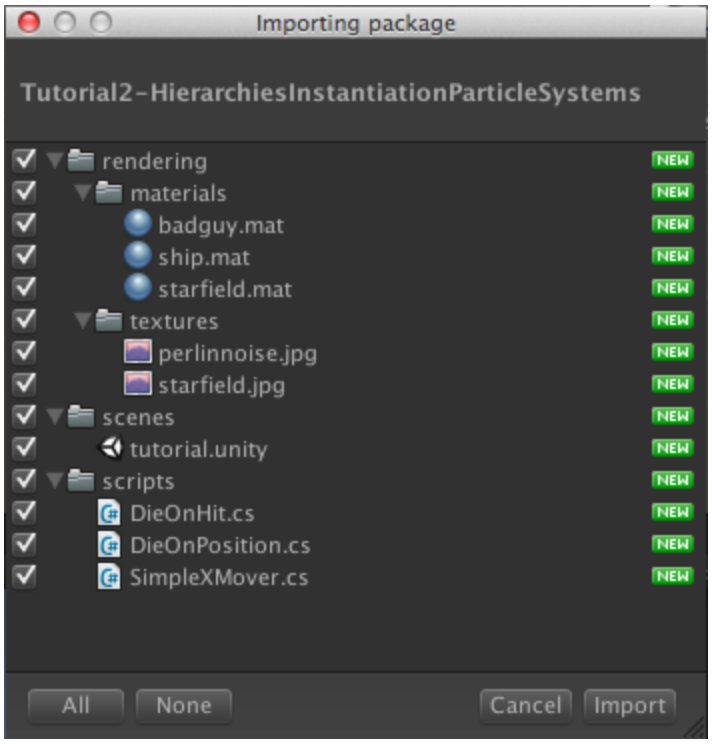
Hierarchies, Instantiation, and Particle Systems

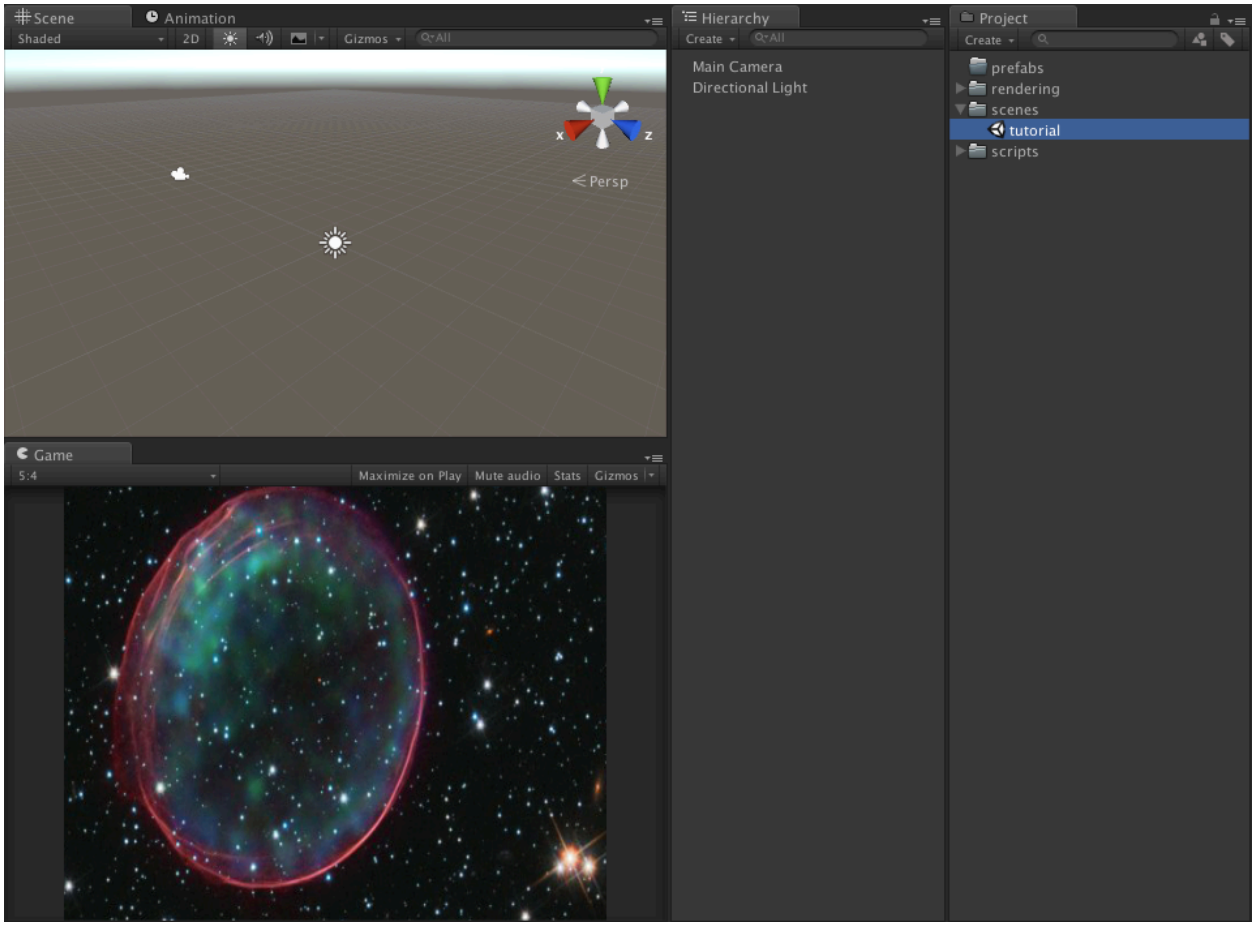
Getting Started

Create a brand new Unity project, and open this file. You should see a screen that looks like this:



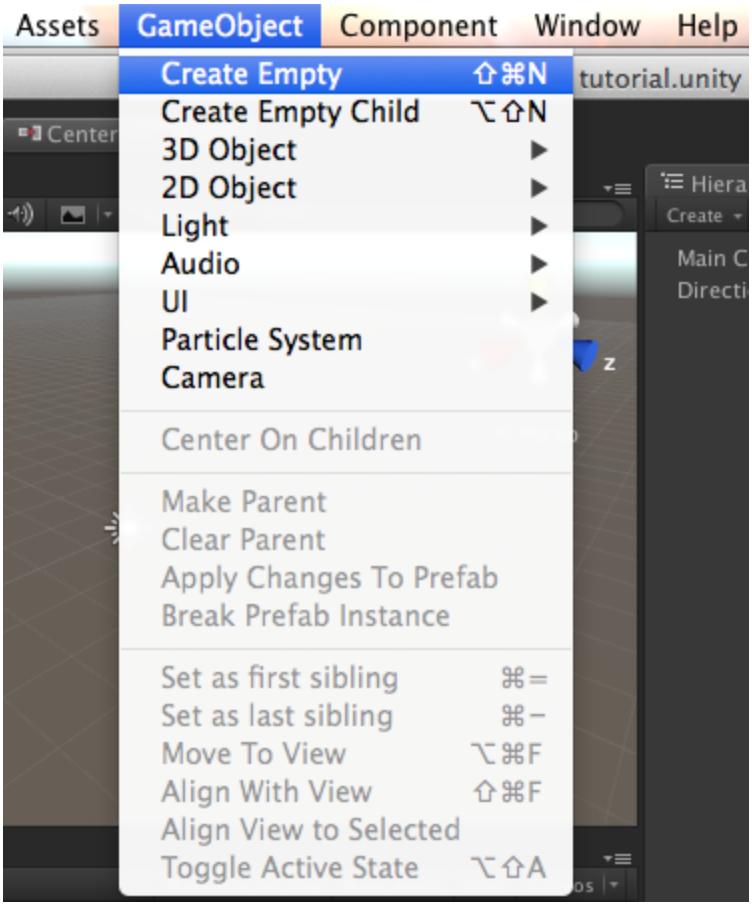
Click on the “Import” button, which will bring the tutorial assets into your project. These are a few textures and materials that will be used in the project. There are also several simple scripts that will be utilized. Feel free to review them, but they were included because they were too simple to merit having you write them yourselves.

Open up the tutorial scene by double-clicking the “tutorial” scene in the “scenes” folder in the Project window. You should see a beautiful starfield, like this:

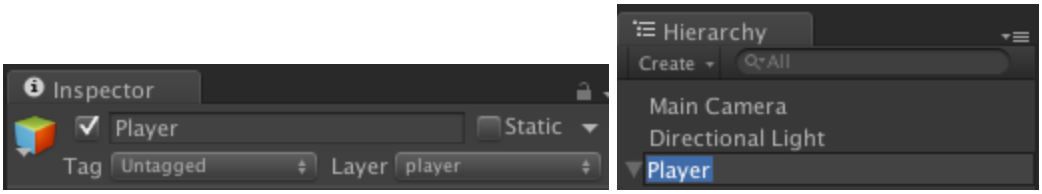


E Pluribus Unum

Game Objects in Unity are made unique by the different Components that they have attached to them. However, it is often the case where having a single GameObject with all of the required components for the gameplay would be far too unwieldy. Fortunately, Unity lets you group GameObjects together. To get started, click on the “GameObject” top-level menu, and select “Create Empty”. A new GameObject will appear in the Hierarchy window, and you will see a relatively empty Inspector window. In the Scene view, you will see controls to move the new object around, but the Game View won’t change at all. Without any meshes or renderers, the “object” has no real shape or presence. However, this is exactly what we want. Before you move on, make sure the new GameObject has a position and rotation of (0,0,0), and a scale of (1,1,1).

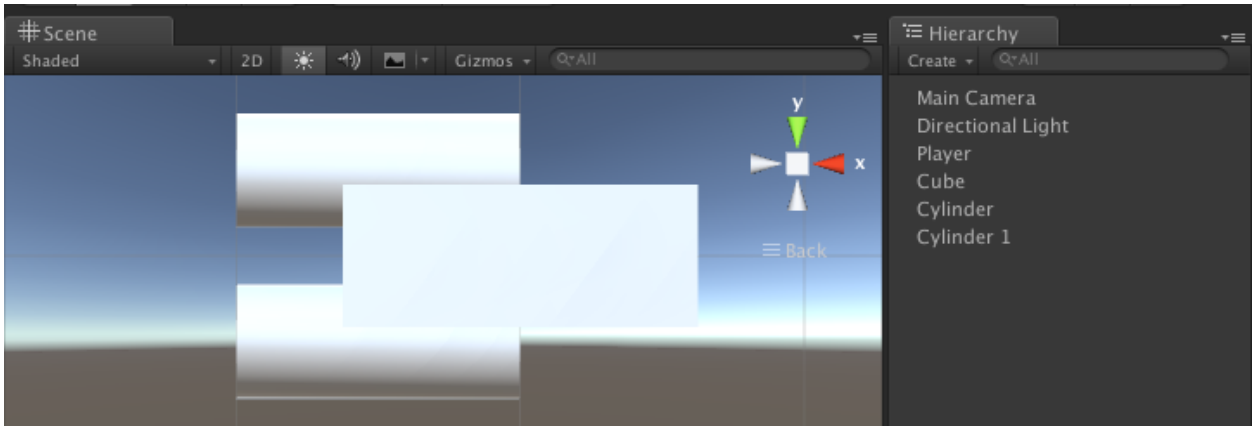


You can change the names of GameObjects in the Inspector or the Hierarchy view. In the Inspector, the text box that shows the object's name is actually a text input field. You can make changes there, and should see any changes reflected in the Hierarchy. If you double-click on the object in the Hierarchy, you will also be able to change the name of the object. For now, change the name of the GameObject to "Player".

