//Moving
private float acceleration = 13f;
```

- Let's jump down to the **Start()** function. At the top let's add the code:

```
//Initialize Variables
max_speed = 5.5f;
wrap_radius = 0.16f;
```

We need to overwrite both of the values for these variables. The default value is 0.0 since that is what is defined in the SpaceObject class. We set them back to their original values to ensure that the player can move and screen wrap properly.

- Next remove the code for the camera and in its place let's write the code:

```
//Find Camera Bounds
FindCameraBounds();
```

Remember that since this function is **protected** the child classes are able to call it.

- In the **Update()** function let's find the code for screen wrapping, remove all of it and then in its place we can write:

```
//Screen Wrap
ScreenWrap();
```

6. That should then do it for the player, lets hop over to the **Asteroid** script. Like before, lets make the script inherit from SpaceObject.

```
public class Asteroid : SpaceObject
```

7. Let's then remove any variables that have a green underline as well as all of the variables for screen wrapping. These should be the only variables for the asteroid:

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

//Graphics
public GameObject asteroid_sprite;
public Sprite[] array_of_sprites;
private float spin_speed = 0f;
```

8. In the **Start()** function we can remove the code for screen wrapping and then replace it with:

```
//Find Camera Bounds
FindCameraBounds();
```

Then just underneath we can write the code:

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

9. Then in the **Update()** function we can do the same thing, except replace the code with:

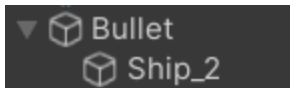
```
//Screen Wrap
ScreenWrap();
```

10. Save all the scripts and then return to Unity. If we now run the game everything should work just as it was.

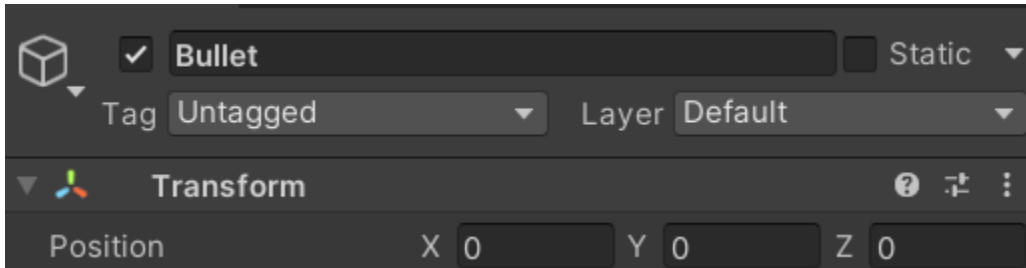
Bullet

1. Let's next get to work on making our ship shoot. We are first going to create the bullet game object.

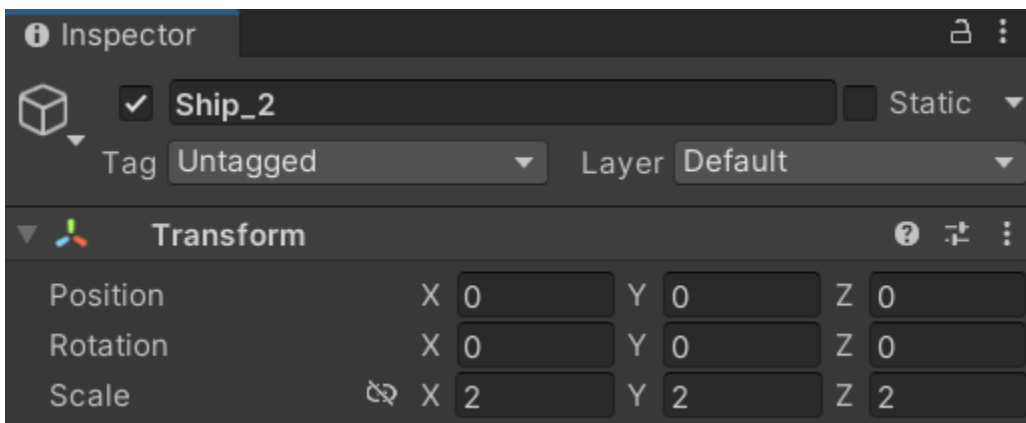
First let's create our bullet game object. We create a new **Empty** object in our scene and **name** it **Bullet**. We are then going to drag in the third sprite of the ship sprite sheet and make it a child of the bullet object.



Zero out the position of the bullet object.



