COMP4002/G54GAM Lab Exercise 01 – Vertical shmup (shoot-em-up)

05/01/19

This lab exercise involves creating a simple top-down scrolling space shooter as an introduction to various Unity principles and features, and doing some scripting in C#. This exercise is a cut-down version of one of the official Unity tutorials, which gives step-by-step instructions and explanation, and you may find it useful to refer to it you get stuck:

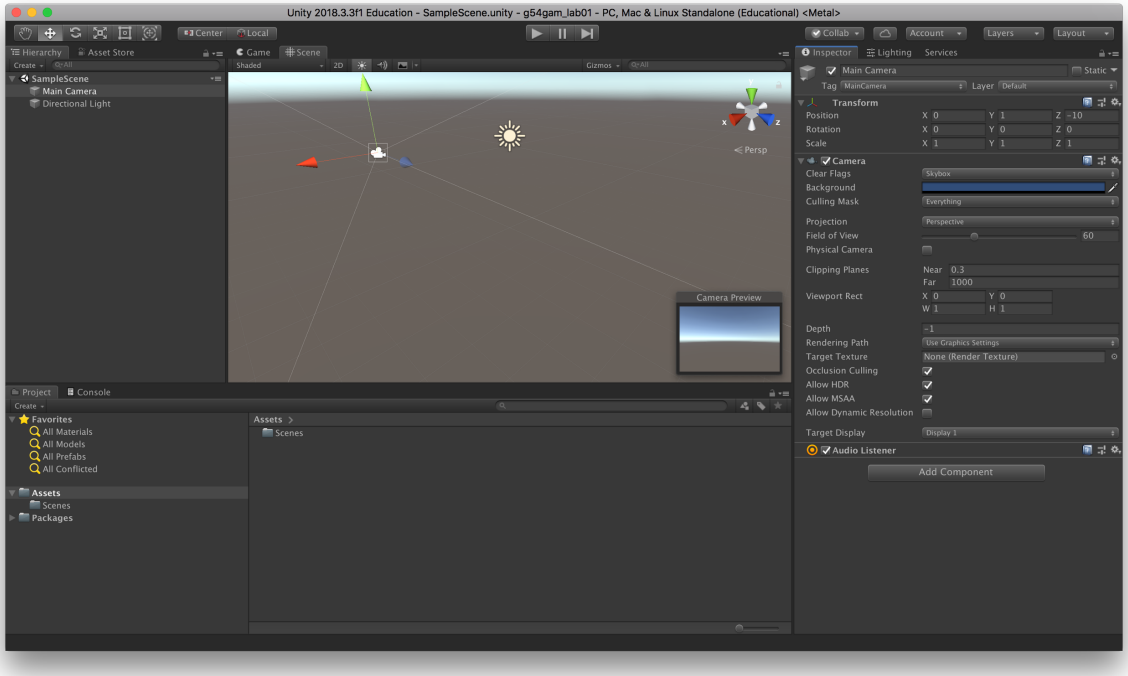
https://unity3d.com/learn/tutorials/s/space-shooter-tutorial

The assets (models, textures etc.) are available as a zip file on Moodle, or via the Unity asset store alongside a completed version of the project here:

https://assetstore.unity.com/packages/essentials/tutorial-projects/space-shooter-tutorial-13866

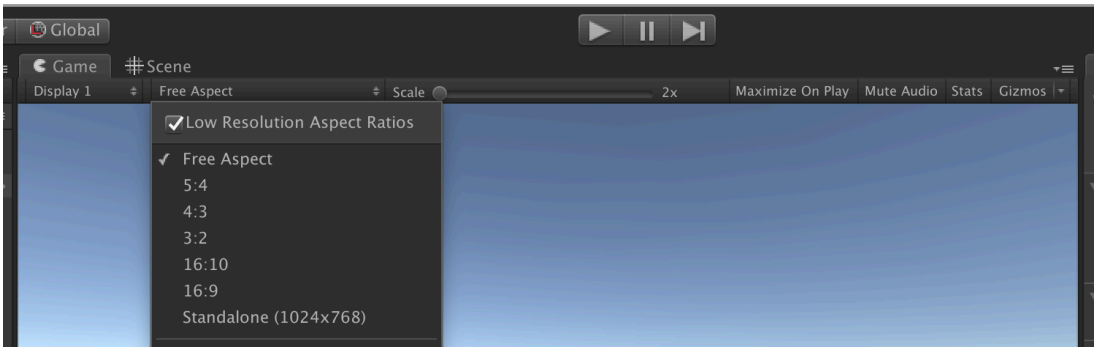
All of the setup and code required for this exercise is available above, however it is incorrect to just copy and paste code, or to attempt to follow instructions in the tutorial by rote, as that is not the aim of the exercise. It is important that you experiment with various parameters and also try to put your own unique interpretation onto what you make. It is also important to note that there may be differences between versions of Unity.

