Unity Tutorial

By Arthur Baney

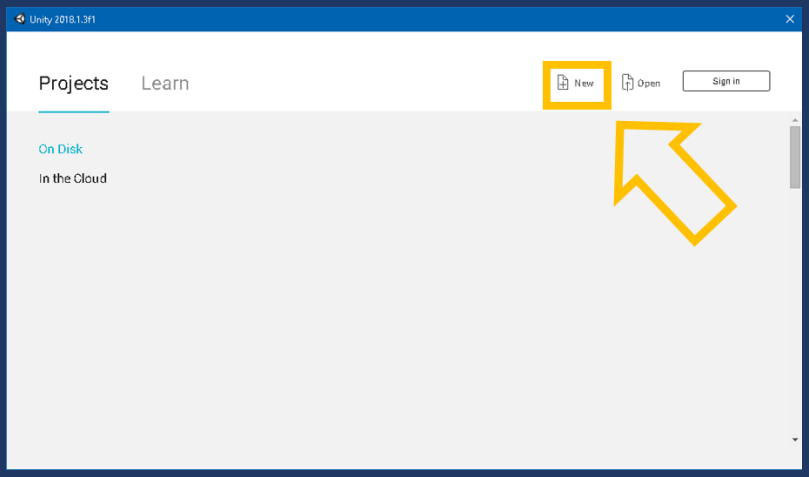
In this tutorial we will walk through how to create a basic game in Unity where you play as an x-wing starfighter and shoot lasers at attacking tie fighters.

By the end of the tutorial, you will have learned how to:

1. Setup, create, and manipulate game objects in the Unity scene editor
2. Add scriptable components to game objects to make them behave how you want them to behave
3. Write some basic C# scripts to control the game
4. Create some basic user interface (UI) with buttons and text labels using Unity's built in UI Canvas system

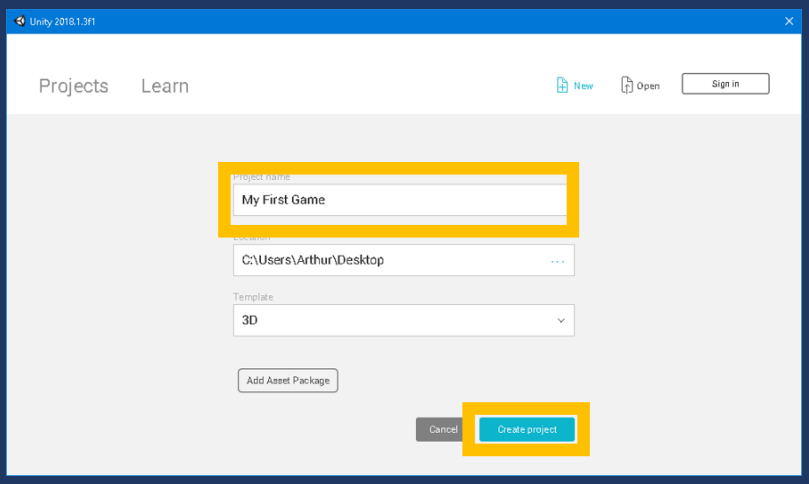
This tutorial will be a fun starting point for you to learn how to create your own 3D apps and games using Unity.

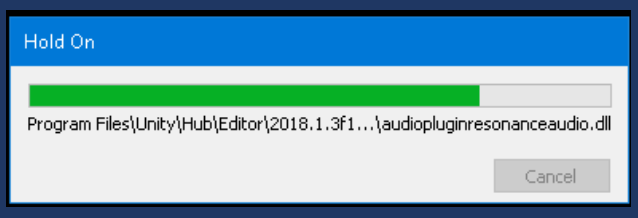
When you first open Unity, you should see a screen similar to this:



Click the “new” button in the upper right corner of the window.

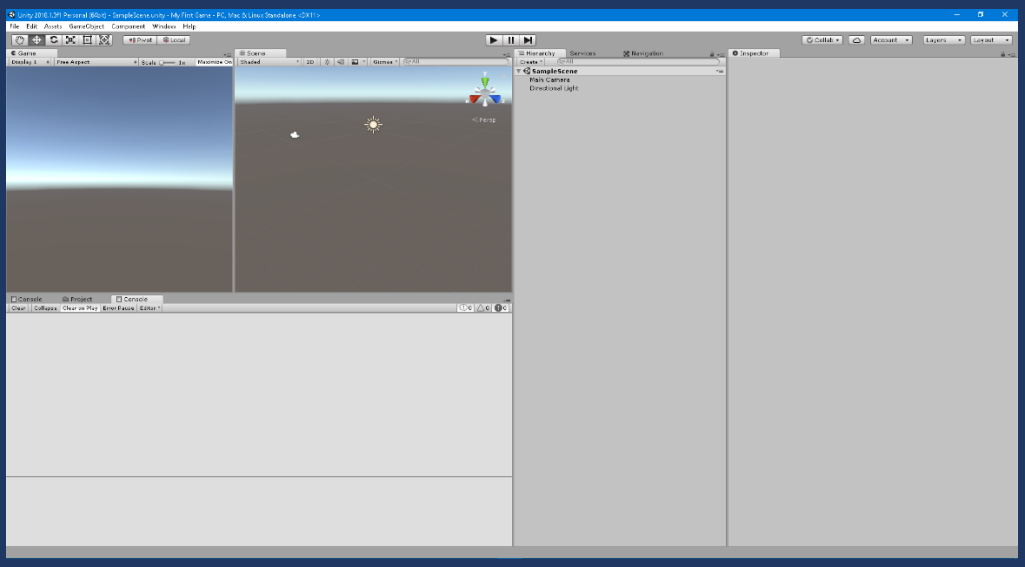
Enter a project name, like “My First Game”, and then click “Create project”