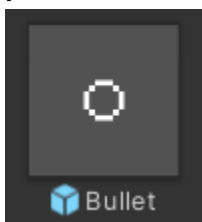
And then zero out the position of the bullet sprite. Also set the scale of the sprite to 2 for the X, Y and Z.



2. Next create a script called **Bullet** and then attach it to the Bullet game object.



3. Next drag the Bullet game object from the hierarchy into the asset folder to turn it into a **prefab**.



We can then go ahead and remove the Bullet that is in the scene.

Shooting

1. Next let's make our ship shoot. A of what we are going to has been covered in the previous project, so we won't go into too much detail.

Open the Bullet script and make the class inherit from **SpaceObject**.

```
public class Bullet : SpaceObject
```

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

```
FindCameraBounds();
```

3. In the **Update()** function let's first write:

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

4. We need to keep in mind that our bullet isn't going to screen wrap, although that would make for a fun twist on the asteroid formular, instead the bullet needs to be destroyed as soon as it goes outside the bounds of the camera. So to do this we can modify the screen wrap code.

Underneath where the bullet is moving let's write the code:

```
//Have bullet destroyed if it leaves the camera's view.
if (transform.position.x > cam_top_right.x)
{
    Destroy(gameObject);
}
if (transform.position.y > cam_top_right.y)
{
    Destroy(gameObject);
}
if (transform.position.x < cam_bottom_left.x)
{
    Destroy(gameObject);
}
if (transform.position.y < cam_bottom_left.y)
{
    Destroy(gameObject);
}
```

5. Lastly let's create a new function called **SetUp**.

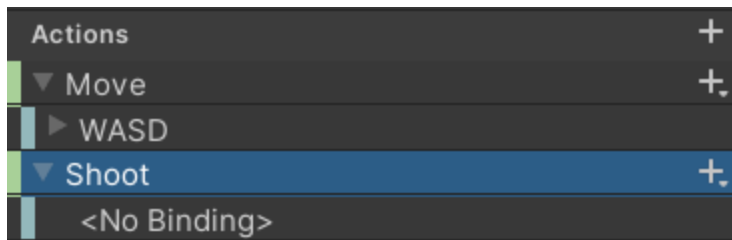
```
public void SetUp(Vector3 direction)
{
    //Set how fast the bullet will travel.
    max_speed = 8f;

    //Set the velocity.
    velocity = max_speed * direction;
}
```

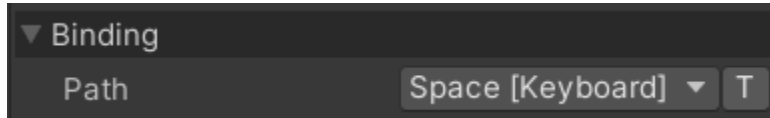
This function takes in a direction vector. It then multiplies this vector by the speed and sets it as the velocity.

This function is going to be called when the player initialized the bullet. If you remember from the last lesson, we are initialize our variables here as to avoid issues with Unity's order of execution.

6. Speaking of the player, save the script and return to Unity and open the **Player Input Action**. Let's create a new action for shooting. Click the **plus sign** to the right of actions and name it **Shoot**.



Click where it says **<No Binding>** and then over on the right set the **Path** property to **Space[Keyboard]**.



7. Close out of the Input Action window and then open up the **Player** script. At the top of the script, let's add the variables:

```
//Shooting
public GameObject bullet;
private float rate_of_fire = 0.1f;
private float shoot_timer = 0f;
```

8. Next lets create the function:

```
public void CaptureShootInput(InputAction.CallbackContext context)
{
    ...
}
```

Inside, lets write the code:

```

if(shoot_timer <= 0)
{
    //Reset timer.
    shoot_timer = rate_of_fire;

    //Instance bullet.
    GameObject new_bullet = Instantiate(bullet);

    //Set velocity of bullet.
    new_bullet.GetComponent<Bullet>().SetVelocity(transform.up);

    //Position bullet at player.
    new_bullet.transform.position = transform.position;
}

```

I want to point out that we are using the `SetVelocity()` function on the bullet script. We are passing in the player's up direction to use as the direction of the bullet.