Unity IDE



When you first start Unity you may be prompted to sign in. You can either create an account or choose to work offline. Either way, click *New* to create a new project on disk in an appropriate location, leaving the default setting as 3D. This will open the main Unity window as above, with an empty scene to build the game in. The Unity IDE consists of multiple tabs:

- Bottom – Project assets and console. This shows assets associated with the project – the contents of the project's assets folder – textures, 3D models, C# scripts, and generic game objects – prefabs – that we want to reuse. Assets can be dragged from this tab into the scene or the hierarchy to instantiate them as game objects. The console tab prints debug logs and errors.
- Top-left – Hierarchy or scene graph. This is a hierarchical list of objects that are currently in the scene. By default, the scene has a camera and a light in it, so we can see, but nothing else. Selecting an object here highlights it in the scene and opens its properties in the inspector
- Right – Inspector. This displays the editable properties of the object selected in the hierarchy. Most objects in the scene will have a transform consisting of location, rotation and scale, as well as other properties depending on the functionality of the object. Functional *components* can be added to the object to extend its functionality, for example physics properties, collision detection, or custom C# scripts.
- Centre – Scene/Game. This presents a 3D view of the scene including the objects currently in it. Objects can be moved, rotated and scaled using their handles. The view onto this scene can be moved and rotated using the middle and right mouse buttons, and set to look along a given axis using the widget in the top right. Immediately above this view is the play button. Pressing this switches to the game tab, and starts the scene "running" – i.e. starts the game. Pressing it again stops the game and returns to the editable scene view. Here you can also set the game view to maximise on play, and also set a desired aspect ratio for the game window.



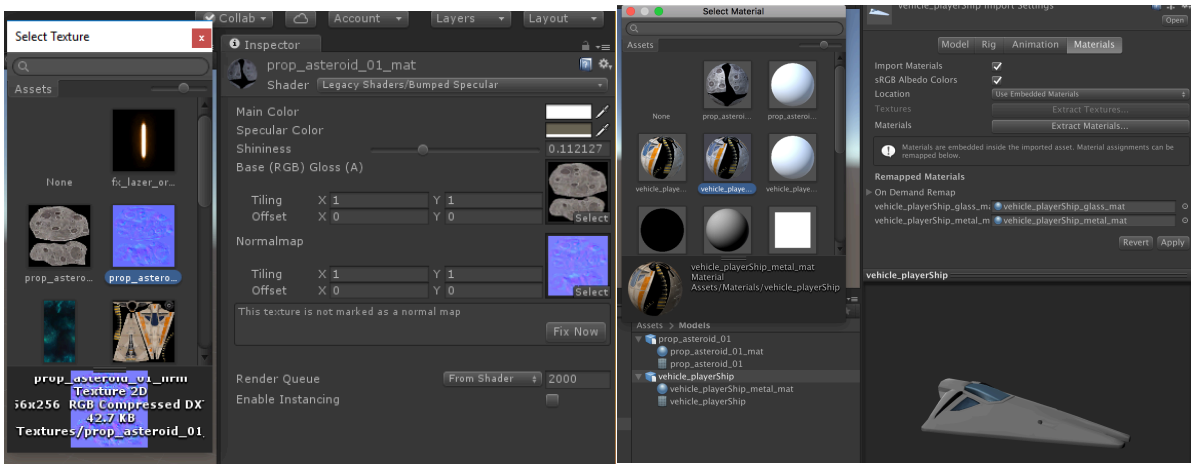
Creating the Environment

Note – the empty scene will either be named *SampleScene* or be unsaved. In the latter case save it with a sensible name (e.g. BasicScene) in the assets folder.