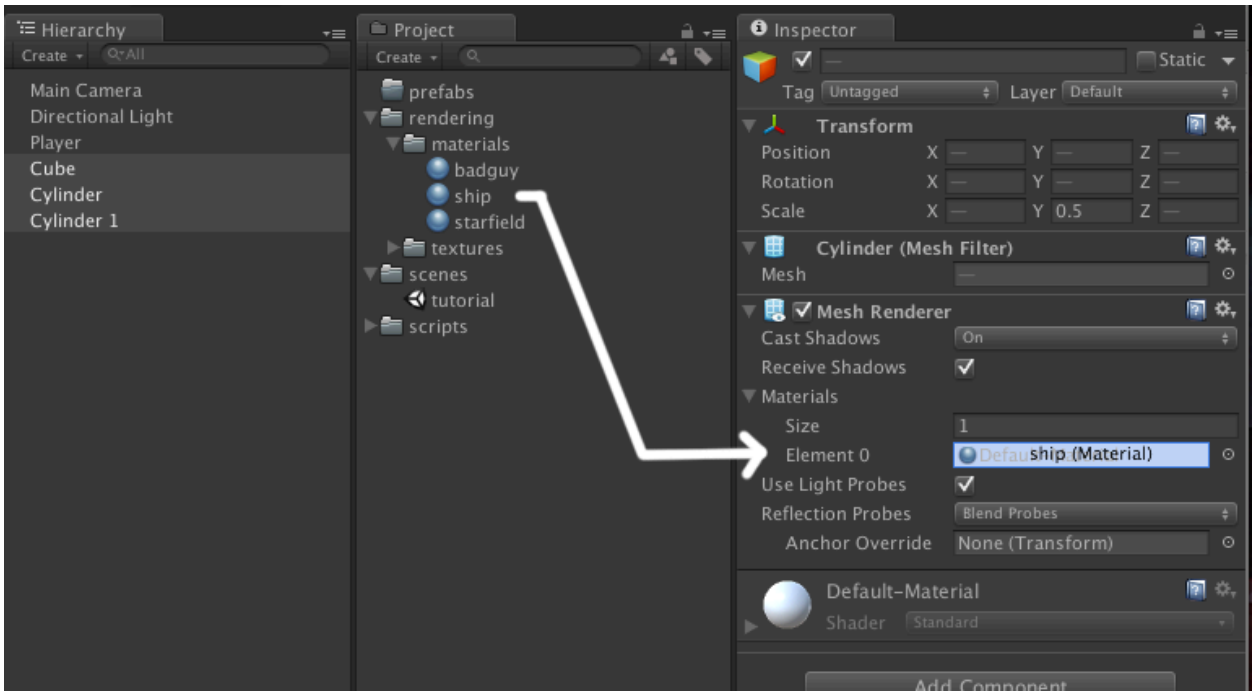


Either of these approaches will work.

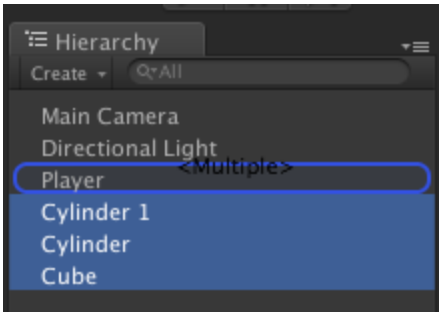
Now, create a new Cube object. Go to the Game Object menu, select "3D Object", and select a Cube. This should be familiar. Remove the default Box Collider that is given to the Cube, make sure that it is positioned at (0,0,0), and change its scale to (1.25, .5, 0.3). Create a Cylinder object, remove its Collider Component. Then position it at (-0.5, 0.3, 0.15), rotate it to (0, 0, 90), and change its scale to (0.4, 0.5, 0.4). Duplicate that Cylinder, and move the duplicate to (-0.5, -0.3, 0.15). You should now have the somewhat spaceship-shaped collection of objects shown here:



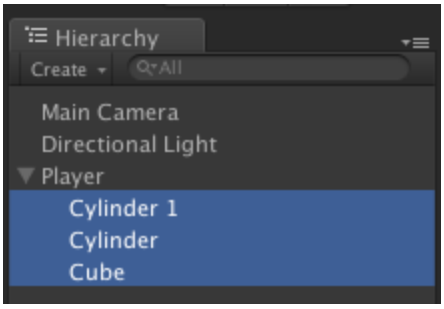
Now, select the Cube and both Cylinders. You can do this in either the Scene or Hierarchy views. You can change all 3 object's materials by clicking and dragging the "ship" material, and drop it into the Material field in the Inspector.



This should make the shapes get a bit shinier! With all three objects still selected, click on them, and drag them on top of the "Player" object that we created earlier.



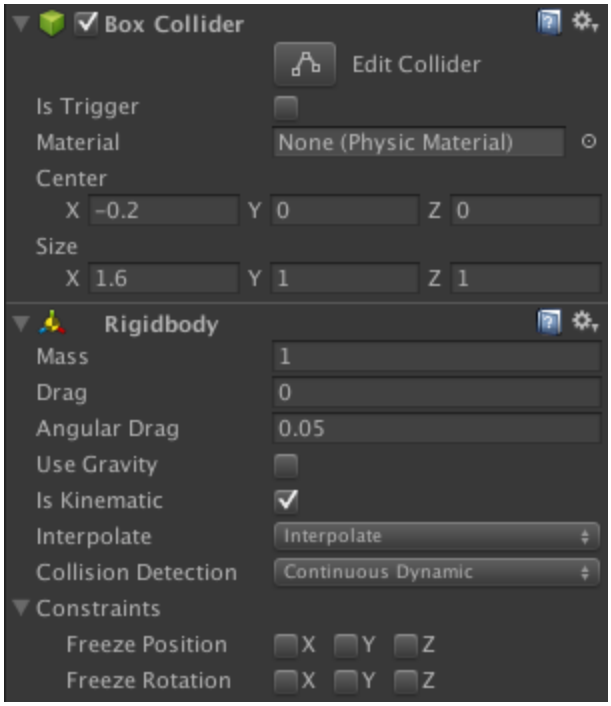
After doing this, your hierarchy should look like this:



Make note of the little arrow next to “Player”. If you click on that, the Cylinders and Cube should become hidden. If you select the Player object and move it, either by using the inspector or in the Scene view, you will see that the cube and cylinders will move as a group.This is because they are now Child objects of the Player object. The Player object is referred to as their Parent. Whenever a Parent object is moved, all of the Child objects will be moved.

Get it Moving

The next thing we’re going to do is get the player ship moving around. We’re going to set up a side-scrolling shooter-style control, like Defender or Zero Wing. Select the Player object, and add Rigidbody and Box Collider Components. Set the Components up as detailed below, and move the Player object to (-4, 0, 0).



The player is set to Kinematic because movement is going to be handled by our script, not the Physics system. It’s important to note if two objects are both set to Kinematic, they won’t collide with each other, so make sure that you pay attention to which objects you set to Kinematic. Create a new C# script in the “scripts” folder in the Project window, named “PlayerMovement”. Open up the new script, and add the following code:

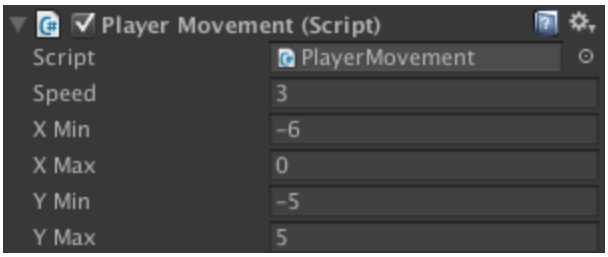
```
public class PlayerMovement : MonoBehaviour
{
    public float speed;
    public float xMin;
    public float xMax;
    public float yMin;
    public float yMax;

    private Vector3 _targetPos;

    void Start()
    {
        _targetPos = Vector3.zero;
    }

    void Update ()
    {
        _targetPos.Set(Input.mousePosition.x,
                       Input.mousePosition.y,
                       transform.position.z - Camera.main.transform.position.z);
        _targetPos = Camera.main.ScreenToWorldPoint(_targetPos);

        _targetPos.Set(Mathf.Max(Mathf.Min(_targetPos.x, xMax),xMin),
                       Mathf.Max(Mathf.Min(_targetPos.y, yMax),yMin),
                       transform.position.z);

        if(transform.position != _targetPos)
        {
            transform.position = Vector3.MoveTowards(transform.position,
                                                     _targetPos,
                                                     speed * Time.deltaTime);
        }
    }
}
```

There are some new things in this script that warrant further explanation.

- _targetPos.Set: TargetPos is a Vector3. The Set function lets you set the x,y, and z values of the Vector3 without creating a new one.
- Input.mousePosition.x/y: In the Input class, there's a Vector2 that can be accessed that tells you what position the mouse cursor is currently at, based on the number of pixels in the screen.
- Camera.main.ScreenToWorldPoint: Camera.main is a reference to the main Camera that is being used to draw the Scene. ScreenToWorldPoint takes in a point in Screen space, and gives you that position in World space. World space is the coordinate system that you move objects around in using the Scene or Inspector windows.
- Mathf.Min/Max: Mathf is a helpful collection of mathematical functions. Min/Max simply take in two numbers, and return the smaller/larger of the two. So this line, combined with the ones above it, takes in the mouse's current position, transforms the pixel position of the mouse into a position that the Player ship could be at, and then clamps that position within the bounds of xMin/xMax and yMin/yMax. The clamping is there to prevent the Player from flying off the screen!