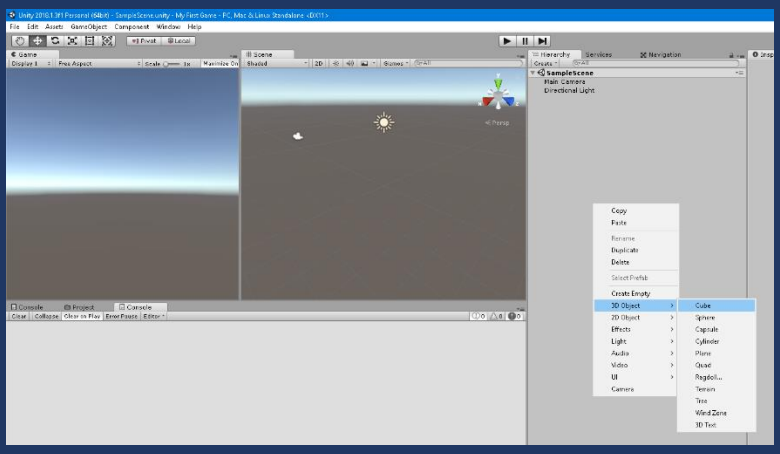


Wait a few seconds as Unity creates your project files…

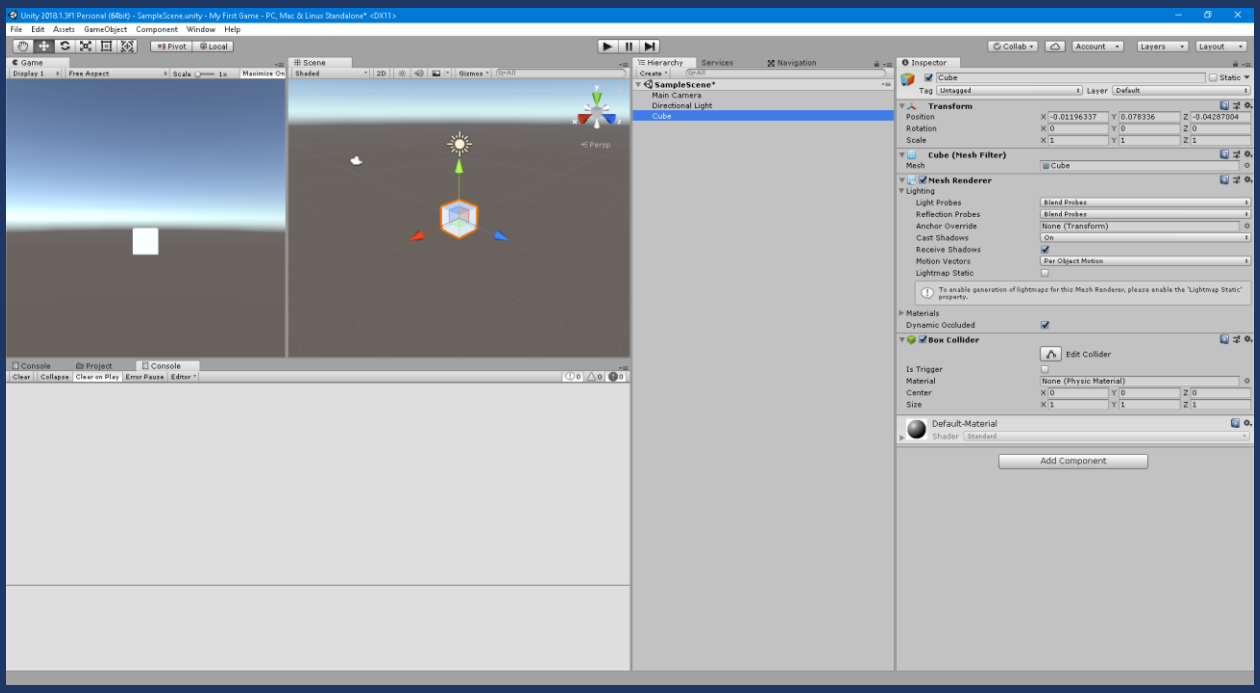
Congratulations, you have created your first blank Unity project!



This is the Unity Editor, where you will be able to do a lot of your game's development, such as importing artwork, sound effects, code, and other game asset files, as well as position game objects in the virtual world, design your user interfaces, attach behavior components and scripts to game objects, and an indefinitely long list of other awesome things which will allow you to create the game of your dreams.



Right now we just have an empty game world, which is pretty exciting, because it's like a blank canvas for us to start creating our dream game on. The possibilities are limitless! Let's try adding something to the scene. Right click in the area under the "Hierarchy" tab and under the 3D Object submenu click "Cube".



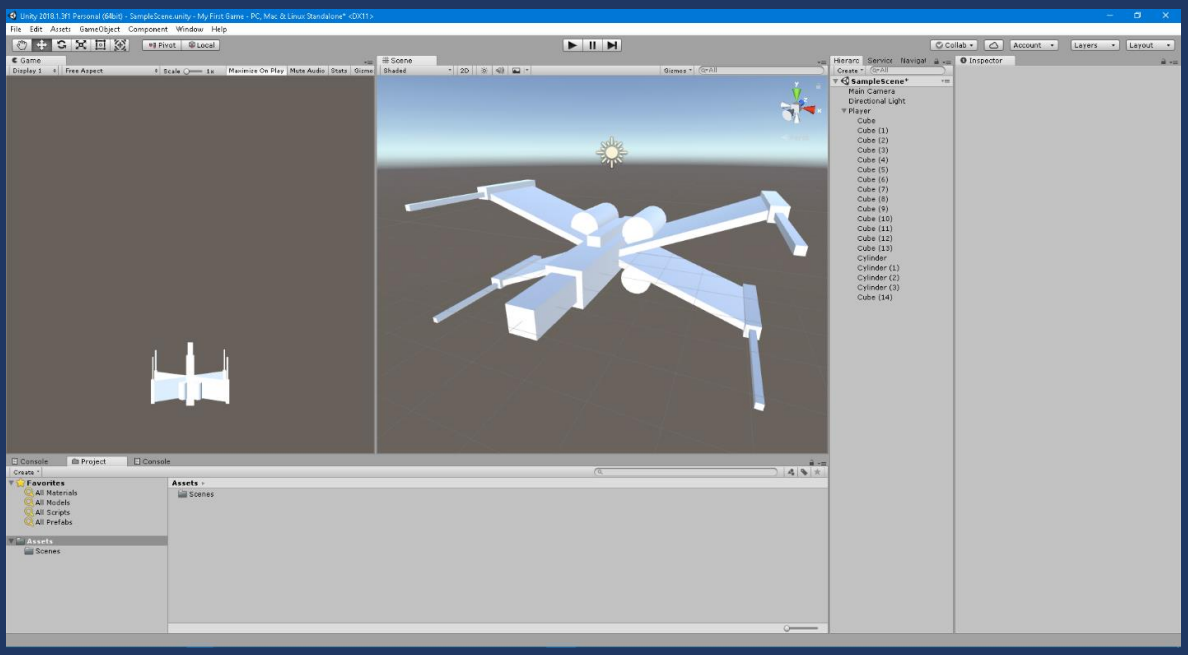
You should now see a cube appear in the scene.