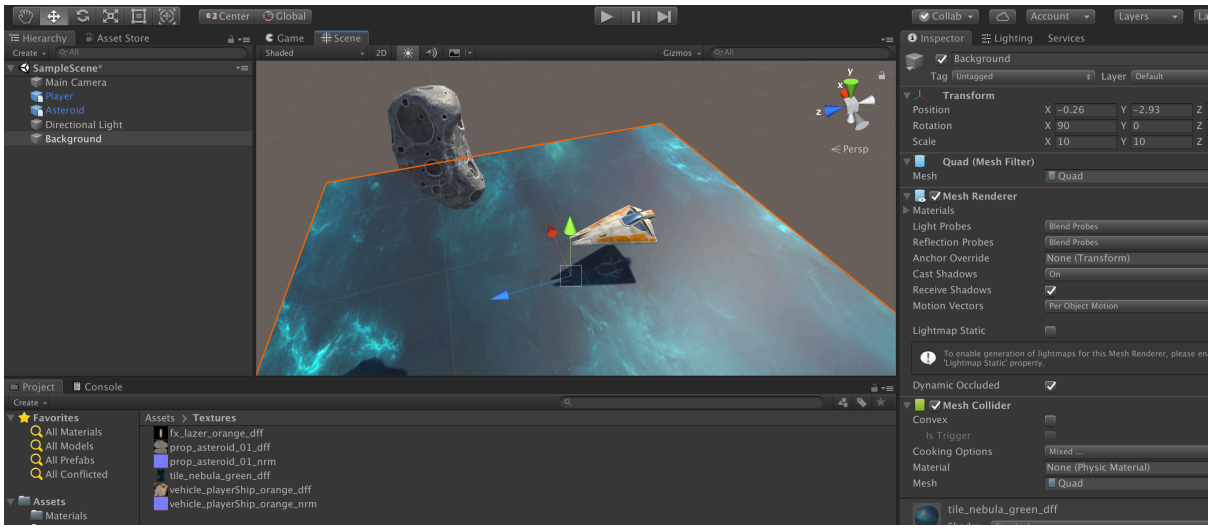
The first step is to import graphical assets for the ship that the player can control, asteroids that they must avoid, and a background image. Note that the process here may vary depending on which version of Unity you are using, and whether on PC or a Mac.

Drag the contents of the downloaded assets folder onto the Project tab in Unity, i.e. the Models, Materials and Textures folders. This has the effect of importing the assets into the project and copying them into your project's Assets folder.

> [This has imported the 3D models but has not associated the correct material for the model – they can be used but will appear plain. First, *textures* – raw image data – need to be associated with *materials* – instructions on how to light and draw the texture. Select each *material* in the assets tab and assign the appropriate *base* texture (the unfolded image of the asteroid) and *normalmap* (the corresponding purple image of the asteroid). Repeat this for the ship materials.
>
> Finally, select each *model* asset and, in the Remapped Materials section of the inspector select the appropriate material assets and click apply – you will be able to see the model is now textured with the appropriate image. Repeat this for the asteroid model.]



Drag the ship model from the Assets collection onto the scene view. Unity will give it a default name, but it is good practice to rename it to something sensible – for example *Player* (hierarchy->right click->rename). It is also good practice to position objects at 0,0,0 unless they need to start somewhere else in the scene. Select the ship in the hierarchy and then use the reset function to reset it to the origin (inspector->transform->gear menu->reset).

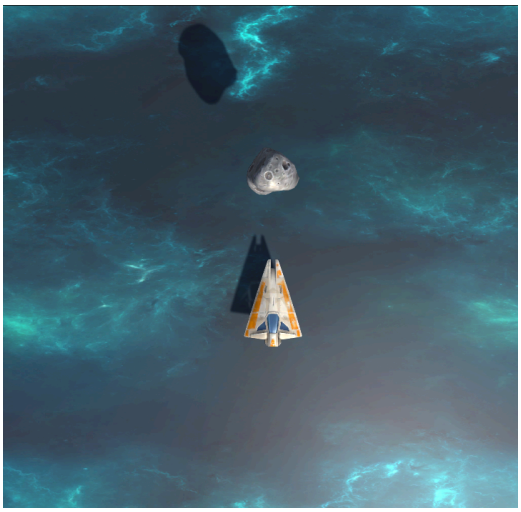


If you like, try dragging another ship onto the player object in the hierarchy. This will create a child object that inherits position, rotation etc from the parent object.

Add an asteroid to the scene in the same manner.

Use the hierarchy->create->Quad menu to create a rectangular panel in the scene to be used as a background. Rotate this by 90 degrees around the X axis to make it flat rather than standing up, and scale it by a factor of 10. Drag the *nebula* from the assets folder onto this quad to make it look interesting. Finally move the quad down (-y) in the scene so that it is below the ship.