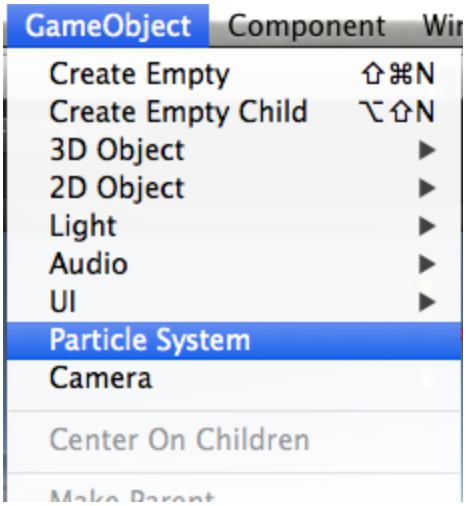
- Vector3.MoveTowards: Vector3 also has a series of very helpful functions. MoveTowards takes in a source position, a final destination, and a distance. It will return a new Vector3 that tries to get to the final destination, but won't move more than the provided distance. If you set the distance to be speed * Time.deltaTime as we do here, it will effectively move from the current position to the destination at the set speed.

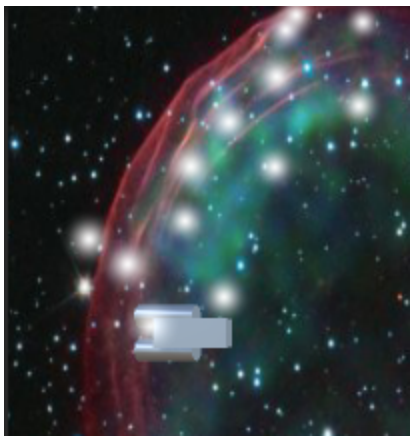
Select the Player object, and add the PlayerMovement Component to it. Set the bounds to the values provided below, and press play. You'll find that you can now move the Player ship around, but you can't move off-screen. It will always move towards your cursor, but moves at the speed that was set in PlayerMovement. If you change the speed to something higher, it'll move faster. If it's set very high, to 1000, the ship will just stay on the cursor at all times, assuming the cursor is in bounds.



Fire Up The Engines

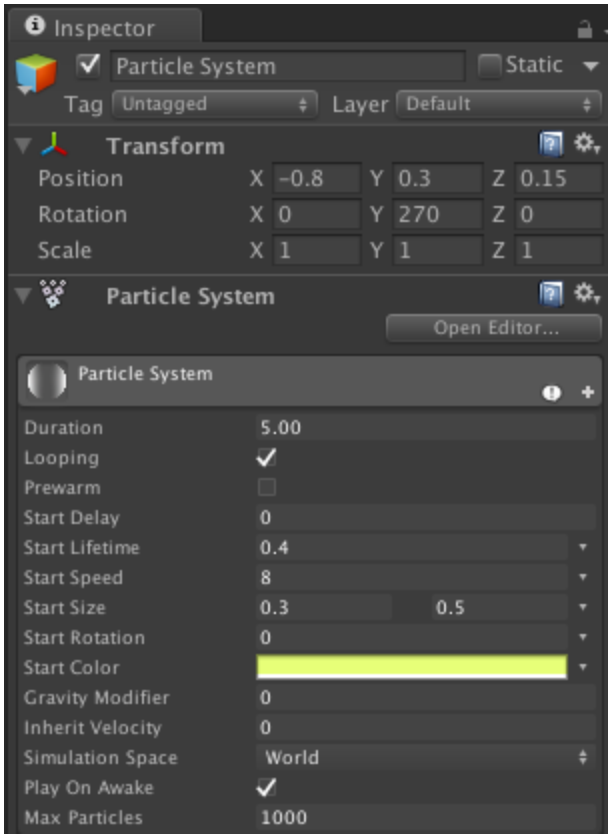
Now our spaceship is moving, but it doesn't look very spaceship-y. One of the easiest effects you can add is fire from the engines! Unity's built-in Particle Systems can be used to make lots of different effects, and fire is one of the easiest to implement. First, click on the GameObject menu, and then select "Particle System". This will create a new Particle System in your Hierarchy, which will start belching white particles upwards in your Game and Scene View. When you have a Particle System selected, it will simulate itself, so you can make tweaks and changes without having to have the game running.





With the Particle System selected in the Hierarchy, you'll also notice that there is a lot of different fields and sections in the Inspector. Particle Systems are incredibly powerful, and can be used to create a wide variety of effects. This power and variety comes from the ability to

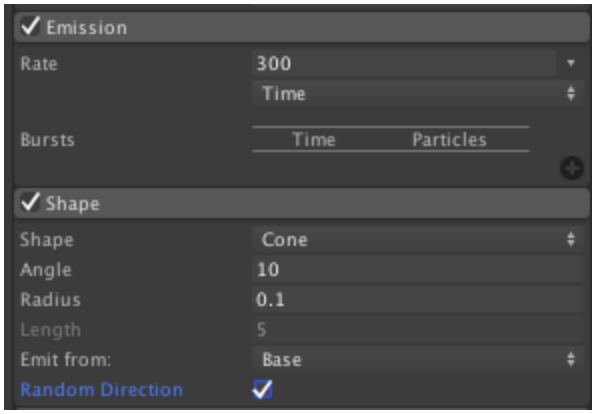
control many variables at once through the Particle System designer. To get our engine fires started, drag the new Particle System and drop it on top of the Player object to make it a Child object. Then, make the following changes to the system:



For “Start Size”, click the little triangle on the right, and then select “Random Between Two Constants”. The Start Color is (70, 135, 255) in HSV space and (231, 255, 120) in RGB space. Some important things to note:

- Duration is the lifetime of the Particle System, not each individual Particle. Changing the Start Lifetime changes how long each particle is visible.
- Start Lifetime/Speed/Size/Rotation/Color are all listed as “Start” and not just Lifetime/Speed/Size/Rotation/Color, because those values can change over time. We’ll be making some of those changes soon!
- Simulation Space is important to pay attention to. If it is set to Local, and the Particle System moves, all of the particles that have been emitted by that System will move with it. For lots effects, this is not the desired behavior, and can lead to the effect looking strange.

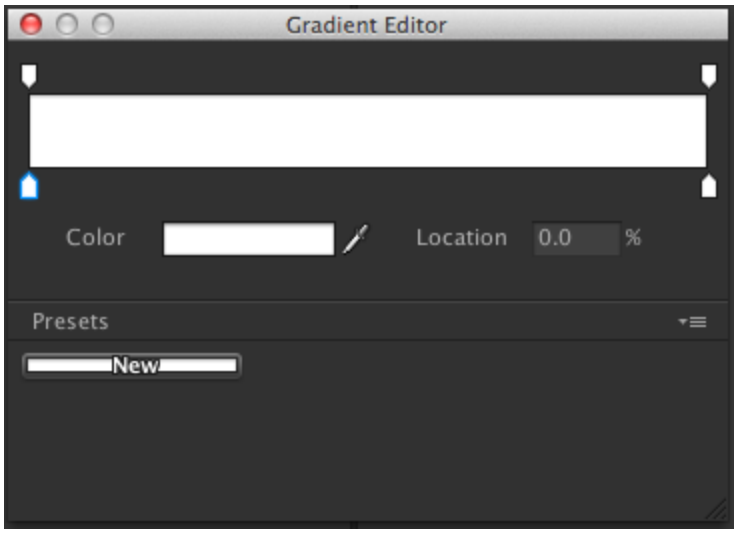
Next, click on the section that says “Emission”, and change the “Rate” value from 10 to 300. Then, click on the section that says “Shape”, change the “Angle” value from 25 to 10, the Radius value from 1 to .1, and check the checkbox that says “Random Direction”. Those sections should look like this now:



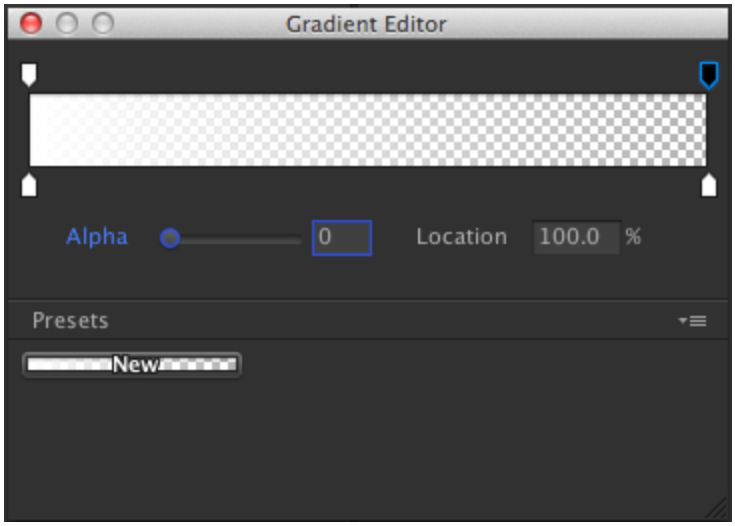
The Emission section is used to control how particles are emitted, and can either be done at either a set rate, or in specifically designed Bursts. The Shape section controls how the particles are placed when they are created, and what their initial direction is.

Currently, the particles are flying out, going a short distance, and disappearing. This doesn't look like exhaust, so we will now make changes to have the particles fade out. First, click the check marks next to the "Color over Lifetime" and "Size over Lifetime" sections, and then click the section headings to expand them.