Let’s build an x-wing starfighter out of cubes and cylinders. Here is a picture of one for reference:

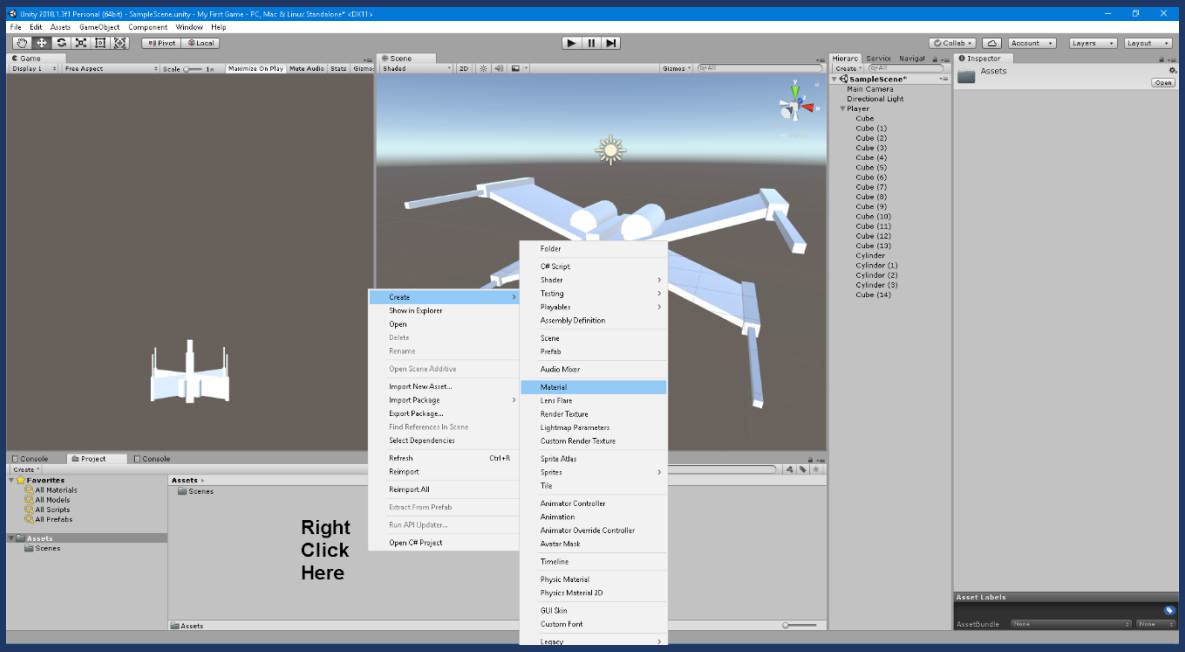
Use the move tool (W), rotate tool (E), and scale tool (R) to create a x-wing fighter model. Use ctrl + D to duplicate objects.

Right click to rotate the view around, and use the middle mouse button to move the view around.

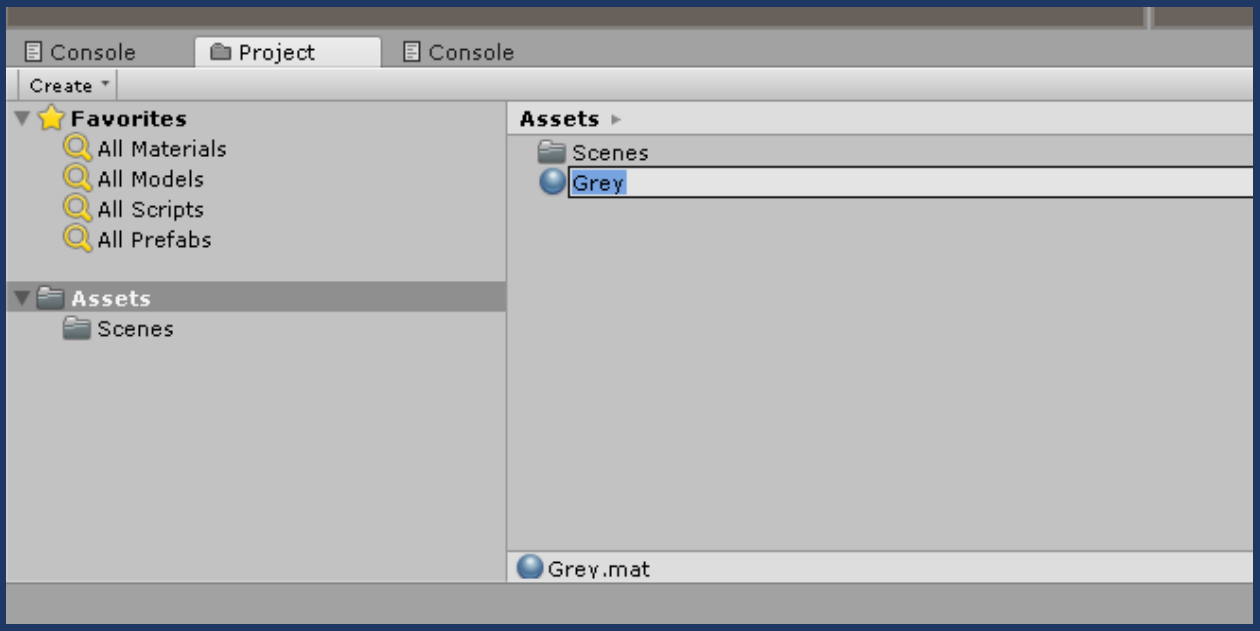
It should look something like this when you are done:



Now that we have the geometry in place, we can start coloring it by adding things called “materials”. To create a material in Unity, right click in the Assets tab and click “Create > Material”.