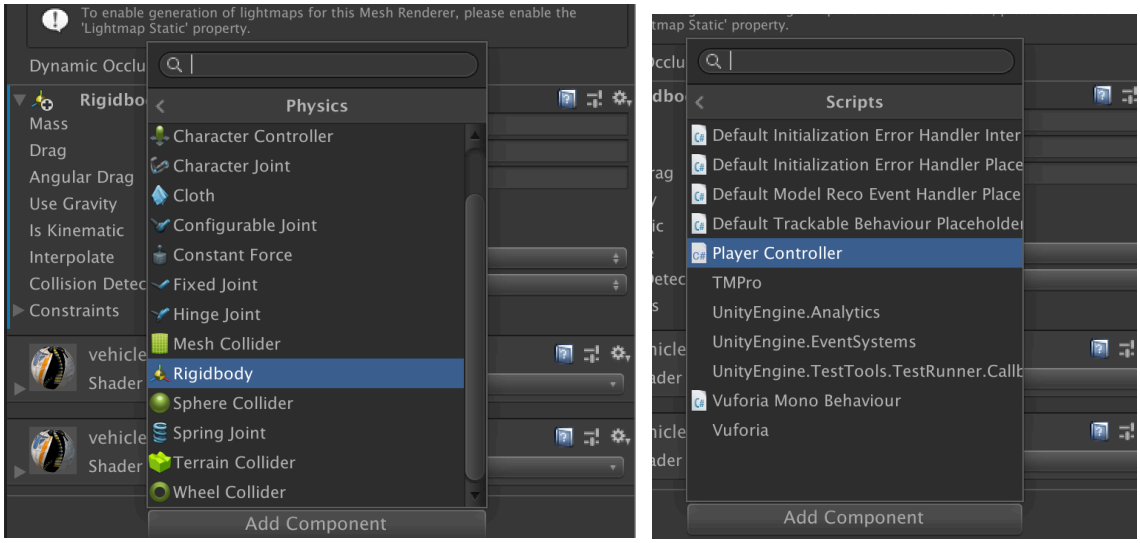
Now select the camera, and move and rotate it (again by 90 degrees around the X axis) so that it is directly above the ship and pointing downwards. Change it to an *Orthographic* camera via the inspector, and change its settings until you feel like you have a good view of the ship in the scene. The orthographic camera, unlike a perspective camera, does not take into account depth – making it ideal for a top-down game.



Playing the game should result in the view switching the camera view and displaying a top-down view onto the objects and background, although at this stage they will not do anything.

Movement and Control

At the moment, the various models in the scene are just that – models that are rendered using a mesh renderer component. To give the ship the ability to move and fly around it needs a physical as well as a visual presence. Select the Player object and in the inspector use the Add Component button to add a Physics->Rigidbody component to the object. This prompts Unity to treat the object as if it had a physical body associated with it – i.e. one that can have forces applied to it, such as gravity, and will move accordingly.



If you play the game now the ship should disappear – gravity has pulled it down through the background quad and it can now longer be seen. Disable the *Use Gravity* flag in the properties for the Rigidbody component and it should remain still.

The next step is to create a *Script* that will apply forces to the rigidbody in response to user input – essentially allowing the player to fly the ship around. Create a new *Script* asset by right clicking in the asset view and creating a new *C# Script*. Name it something sensible such as PlayerController.

You should now be able to add the new script to the Player object as a component in the same way that a Rigidbody component was added to it. Double clicking the script asset will open it for editing in Visual Studio.

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

public class PlayerController : MonoBehaviour
{
    public float speed = 5.0f;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float horizontalMovement = Input.GetAxis("Horizontal");
```

```
        float verticalMovement = Input.GetAxis("Vertical");

        Debug.Log("Input: " + horizontalMovement + " " + verticalMovement);

        Rigidbody r = GetComponent<Rigidbody>();