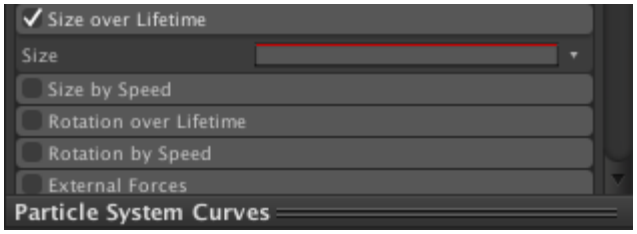
Click on the Color field, and you should be presented with a window that looks like this:



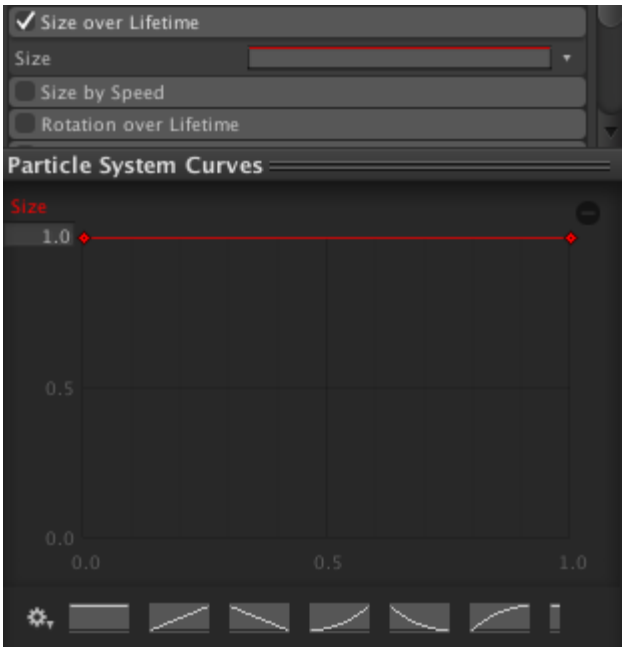
There are four arrows that show the progress of the gradient. The two on the bottom control the color. Since they're both white, that means that the Start Color will not change. The two arrows at the top control the transparency of the particle. The arrow on the left side represents the transparency when the particle spawns, and the arrow on the right is the transparency when the particle's lifetime ends. Click on the arrow on the right side, and then drag the "Alpha" slider down to 0, like this:



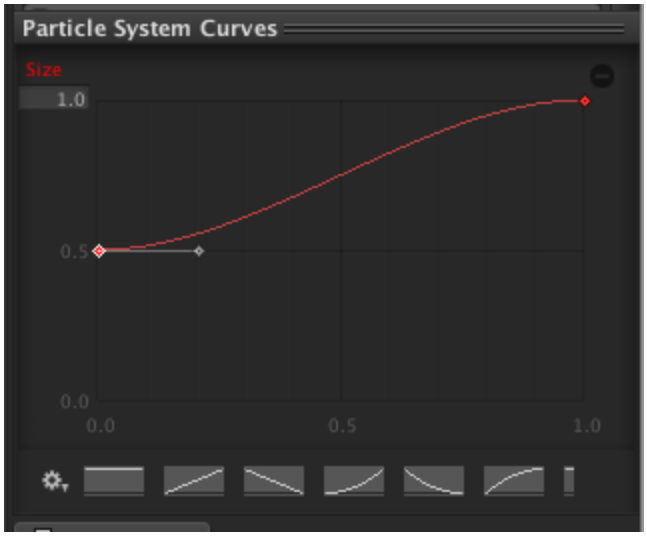
If you look at the Particle System now, you'll see that instead of just disappearing, the particles appear to fade out. Now, go to the "Size Over Lifetime" section. Click on the "Size". You might not see anything change in the Inspector window. If that's the case, click on the bar that says "Particle System Curves". This should bring up an editable graph.



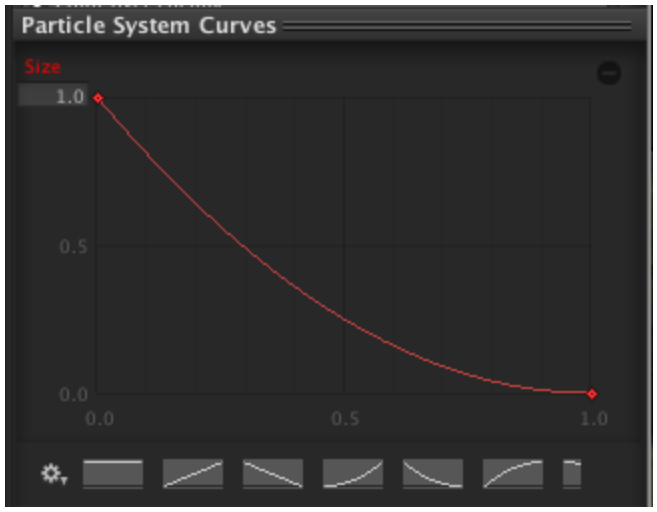
<- Click here to get…

This window 

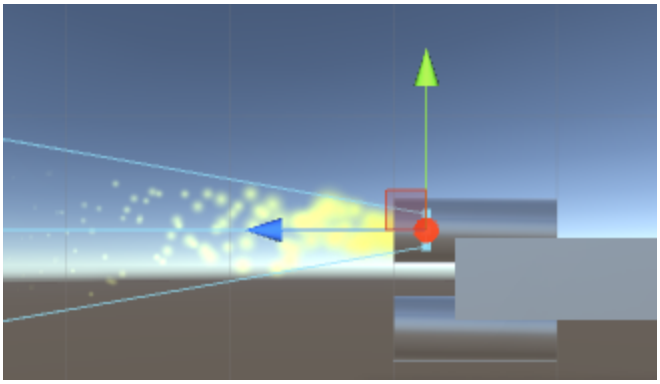
In this window, the red diamond shapes represent the start and end of the particle's lifetime. Click and drag the left-side diamond down to 0.5. This will start the particles at half of the Start Size we defined earlier.



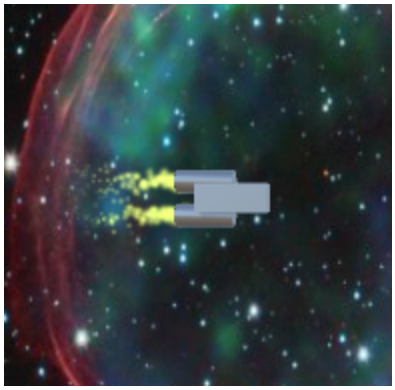
The grey diamond that appears when you click on the red diamond is a control point. You can click and drag that to control the curve from 0 to 1. If you like, you can also just click one of the curves at the bottom of this screen, and Unity will set the curve for you. Select the 5th curve, which starts at 1.0, and drops to 0.0:



The particle system that is currently animating will have particles that get smaller and fade out, adding a nice taper effect.

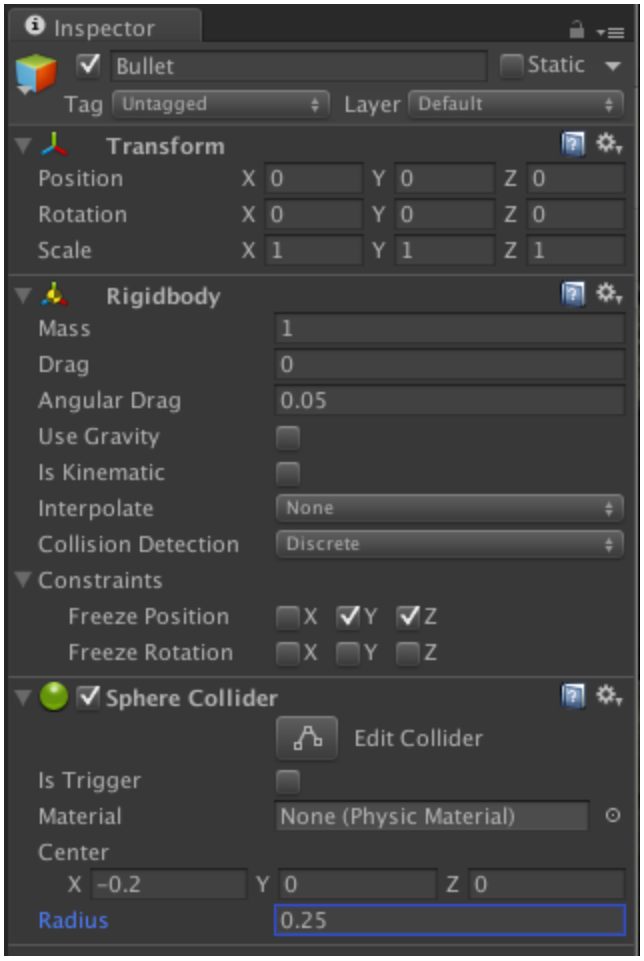


Now, since we have two engines, create a duplicate of that particle effect, and move it to (-0.8, -0.3, 0.15). Press play now, and when your ship moves around, you'll see that both engines look like they're firing!



Fire Forward Phasers

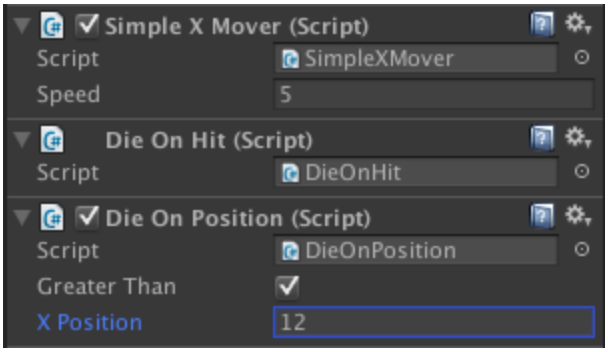
Our next task will be to have our ship fire laser bullets. To start, create a new empty GameObject (using the GameObject menu). Make sure that the new object is named "Bullet", and placed at (0, 0, 0). Add Rigidbody and Sphere Collider components, and set up their parameters like this:



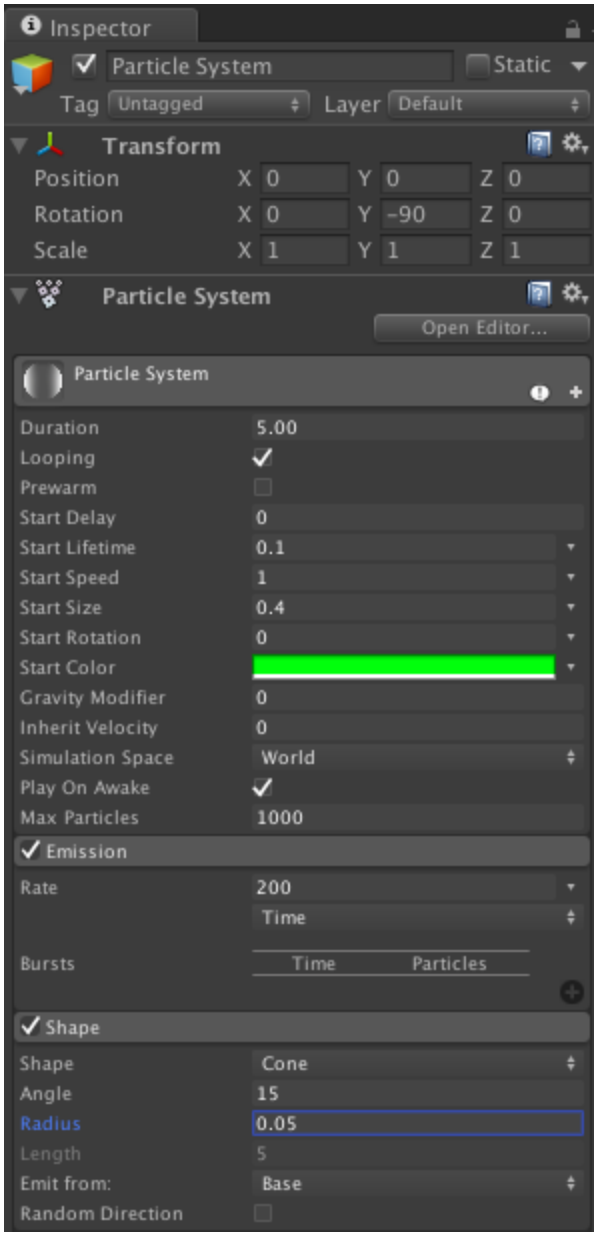
You should also add a “SimpleXMover” component, which is a script that was provided in the setup files. All this script does is give an object a speed to move at. Set the speed to 5. Also add “Die On Hit” and “Die On Position” components. These are also provided scripts, that remove the object when something happens. Open up the DieOnHit script, and you’ll see its very simple code:

```
public class DieOnHit : MonoBehaviour {

    void OnCollisionEnter(Collision c)
    {
        Destroy(gameObject);
    }
}
```