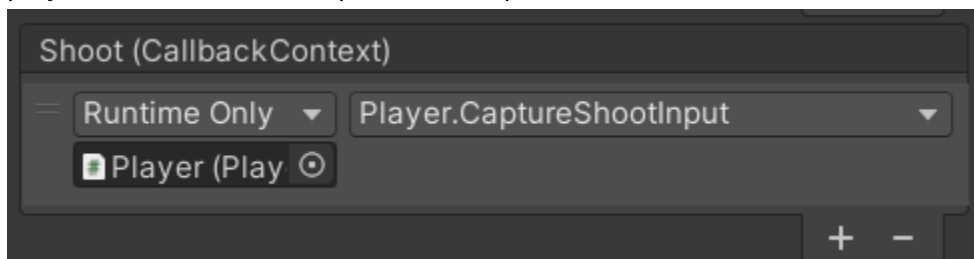
9. Then in the **Update()** function let's write the code:

```

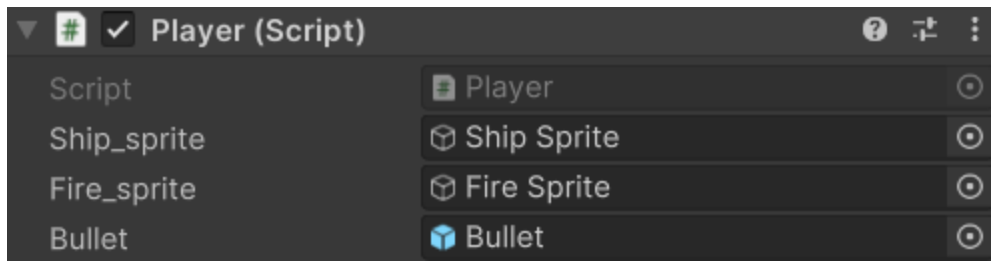
//Countdown the time to shoot.
if (shoot_timer > 0)
{
    shoot_timer -= Time.deltaTime;
}

```

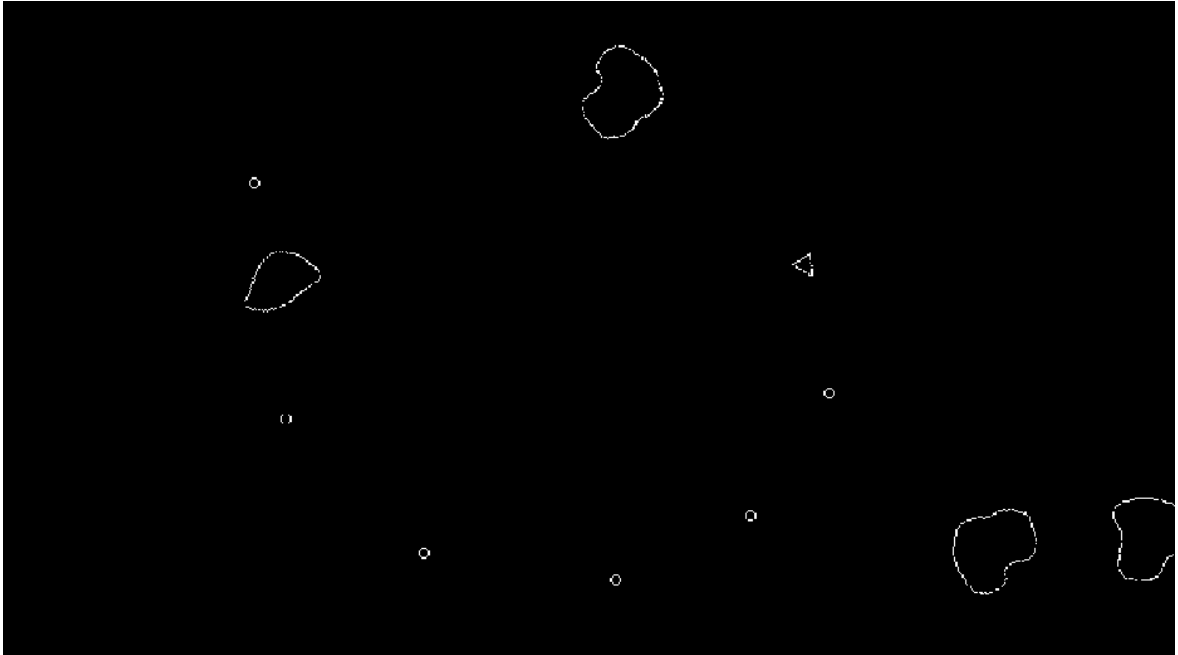
10. Save all the scripts and return to Unity. We need to go to the **Player Input** component on the player and connect the `CaptureShootInput` function to the Shoot action.



Then we need to drag the Bullet prefab and drop it into the Bullet slot on the player script component.



11. If we run the game the player should now be able to shoot!



GitHub

1. Push the project to the GitHub repository.