        r.velocity = new Vector3(
            horizontalMovement * speed,
            0.0f,
            verticalMovement * speed);
    }
}
```

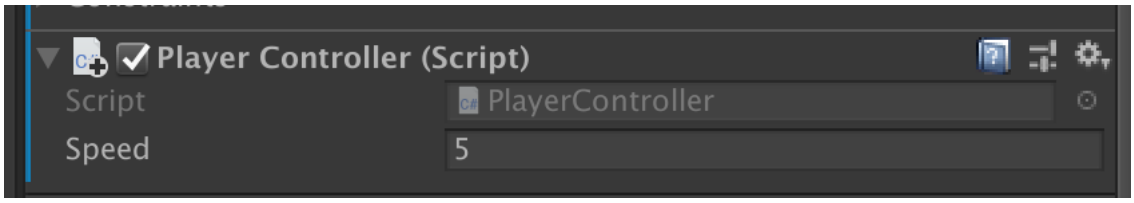
Consider the above C# code. There are two methods within the PlayerController object. Start() is called when the object is first instantiated. Update() is called once per frame – i.e. whenever the screen is updated (this is a simplification, but will suffice for now).

Each time Update() is called, the local variables horizontalMovement and verticalMovement are populated with the current user input. This is a number between -1.0 and 1.0 for each axis; forwards-backwards, left-right, and by default is controlled by the WASD keys and the cursor keys.

The Debug.Log call prints out the current input to the console.

Next, a reference to the Rigidbody component created earlier is obtained, and the velocity of the rigidbody is set based on the user input multiplied by the speed variable. As the scene is a 3D environment the rigidbody's velocity is a Vector with a speed in the X, Y and Z axes.

Note that making the *speed* variable public causes it to appear as an editable property in the inspector view for the Player object.



Save the script then play the game and ensure that you can fly the ship around. As the movement of the rigidbody is unconstrained the ship can fly off the screen. Modify the Update() function to limit the position within the view of the camera by updating the position of the rigidbody having set its velocity. xMin, xMax, zMin and zMax are float variables that should be exposed publically in the same manner as the speed variable.

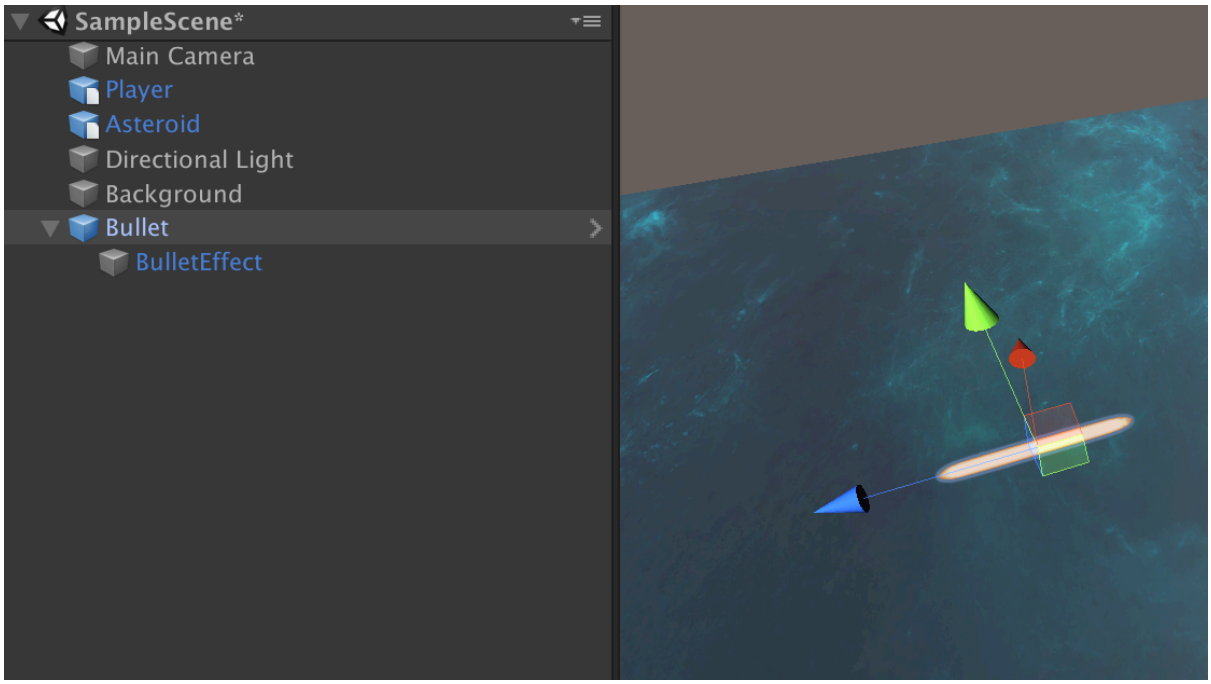
```
        r.position = new Vector3(
            Mathf.Clamp(r.position.x, xMin, xMax),
            r.position.y,
            Mathf.Clamp(r.position.z, zMin, zMax));
```

Firing

The next step is to give the ship a weapon to fire at asteroids. We're going to separate the logic of bullets from their visual display using children in the hierarchy, and then turn the object into a *Prefab* – a template that can be instantiated multiple times programmatically.

Create a new empty game object (Create Empty) in the hierarchy, give it a sensible name and reset its transform. Now create a Quad (called BulletEffect in the screenshot) in the hierarchy, and rotate it so that it is flat to the camera. Apply the *lazer* material to it – you might need to experiment with the Rendering Mode of the material as well as the Transparency settings of the texture for it to appear properly transparent. Drag the Quad over the Bullet object to make it a child. This means that when the parent Bullet object is created and moved, the BulletEffect Quad will also move, but will also be rotated relative to its parent ensuring it appears flat to the camera.

You should also remove or disable the MeshCollider component from the BulletEffect child, or Unity will print warnings.



Go back to the parent bullet object, and as with the ship add a Physics->Rigidbody component to it. Next, we want to give the bullet movement. As before, add a new script component to the bullet, and call it Mover. This code will give the bullet forward velocity when it is instantiated.

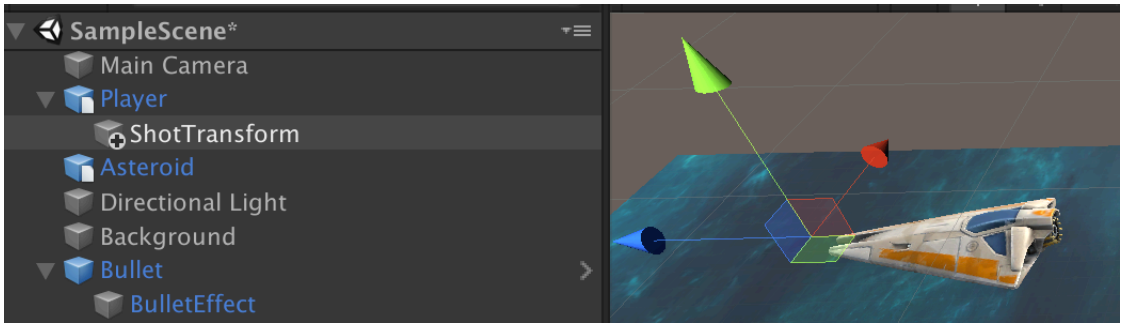
```
public float speed = 5.0f;