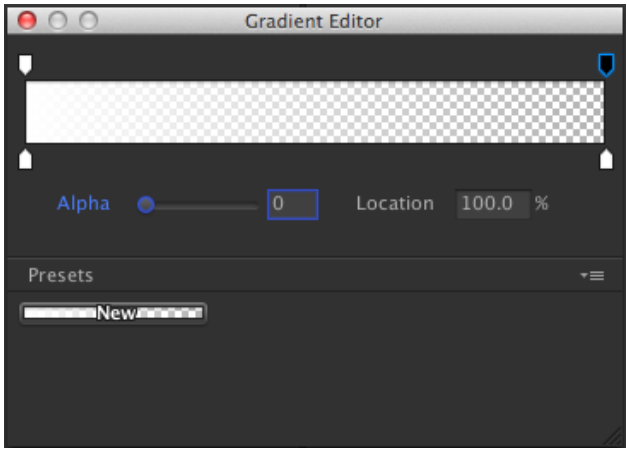
When two objects that have Colliders run into each other, Unity sends a message to all components called OnCollisionEnter. When this script gets this message, it calls the Destroy function, and passes its own gameObject as a parameter. This will cause the object to be removed from the scene if it hits something. DieOnPosition is similar, except it checks the position of the object. If the position is too far from the point defined in the parameters the object is removed. This is to keep our scene from getting too cluttered. Before you move on, make sure that SimpleXMover and DieOnHit have their values set properly:

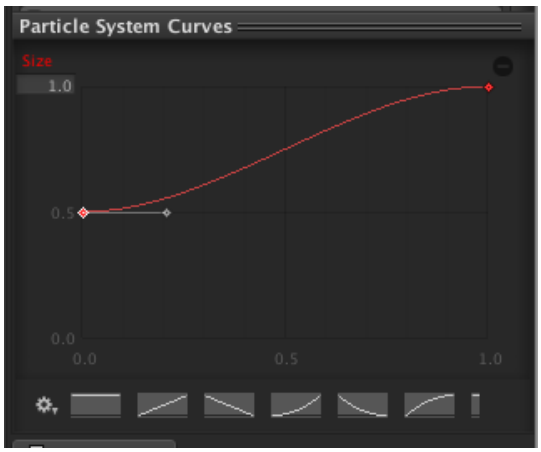


Now, create a new Particle System (GameObject menu -> Particle System). Drag the Particle System on top of the Bullet object, to make it a Child object. Make sure that the Particle System's position is (0, 0, 0), and set its rotation to (0, -90, 0). Set the Particle System up like so:

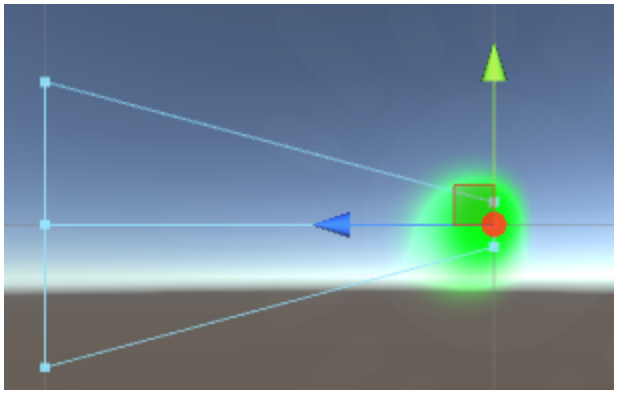


Set up the Color Over Lifetime like we did with the engine, with the Alpha on the right being set to 0. Also set the Size Over Lifetime. However, this time, set the left control point to 0.5, and the right control point to 1.0:

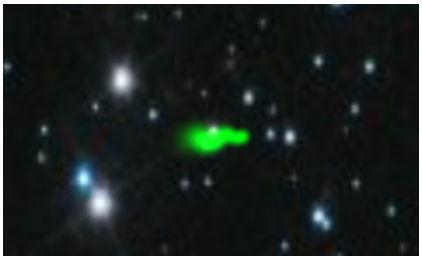




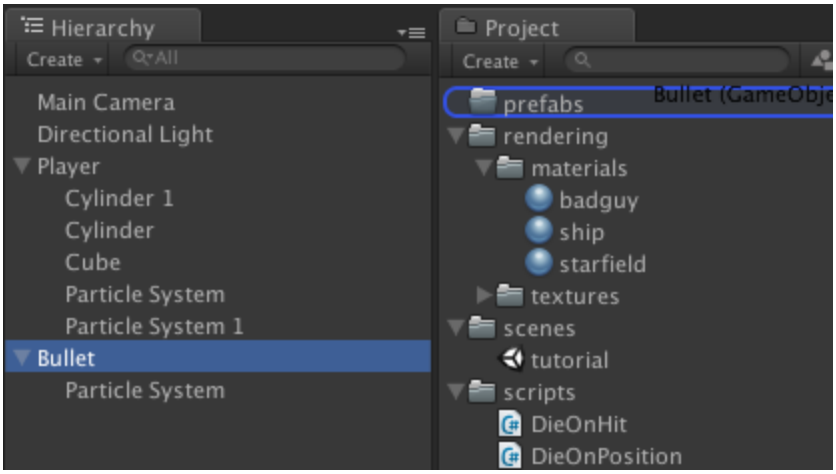
The simulated Particle System will just look a single green dot that jitters. This is expected!



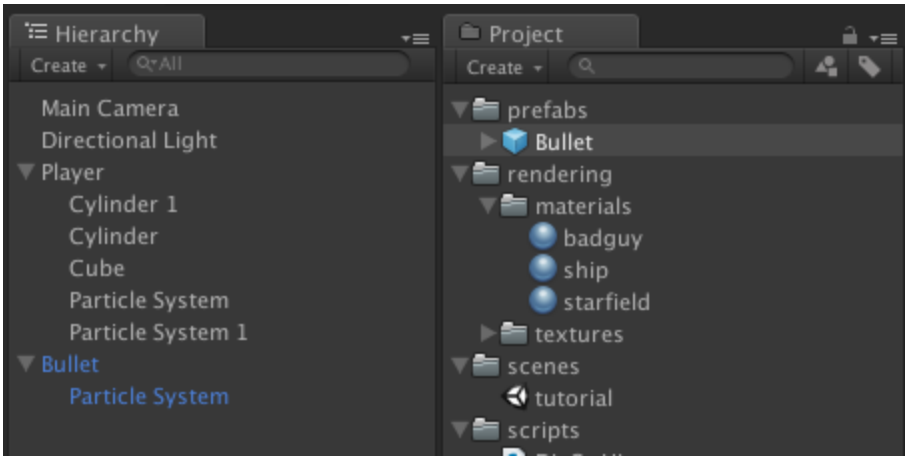
If you press the Play button now, you'll see what effect this has. Since the bullet moves to the right, the particles form a little trail.



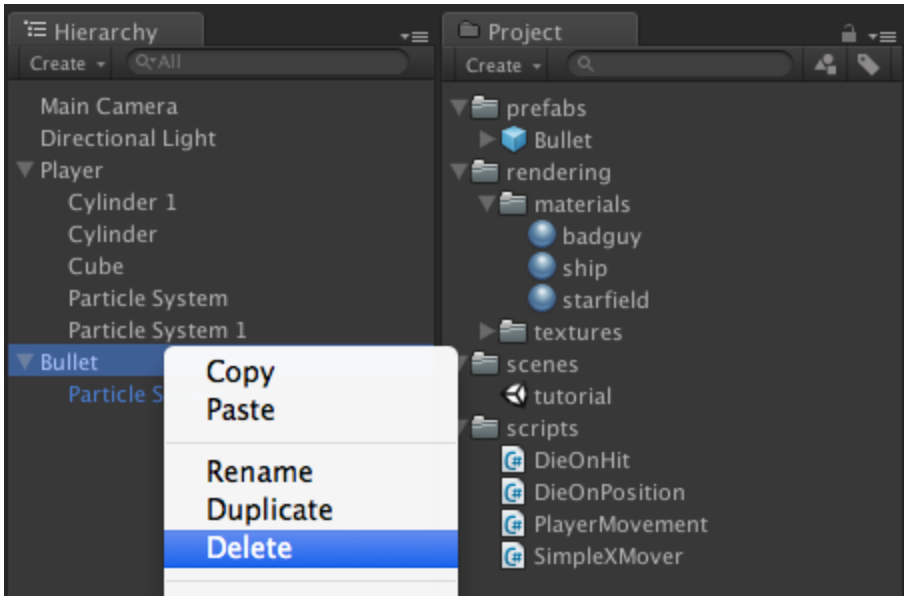
Now we have a single bullet, but we're planning on firing lots of them! Fortunately, Unity has a helpful system that we can use. We're going to turn this bullet into a Prefab. Prefabs are Game Objects that are created in a scene, but then stored in the Project. To create a Prefab, click on the Bullet object, and drag it into the provided "prefabs" directory in the Project window.