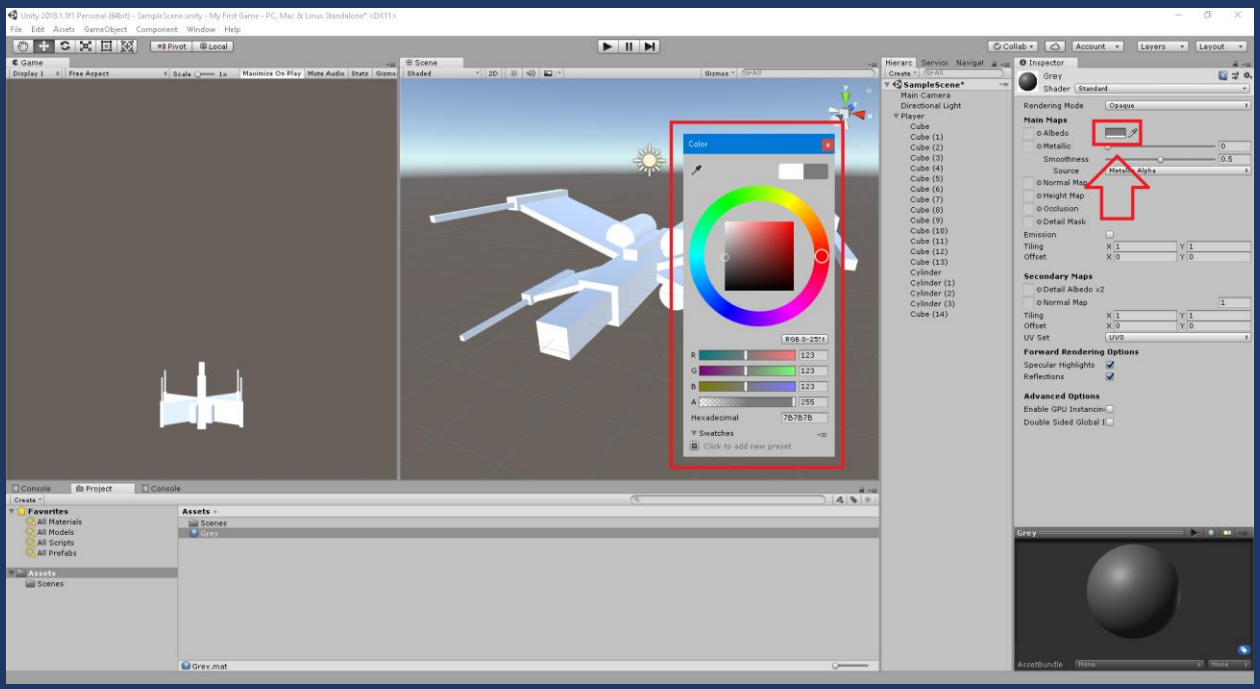


After clicking Create > Material, it will create a material in the assets folder and prompt you to enter a name for it.



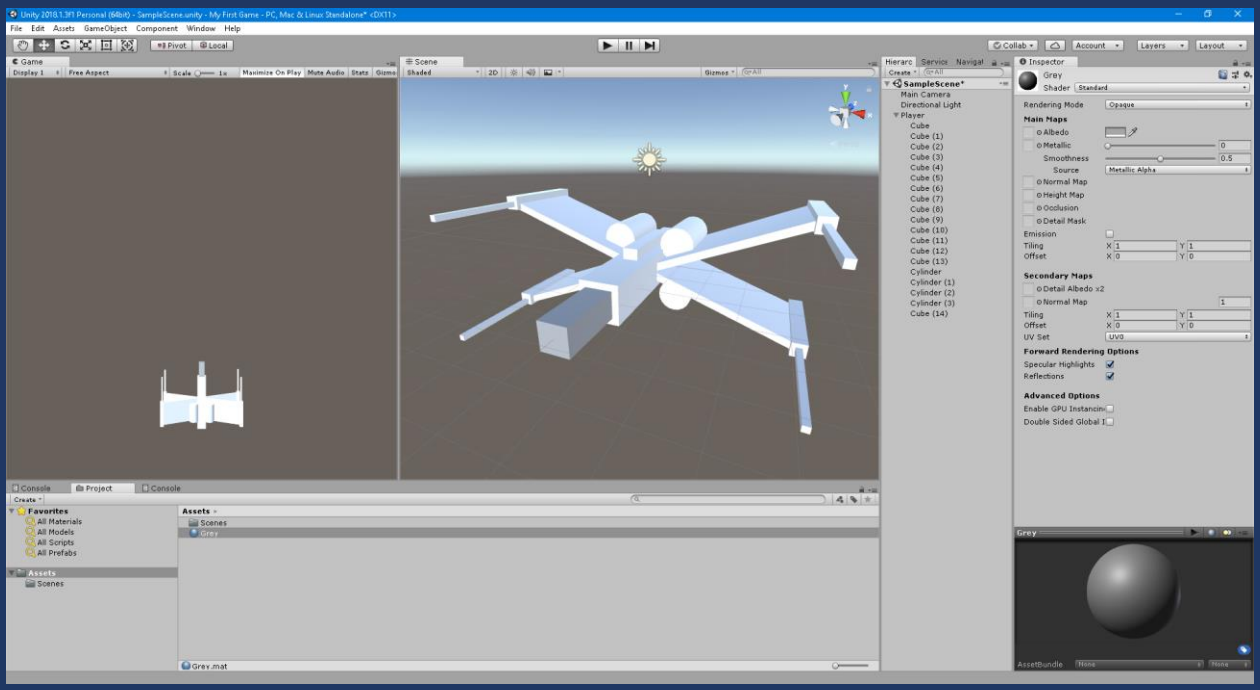
Let's call this first material "Grey" and use it to color the body of our x-wing. You can always rename the material by clicking on it and then pressing <f2> on windows or <return> on mac.

Let's set the color of the material in the Inspector tab by clicking on the color chooser button labeled "Albedo":

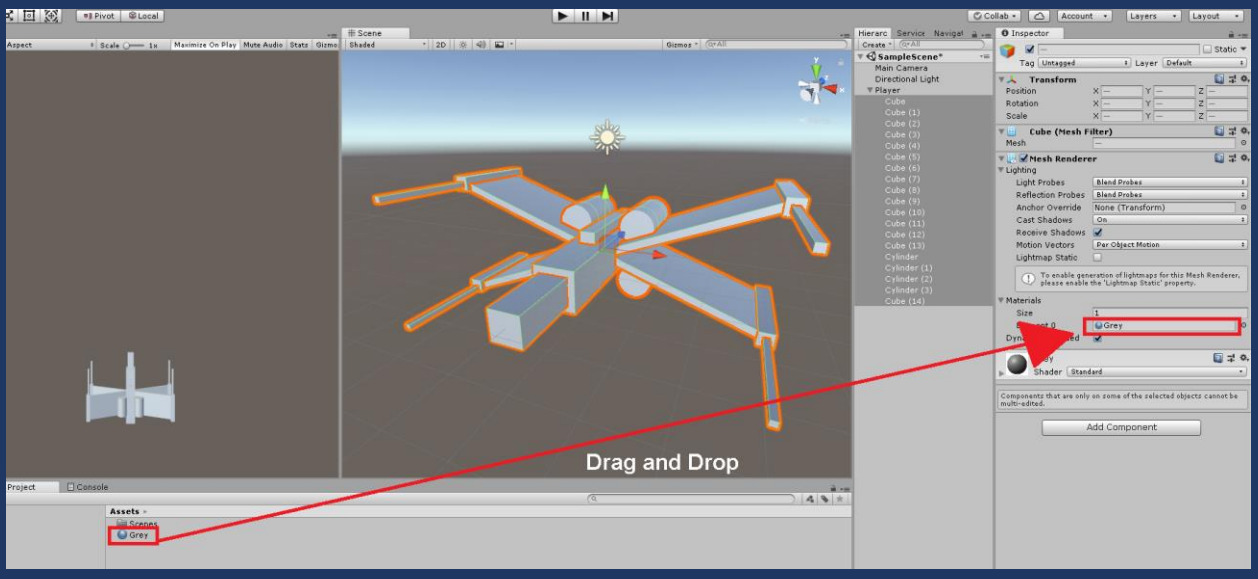


Choose a greyish color similar to the wing color of the x-wing, and then hit enter / return to save the color choice.