// Start is called before the first frame update
void Start()
{
    Rigidbody r = GetComponent<Rigidbody>();
    r.velocity = transform.forward * speed;
}
```

Playing the game, you should see the bullet object in the scene moving upwards, away from the ship.

Create a folder in the asset tab and name it *Prefabs*. Now drag the Bullet *from* the hierarchy into this new folder, selecting to create an Original Prefab if prompted. This creates a reusable asset from our Bullet, essentially a template or a class from which multiple instances can be instantiated.

Return to the Player object. Create a new empty Game Object as a child of the Player object (named ShotTransform in the screenshot below), and place it just in front of the ship. We will use the location of this object as the position to create bullets, so that they appear from the front of the ship.



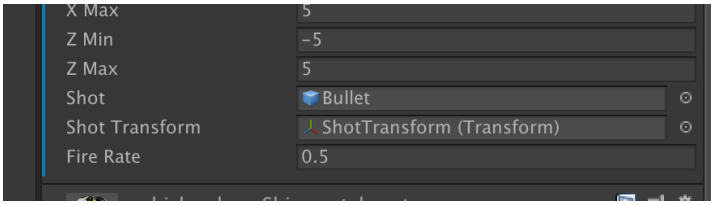

```
public GameObject shot;
public Transform shotTransform;

public float fireRate = 0.5F;
private float nextFire = 0.0F;

void Update()
{
    ...