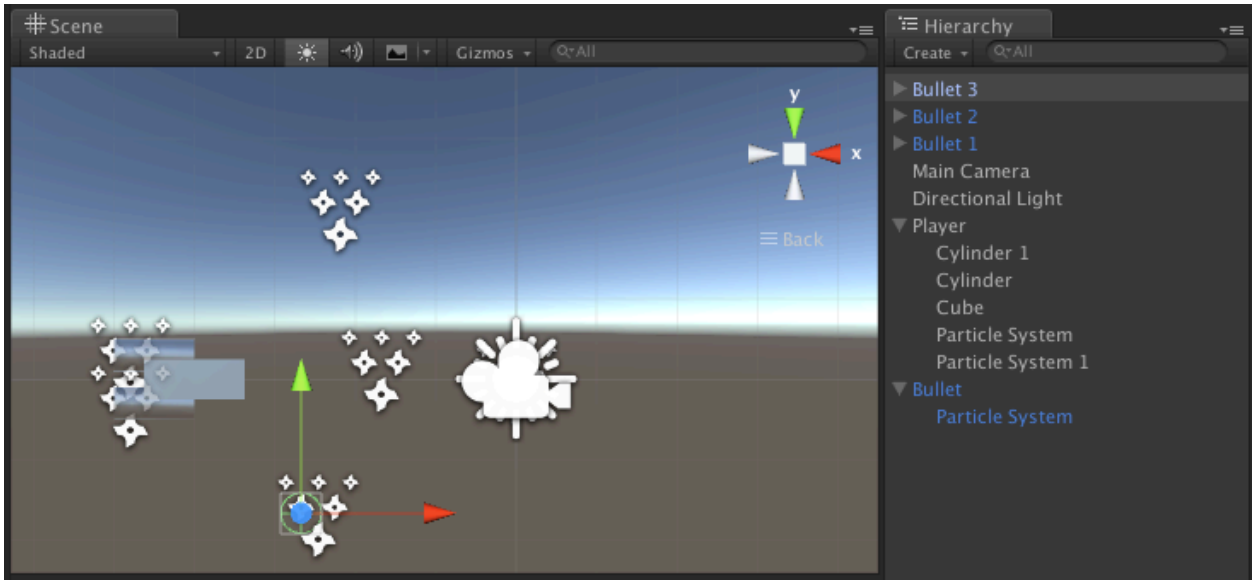
If you do this correctly, there will be a “Bullet” object in the Project window, under the “prefabs” folder. Also, the Bullet object in the Hierarchy will have blue text. The blue text indicates that the object is an Instance of a Prefab.



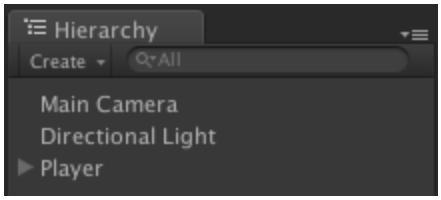
Now delete the Bullet in the Hierarchy. Right-click on it and select “Delete”. Make sure you’re deleting the one in the Hierarchy!



You've deleted the Instance of the Prefab, but the original Prefab is still in the prefabs directory in the Project view. You can click on the Bullet prefab, and drag it into the scene. This will create a new Instance. In fact, you can drag the prefab into the scene multiple times, this will just create more Instances of the Prefab. If you press Play, you'll see all of the Instances move across the screen.



We want the bullets to fire when the player clicks their mouse button, not just at the start of the game. So go through and delete all of the Bullet Instances in the Hierarchy. Your Hierarchy should look like this when you're done:



Now, we will create the script that will fire bullets when the player clicks. Create a new C# script in the “scripts” folder named “BulletShooter”, and add this code to it.

```
public class BulletShooter : MonoBehaviour
{
    public float fireTime;
    public GameObject bullet;
    public Vector3 positionOffset;

    private float _timeToFire;

    void Start () {
        _timeToFire = 0f;
    }

    // Update is called once per frame
    void Update ()
    {
        _timeToFire -= Time.deltaTime;

        if(Input.GetMouseButton(0) && _timeToFire <= 0f)
        {
            Instantiate(bullet, transform.position + positionOffset, Quaternion.identity);
            _timeToFire = fireTime;
        }
    }
}
```

This code takes in three public parameters:

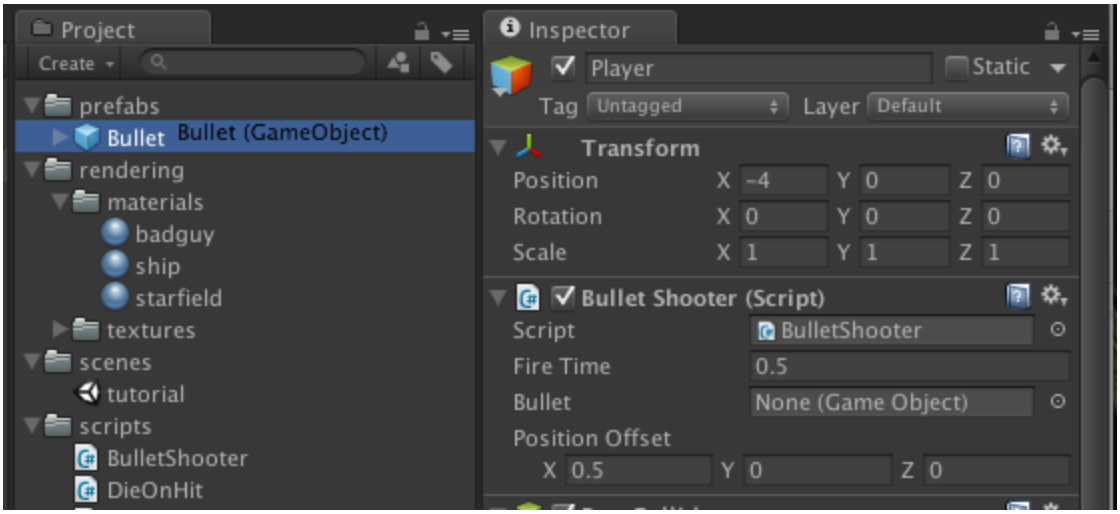
- fireTime: A float that represents how often the ship can fire.
- bullet: A GameObject that is our bullet object. In Unity, you can have GameObjects as references in scripts. These GameObject references can either be objects that exist in a Scene, or a Prefab. We’ll be making use our Prefab here.
- positionOffset: This represents where our bullets are going to fire from. We don’t want bullets to fire exactly our ship’s position, because that position is the center of the ship, and we don’t want our bullets to hit our own ship!

In the update function, _timeToFire is decremented by Time.deltaTime. This is because in the next line, we check to see if the mouse button is down using Input.GetMouseButtonDown(0). If the mouse button is down, we also check that _timeToFire is less than or equal to 0. If both of those are true, then we’re going to fire a bullet!

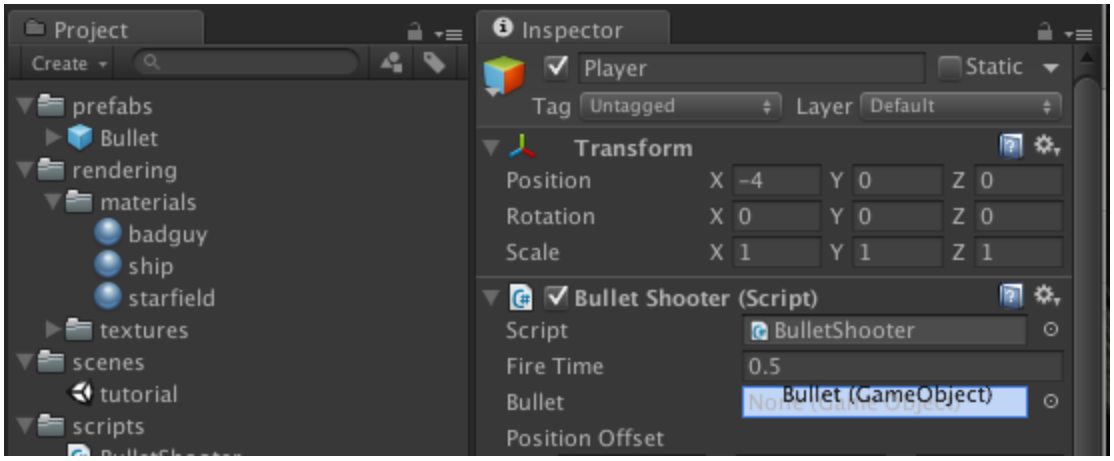
To do that, we use Unity’s built-in Instantiate function. That function takes in a GameObject and makes a new Instance of that object. We also provide a position, in this case the position is just our position (transform.position) plus the provided offset. The Instantiate call also requires a rotation. Rotations in Unity are done with Quaternions. You don’t need to worry about this now, just use the built-in value Quaternion.identity. After we create the new bullet with Instantiate, we set _timeToFire to fireTime, so that the player can’t fire again until it decrements down to 0.

Go back to Unity now, and add the BulletShooter component to the Player. Set “Fire Time” to 0.5, and the Position Offset to (0.5, 0, 0). To set the Bullet parameter, click on the Bullet prefab in the Project view, drag it over the field that says “None (GameObject)”, and release the mouse button to drop it. You can also click the little circle next to “None (GameObject)” and select the Bullet prefab from the Assets tab in the window that pops up.

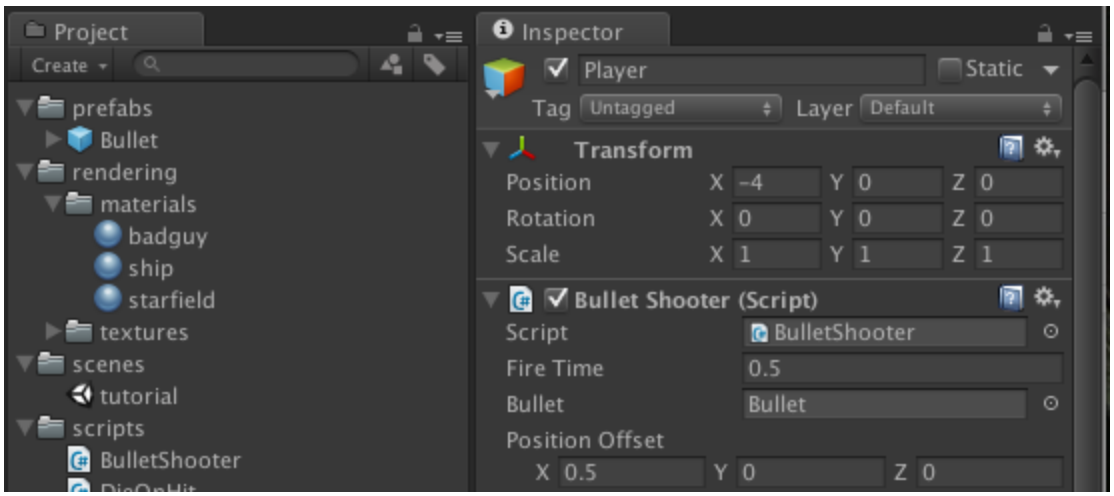
Click



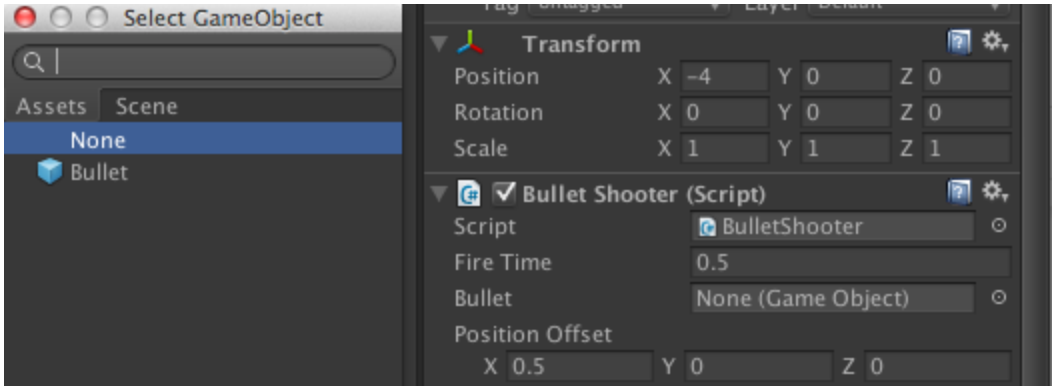
Drag



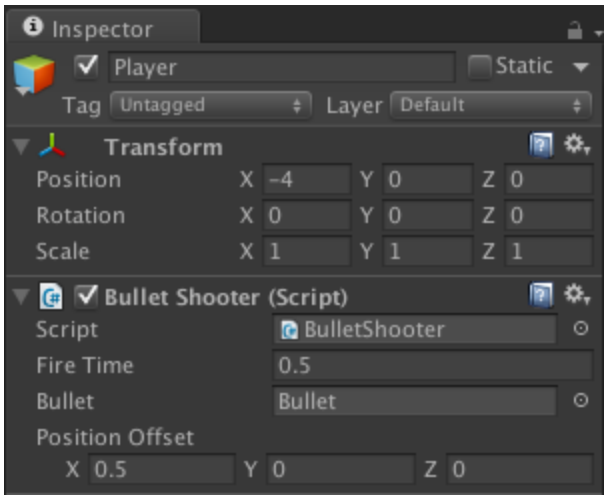
Release!