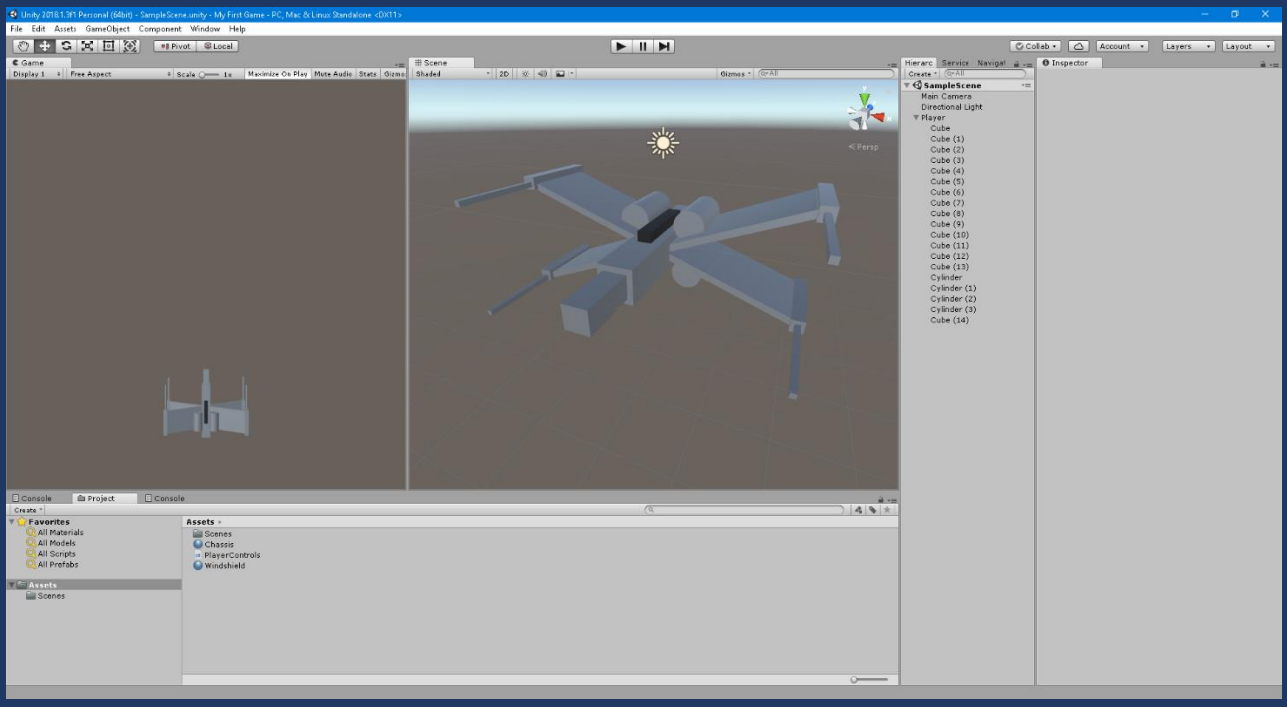
You can now drag and drop the material from the assets tab onto objects in the scene and see how they change color:



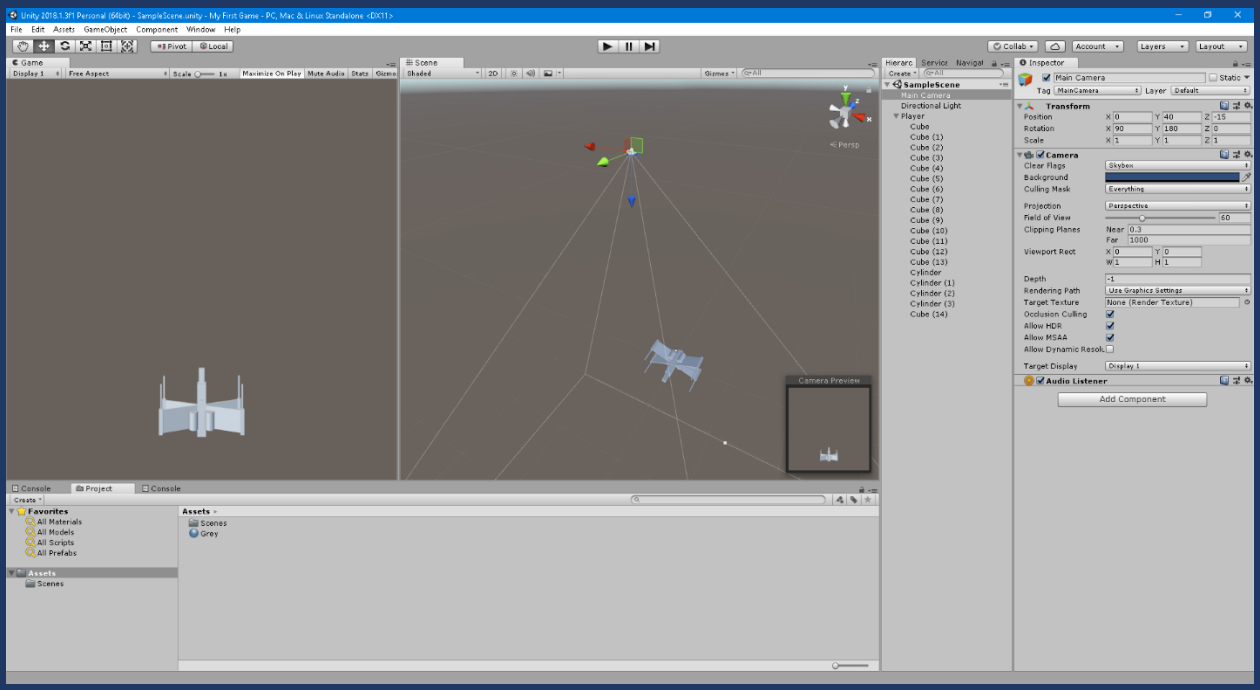
You can set the material for multiple objects at a time by selecting all the objects you want to set the material for in the Inspector tab (use ctrl + click on windows to select multiple, or command + click on mac), and then dragging and dropping the material from the assets tab into the materials slot in the inspector tab.