    if (Input.GetButton("Fire1") && Time.time > nextFire)
    {
        nextFire = Time.time + fireRate;
        Instantiate(
            shot,
            shotTransform.position,
            shotTransform.rotation);
    }
}
```

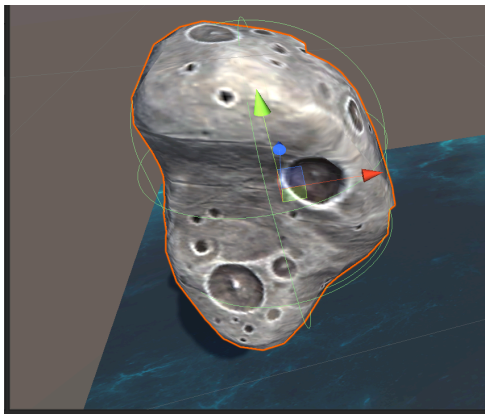
This code, added to the PlayerController script and further extending the Update function, allows the player to fire at a defined rate. It instantiates an instance of the object held in the *shot* variable, with a transform (position and rotation) as specified by the *shotTransform* variable. The final step is in the inspector to specify values for these variables – the bullet prefab and the child object at the front of the ship respectively.

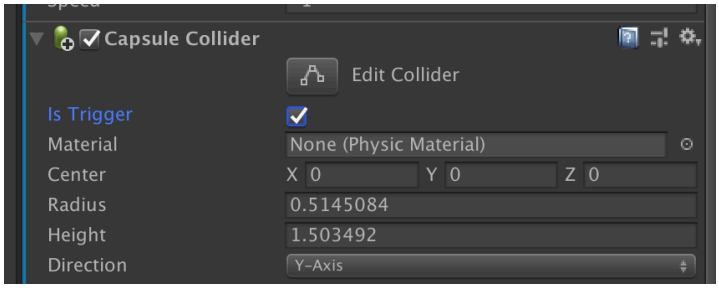


Ensure That you can fly the ship around and fire bullets (the CTRL key and left mouse click are the default fire inputs).

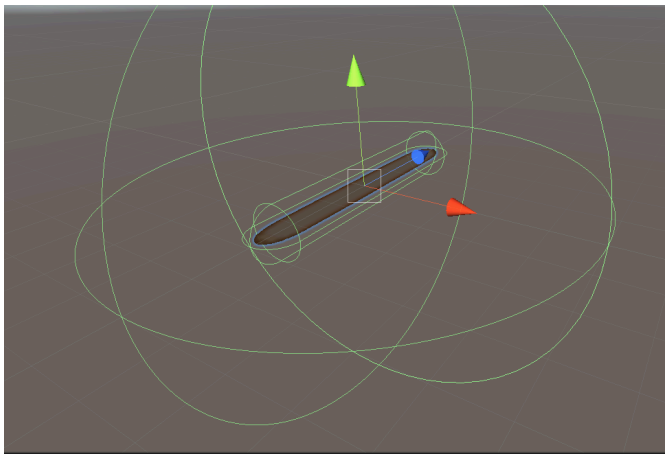
Asteroids and Detecting Collisions

The asteroid object currently does nothing – here we want to be able to shoot it, and have it destroy the ship if it collides with it. As before, give the asteroid in the scene a Rigidbody component. Attach your *Mover* script component to the asteroid, but this time set the speed to a negative value – this will give the asteroid a velocity down the screen towards the player.





Add a *Physics->Capsule Collider* component the asteroid. This will give the asteroid a measurable presence within the physics simulation and allow Unity to calculate when other objects touch or intersect with it. The capsule collider is a primitive capsule shape that should roughly coincide with the shape of the asteroid with some manipulation, but can be much more efficiently used for collision detection than the asteroid shape itself. Enable *Is Trigger* within the component – this will allow it to trigger collision events, and so we can tidy it up if it escapes from the boundaries of the game.



Repeat this process with the bullet parent Prefab, again making it a trigger. Similarly, add a capsule collider to the ship, however this time you do not need to enable it as a trigger.

Finally, we need to script what Unity should do when a physics event (i.e. a collision) occurs. Create a new script component for the asteroid object called something like DestroyByContact. This code will override the physics event when the asteroid encounters another collider (the ship or bullet colliders you have just created), and as is hopefully obvious will destroy both the object that it has collided with (the ship or the bullet), and itself.

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

```
using UnityEngine;

public class DestroyByContact : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    void OnTriggerEnter(Collider other)
    {
        Destroy(other.gameObject);
        Destroy(gameObject);
    }
}
```

Check that this works as desired with the single asteroid in the scene. If it does, drag the asteroid into the Prefabs folder, and try adding several asteroids to the scene to see how the game works. You might notice that asteroids also destroy each other if collide with one another.

Putting it all together

You should now have the basic core mechanic of a simple game – fly around, shoot asteroids, avoid being destroyed. The final step is to create waves of asteroids to provide a challenge for the player.

Create a new empty game object and name it GameController. This object will have no visual representation or physics behaviour in the scene, but will control the overall logic of the game. It will have an infinite loop that continually spawns asteroids for the player to shoot and/or dodge. Create a new script for this object (call it GameController). This code is more complicated, so consider what it does:

```
    public GameObject hazard;
    public Vector3 spawnValues;
    public int hazardCount;
    public float spawnWait;
    public float waveWait;

    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(SpawnWaves());
    }

    IEnumerator SpawnWaves()
    {
        while (true)
        {
            for (int i = 0; i < hazardCount; i++)
            {
                Vector3 spawnPosition = new Vector3(
                    Random.Range(-spawnValues.x, spawnValues.x),
                    spawnValues.y,
```