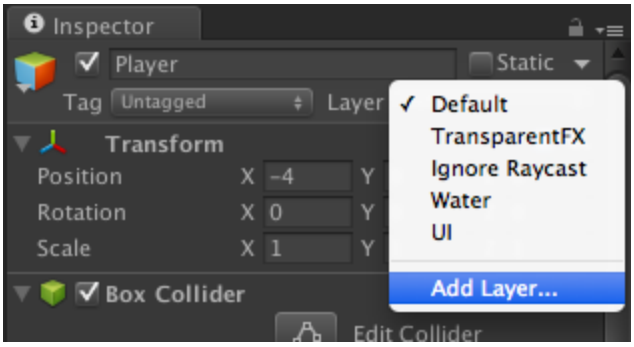
Or click the little circle, and select from the popup window.



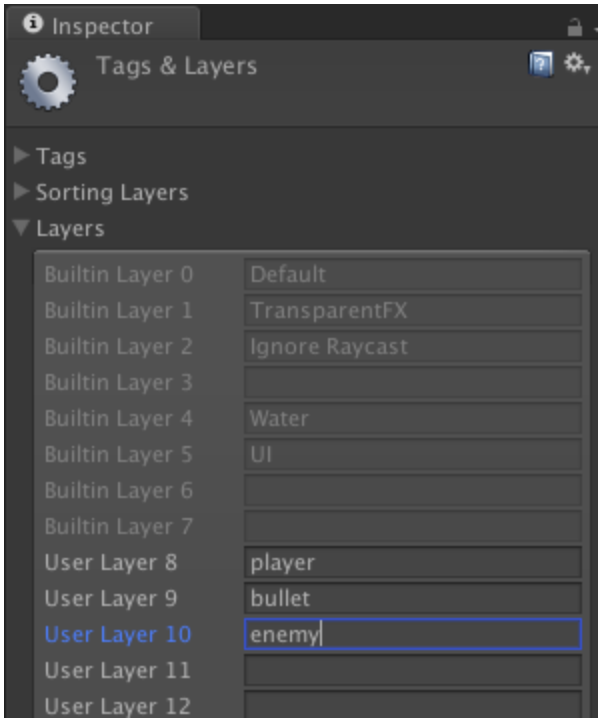
Either way, the bullet parameter should now look like this:



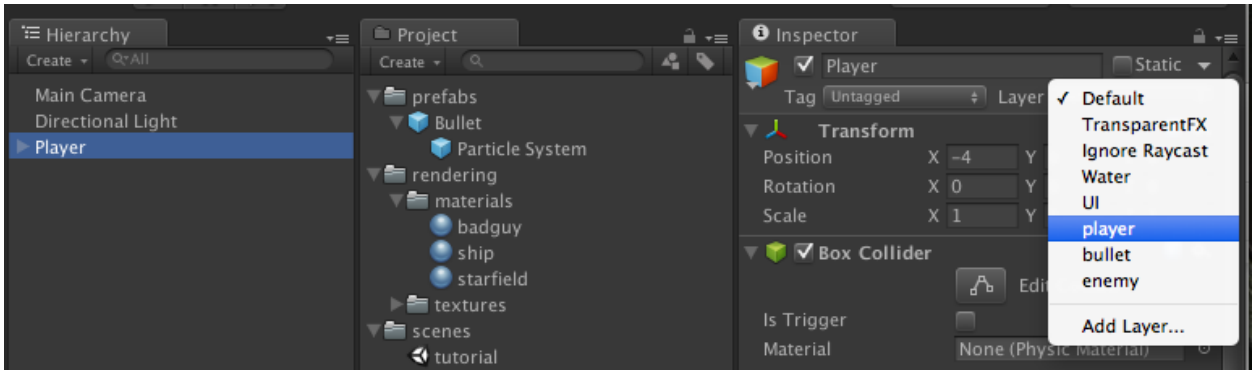
Press the “Play” button now, and when you press the mouse button, your ship should fire bullets! Except it doesn’t! That’s because the bullet is still spawning inside of the ship’s Collider, and then DieOnHit is destroying the object before you can even see it. We could keep tweaking the Position Offset, but there is a better way to make sure that the bullets don’t hit our ship.
In the upper-right hand side of the Inspector, all objects have a Layer value. For all of our objects, it’s currently set to “Default”. Click on the Player’s layer, and then select “Add Layer”.



In the inspector, you’ll see a new display that lists the different layers that exist. All you need to do is add 3, “player”, “bullet”, and “enemy”.

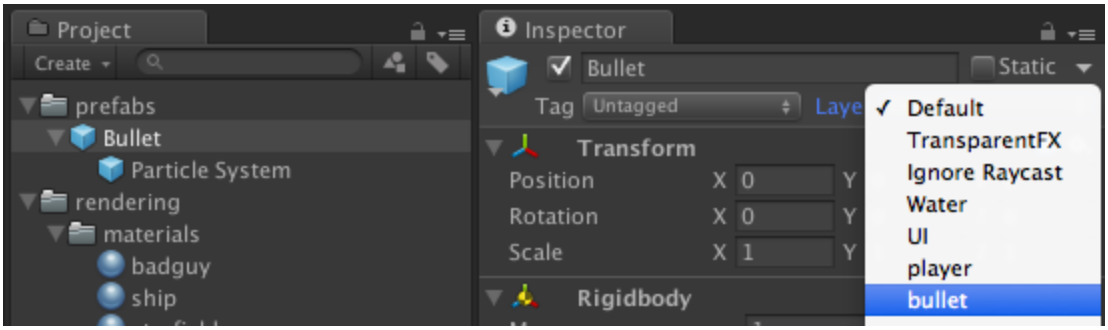


With that done, click on the Player. Select their layer, and set it to the newly-created “player” layer.

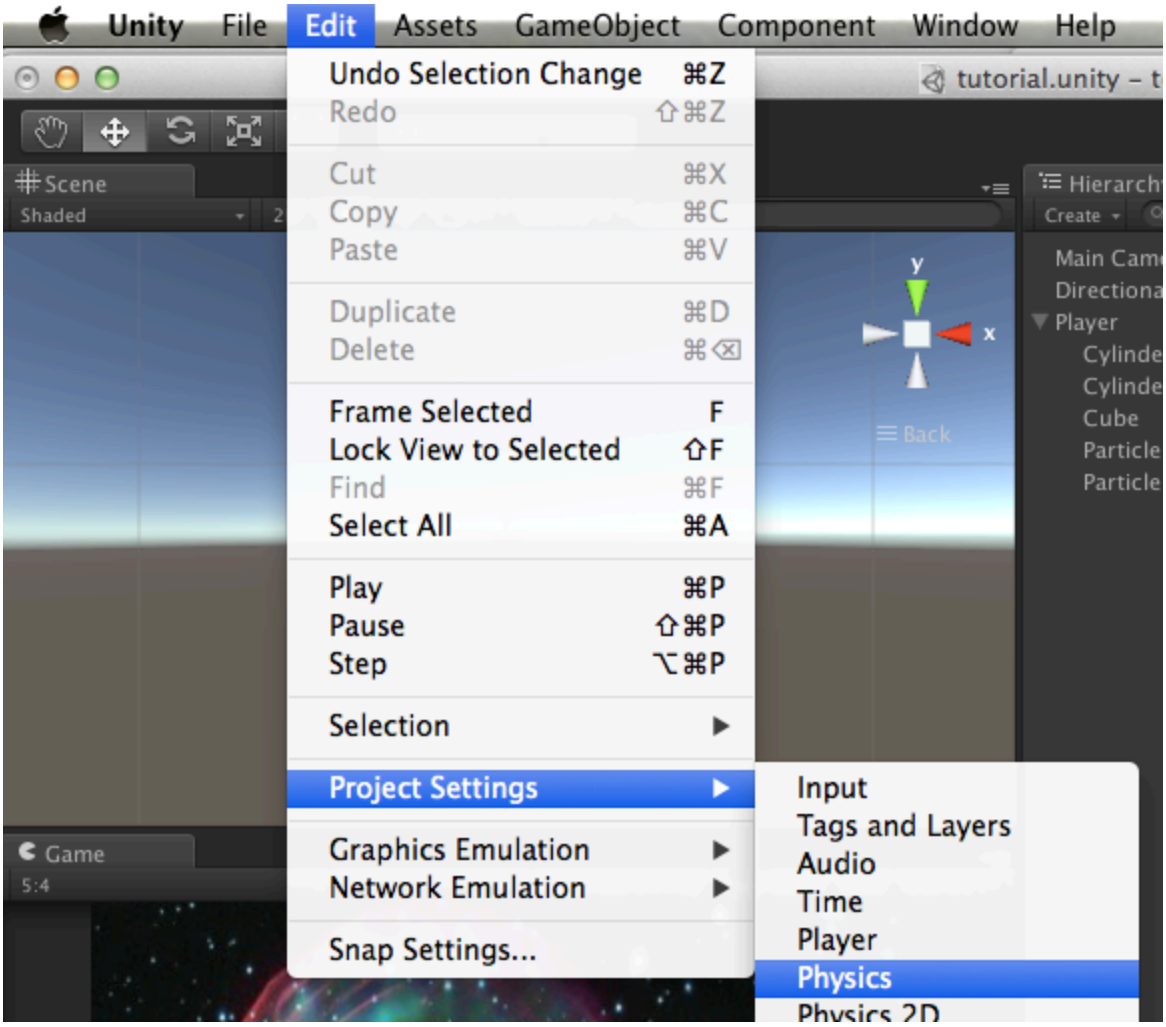


When you do this, a dialog will pop up asking you if you’d like to change the layer of the child objects. In this case, it doesn’t really matter, since none of the child objects have colliders, so go ahead and click “Yes, change children”.