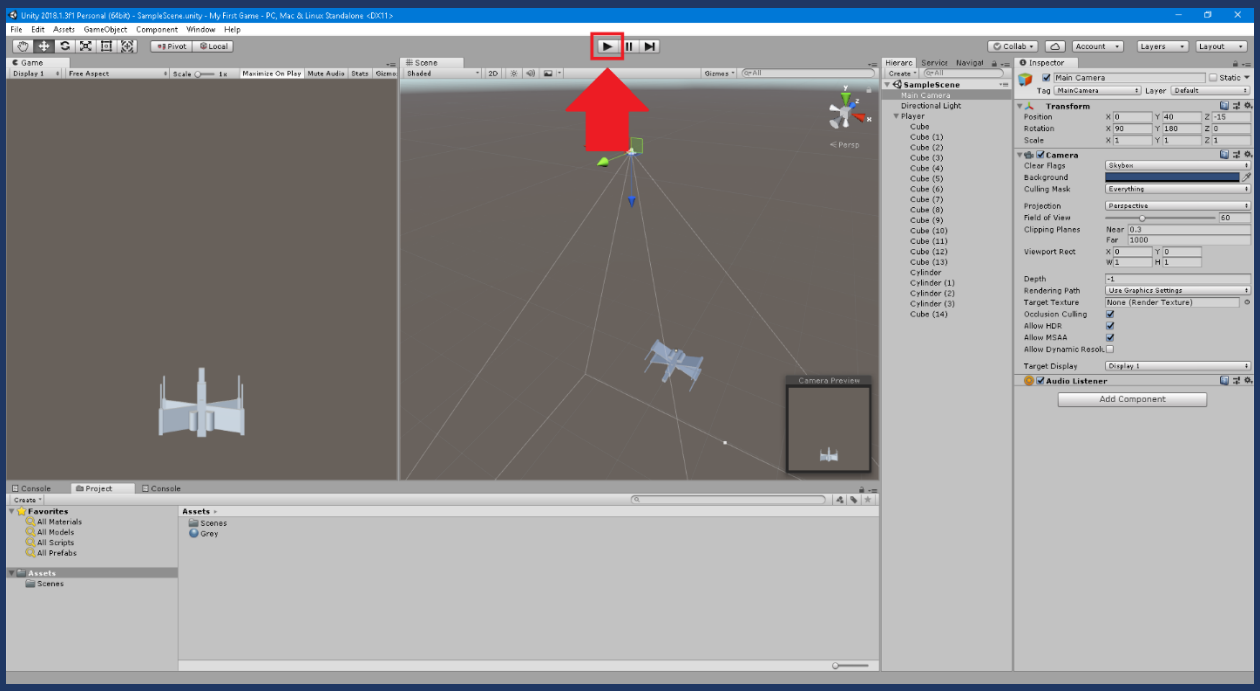
Try making your x-wing look like this: (or color it however you want to!)



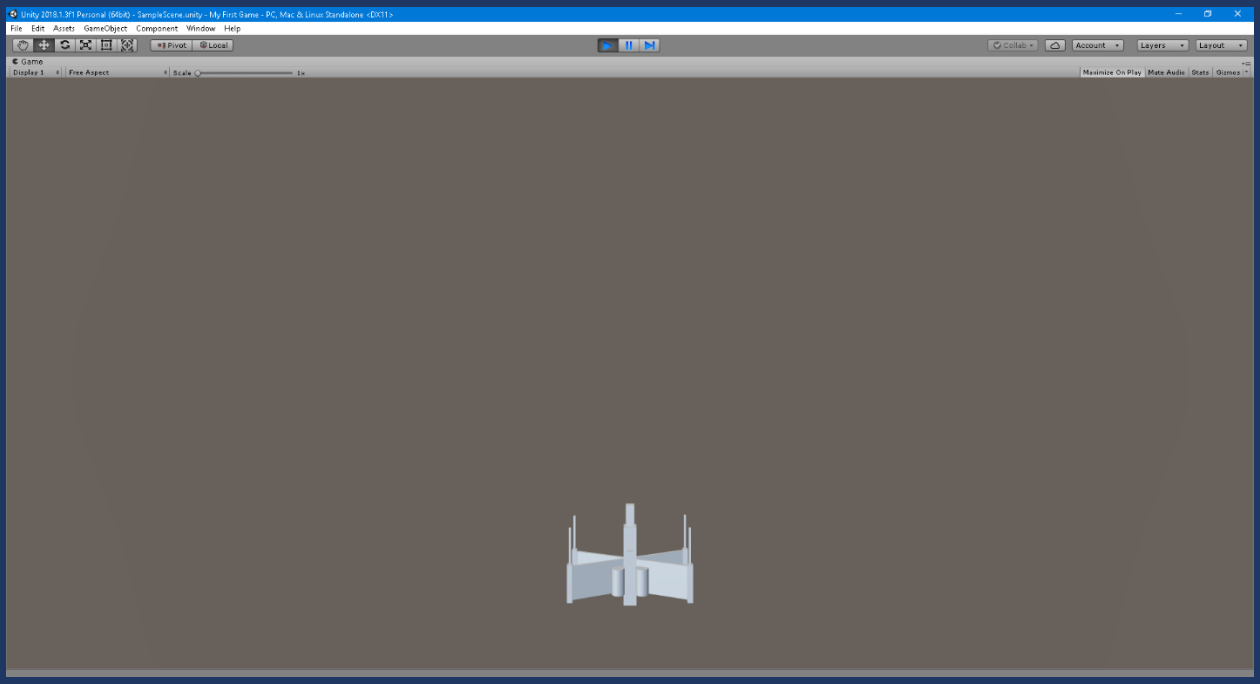
Position your camera object so that it is above the x-wing looking down, and rotate it such that the x-wing is pointing upwards on the screen in the “Game” view.



Great! Now we have a player character, so let's try playing the game! To run the game, press the "Play"  button at the center top of the editor.

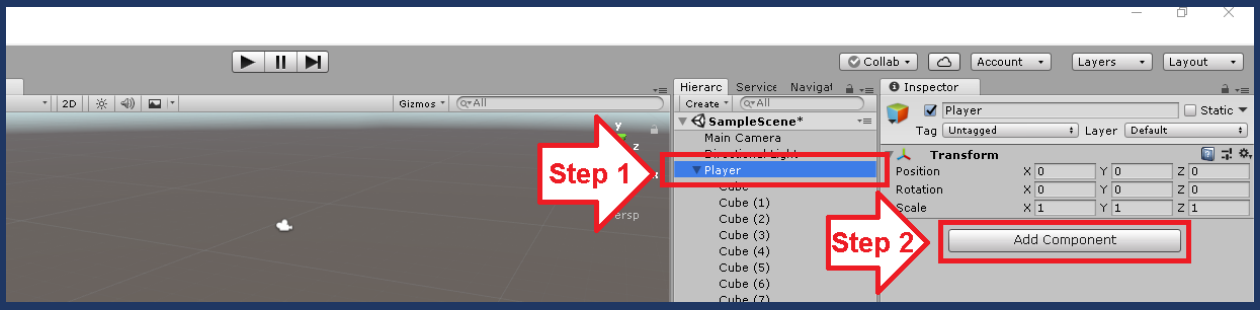


After clicking the play button, you should see a game view like this:

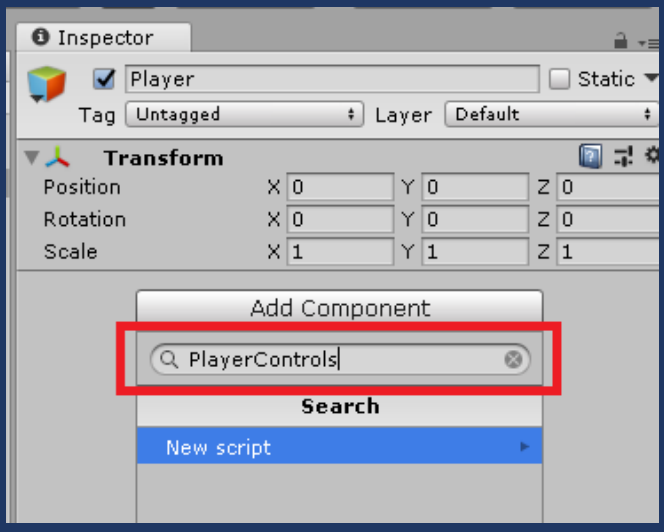


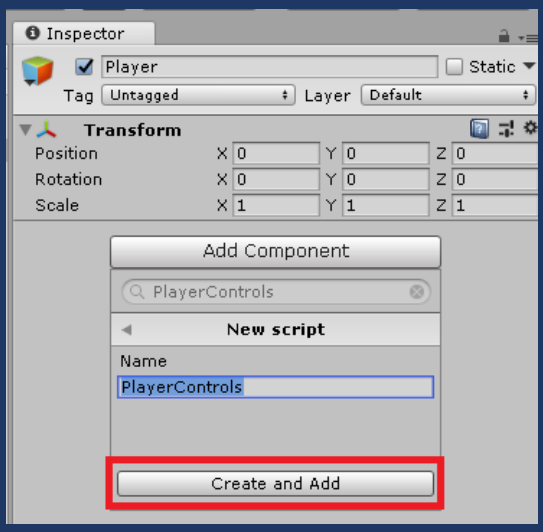
Cool! We now have an empty game with a player character x-wing starfighter. We can't move the x-wing yet, because we haven't programmed any controls for it yet. So let's do that next.

Click the play  button again to exit the game and return to the Unity Editor, then select your Player object in the scene Hierarchy tab, and click the “Add Component” button.



In the input that pops up, type “PlayerControls” and then click “New script” and then “Create and Add”

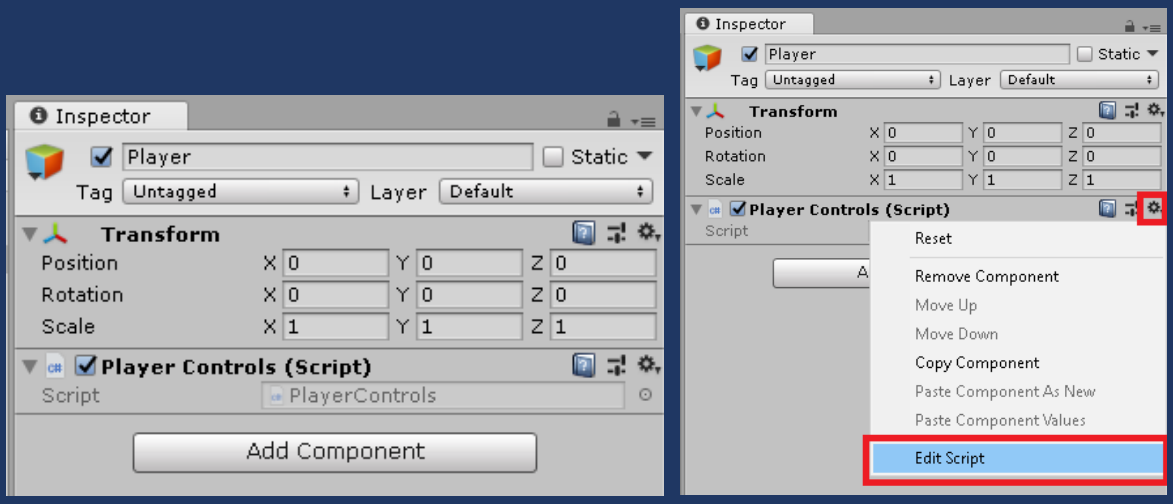




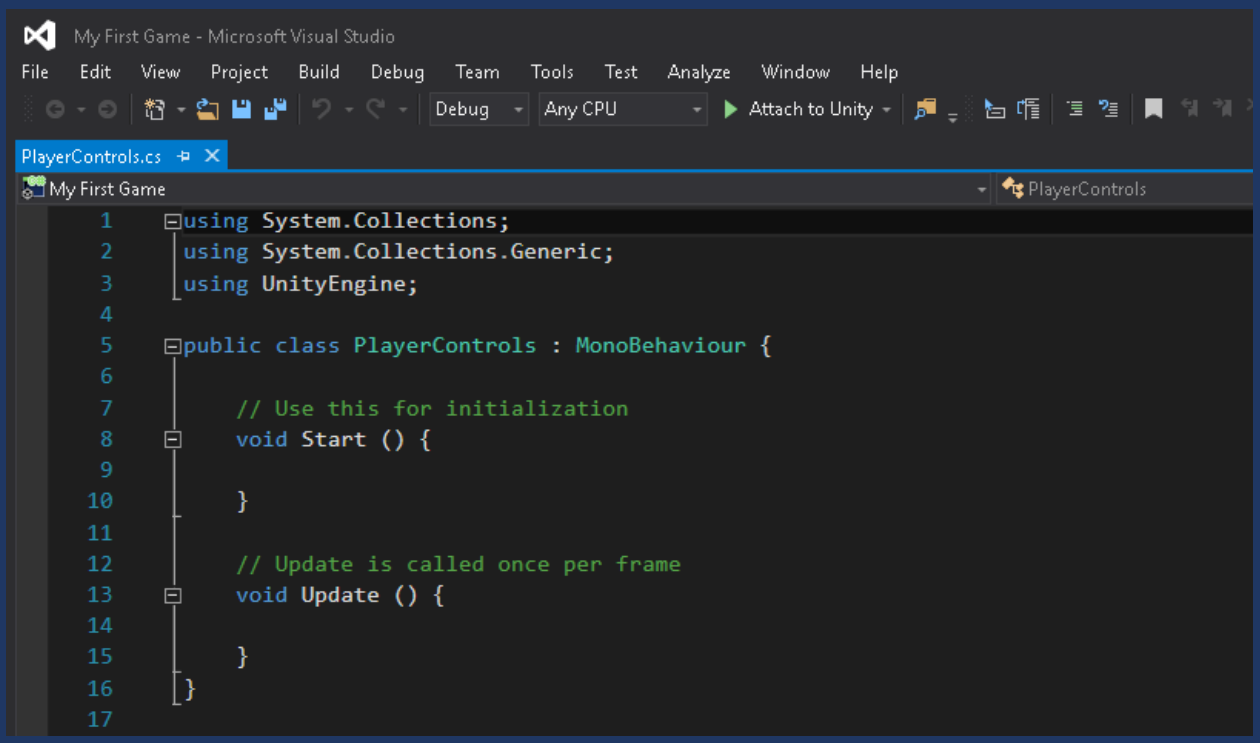
After this you will see a new “Player Controls” component is created and attached to your Player game object.