```
                spawnValues.z);
            Quaternion spawnRotation = Quaternion.identity;
            GameObject asteroid =
                Instantiate(hazard, spawnPosition, spawnRotation);
            asteroid.GetComponent<Rigidbody>().angularVelocity =
                Random.insideUnitSphere * 10;
            yield return new WaitForSeconds(spawnWait);
        }
        yield return new WaitForSeconds(waveWait);
    }
}
```

When the GameController is instantiated – at the beginning of the game, as the object is in the hierarchy - the Start() method starts a Coroutine. This is a C# mechanism that Unity uses to handle threading. If we have an infinite loop called from the StartMethod and we do it in the main thread, the "constructor" will never return, and the game logic will appear to hang.

Coroutines must return an IEnumerator, specifically a WaitForSeconds object, which we don't need to understand the details of yet, but which allows us to execute some code, but then yield control back to the rest of the engine until a certain amount of time has elapsed. WaitForSeconds is like using Thread.sleep(time) in Java, but we're returning control to the main thread until the time is up. This gives multiple objects within the scene the semblance of their own individual thread, but a single thread services all of the Coroutines in turn.

In the code above we have a simple for loop, that generates *i* hazards (in this case the asteroid) waiting for a small amount of time (spawnWait) between each one, so that they're a little spaced out coming towards the player. It then waits for a longer time between these *waves* of asteroids (waveWait) before starting again.

The position of each new asteroid is randomised across the x axis (the width of the screen) just off the top of the screen out of view). These values, and the object to create (the asteroid prefab) are again exposed via the inspector so that we can assign them).

You may have noticed that every bullet that is fired that doesn't hit an asteroid remains in the hierarchy travelling upwards, and every asteroid that the player does not shoot remains travelling downwards. This is essentially a memory leak – these are unwanted objects that are taking up memory. Create a script component, *DestroyByTime* that calls the Destroy function with a delay.

https://docs.unity3d.com/ScriptReference/Object.Destroy.html

```
void Start()
{
    Destroy(gameObject, 20);
}
```

Here, *gameObject* refers to the Game Object to which the script is attached.

Scoring

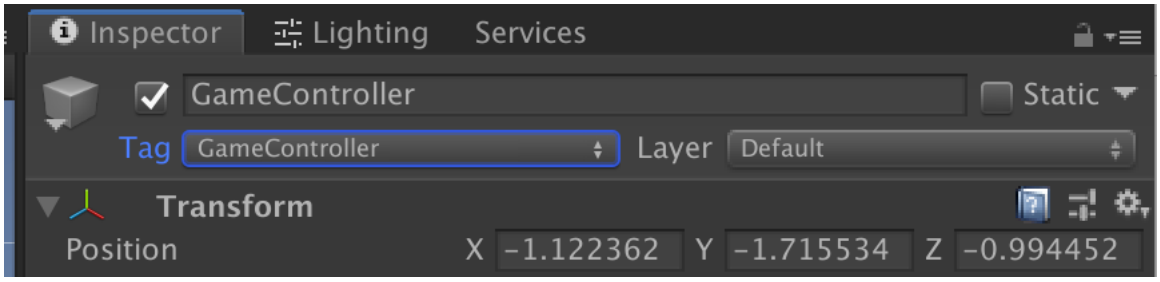
The final addition to the game is to keep track of the score in some way. At this stage we will simply print the score to the console. Firstly, add a new variable and a function to the GameController script to maintain the value of the score:

```
int totalScore = 0;

public void AddScore(int score)
{
    totalScore = totalScore + score;
    Debug.Log("Current score:" + totalScore);
}
```

To make use of this function other objects will need to have a reference to the GameController script component that is attached to the GameController object. However, as these objects (asteroids) are instantiated at runtime we cannot set the value of this property via the inspector this time. Instead, when instantiated the asteroids can find objects with a certain tag.

Add the tag *GameController* to the GameController object:



Next, in the DestroyOnContact script, retrieve the object with this tag, retrieve the script component from this object, and keep a reference to the component as a variable:

```
GameController gameController;

// Start is called before the first frame update
void Start()
{
    GameObject gameControllerObject = GameObject.FindWithTag("GameController");
    if (gameControllerObject != null)
    {
        gameController = gameControllerObject.GetComponent<GameController>();
    }
    if (gameController == null)
    {
        Debug.Log("Cannot find 'GameController' script");
    }
}
```

Then, when the asteroid is destroyed, we can call the AddScore method on the GameController script and increment the score.

```
void OnTriggerEnter(Collider other)
{
    gameController.AddScore(10);
    Destroy(other.gameObject);
    Destroy(gameObject);
}
```

As an extension, restart the scene when the ship is destroyed using the following function call:

```
Application.LoadLevel(Application.loadedLevel);
```