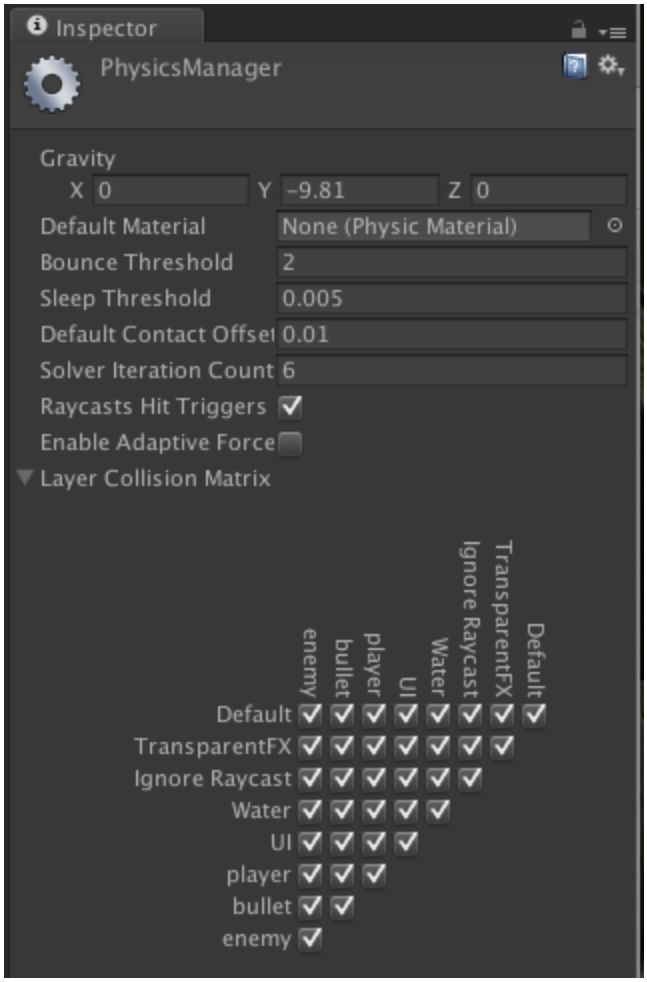
Next, select the Bullet prefab in the Project window, and change its layer to be “bullet”.



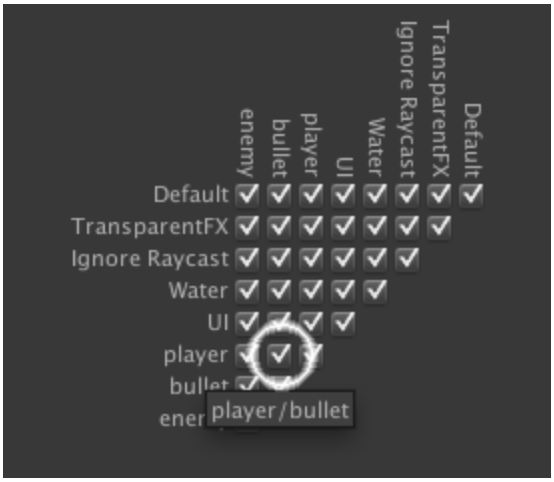
The last step will make sure that bullets and the player never collide. Select the top-level Edit menu, select “Project Settings”, and then “Physics”.



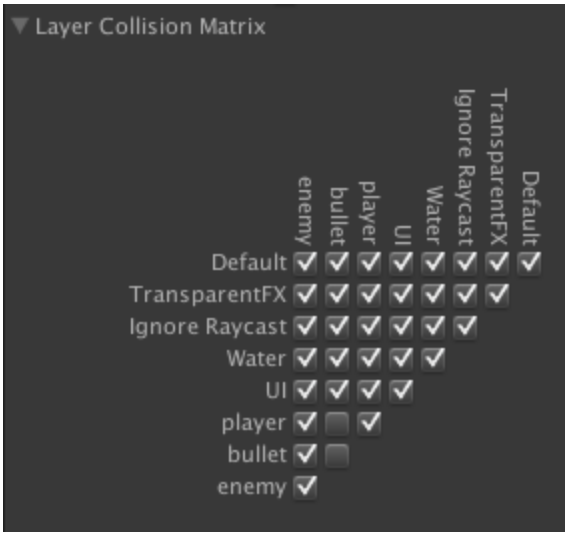
The Inspector window will change, and show you your Physics settings:



The grid of check marks at the bottom is called the “Layer Collision Matrix”. Using this, you can define which layers collide with which other layers. By default, all layers collide with all other layers. However, in the case where you don’t want objects on specific layers to collide (like our ship and bullets), you just have to find the check box for those two layers, and uncheck it.



The checkbox that says player and bullet objects should collide

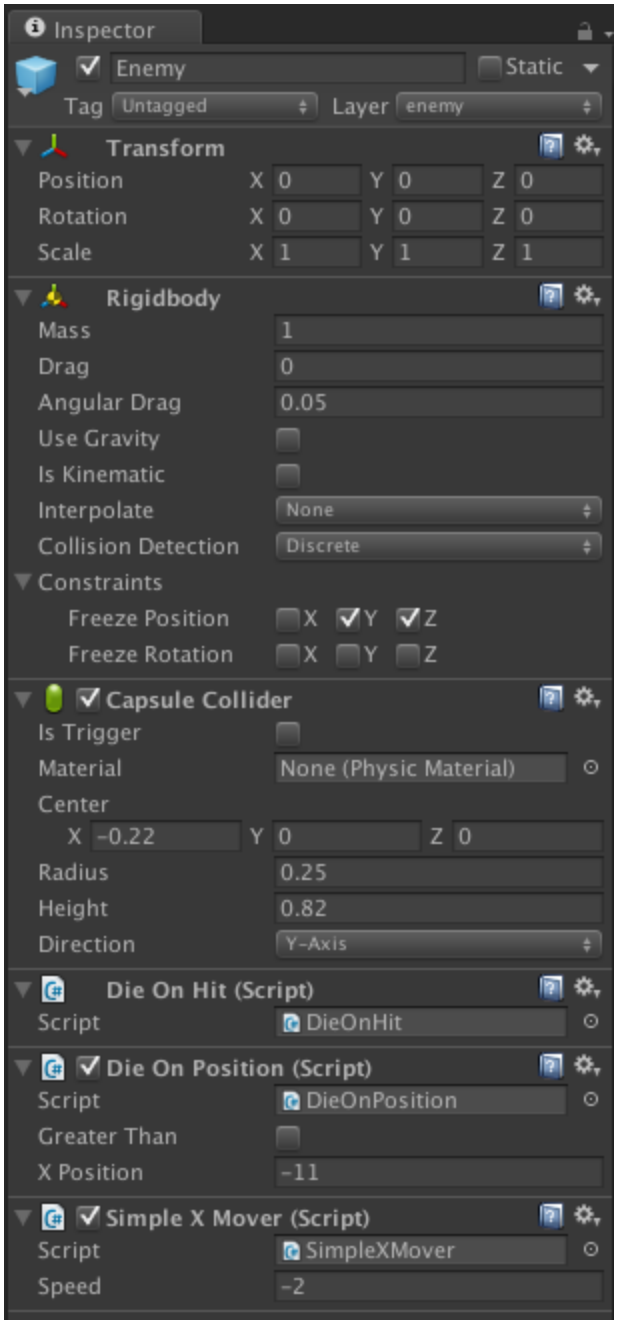


In the image above, we have also unchecked the box that controls bullet/bullet collisions. So bullets won't collide with each other now, either. If you press the "Play" button now, you'll see that bullets fire without any issue now!

Enemies On Scope, Captain

This game has a ship that moves and shoots, but nothing to shoot at! We can fix that using the techniques we've covered.

- Start by creating an empty GameObject, name it "Enemy", set its position to (0, 0, 0), and its layer to "enemy".
- Create a Sphere, and add it as a child of the Enemy object. Remove its collider, and set its position to (0, 0, 0), and its scale to (0.7, 0.5, 0.5).
- Create a Capsule, add it as a child of the Enemy object. Remove its collider, set its position to (-0.25, 0.3, 0), rotation to (0, 0, 90), and scale to (0.2, 0.2, 0.2).
- Duplicate the Capsule, and move its position to (-0.25, 0.3, 0).
- Select the Sphere and two Capsule objects, and set their Mesh Renderer's Material to the "badguy" material.
- Select the Enemy object, and add the following components: Rigidbody, Capsule Collider, Die On Hit, Die On Position, and Simple X Mover. Set the parameters like this:



- Select the Enemy prefab, and drag it into the “prefab” folder, making it a prefab.
- Delete the Enemy in the Hierarchy.
- Add a new Empty GameObject. Name it “EnemySpawner”
- Create a new C# script in the “scripts” folder named “EnemySpawner”.
- Edit the EnemySpawner script to have this code:

```
public class EnemySpawner : MonoBehaviour
{
    public GameObject enemy;
    public float spawnX;
    public float spawnYMin;
    public float spawnYMax;
    public float spawnTime;

    private float _timeToSpawn;

    void Start ()
    {
        _timeToSpawn = 0f;
    }

    void Update ()
    {
        _timeToSpawn -= Time.deltaTime;

        if(_timeToSpawn <= 0f)
        {
            Instantiate(enemy, new Vector3(spawnX, Random.Range(spawnYMin, spawnYMax), 0f), Quaternion.identity);
            _timeToSpawn = spawnTime;
        }
    }
}
```

- Add the EnemySpawner component to the EnemySpawner object in the Hierarchy. Configure the EnemySpawner like this:

With everything set, press the Play button. Now, every 1.5 seconds a new Enemy Instance will be created off the right side of the screen at a random Y position. Since the Enemy has a DieOnHit Component, when your bullets hit them, they will die!

Wrap-Up

In this tutorial, we covered a lot. However, understanding how to use Prefabs, Instantiation and nested object Hierarchies is very important for building more complicated games in Unity. Particle Systems are also important to understand, as they can be used to create interesting effects.

About content creation, I thought that autodesk Maya or Max is a better option to go the professional way but doing it from scratch takes a long time. Therefore there is a second way to get content is the Unity Assets Store.