Components in Unity are scriptable behaviors that you can attach to any game object. You can use these scripts to control every aspect of the game, like what the players controls are, how the AI players behave, what the win / lose conditions are, what the rules of the game are, and pretty much every other aspect of the game.

To edit the Player Controls script, click on the gear icon in the top right corner of the component, and then click “Edit Script”.



The player controls script should now open in a code editor:

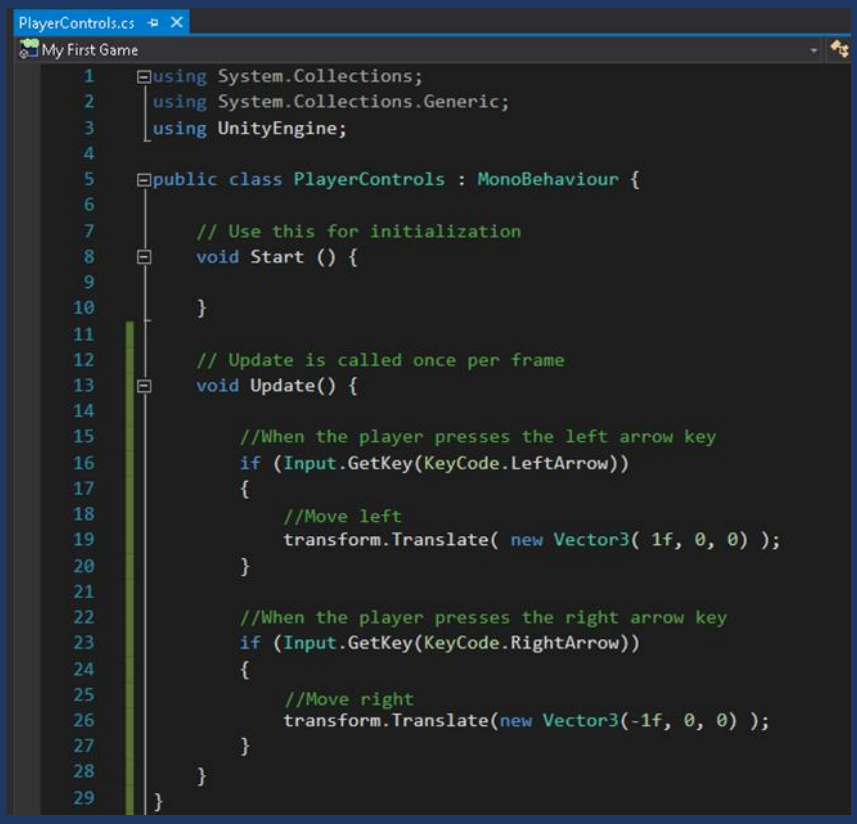


Unity uses C#, a great object oriented language for game development which is easy to learn for anyone who is already familiar with Java, C, or C++, or any similar language. By default, every component script you create will have a Start method and an Update method. The Start method code will be executed the instant the game object spawns in the game world, and can be used to initialize variables, setup scenes, and do a lot of other logic that only needs to be done once when the object spawns in the scene. The Update method is run continuously throughout the duration of the game for as long as the game object is still enabled in the scene.

The Update method is useful for scripting your player controls, in game logic, animations, AI behaviors, and any other code that needs to be run continuously throughout the duration of the objects lifecycle in the game world.

By default, every component script inherits behavior from the class “MonoBehaviour”, which is a script that describes all the common functionality of every Unity component. Every component that you want to attach to a game object in the Unity editor should inherit from MonoBehaviour in order for it to work in the editor. MonoBehaviours also have a bunch of other useful common attributes for game development which we will discover later.

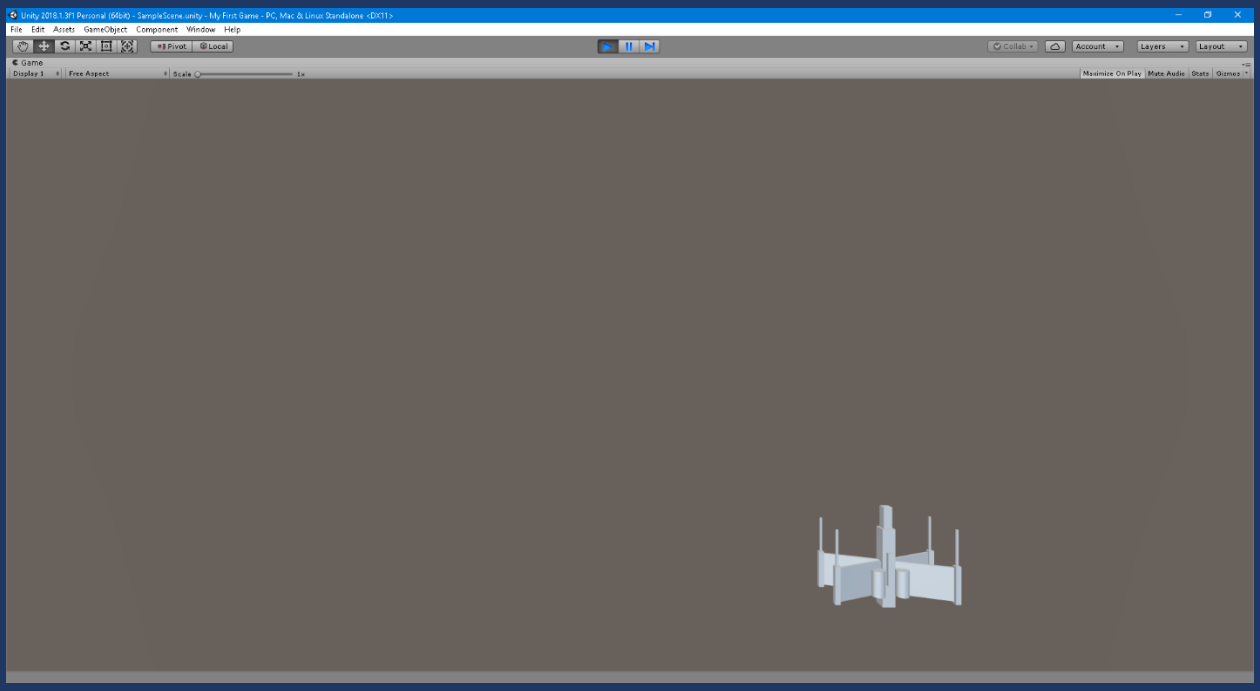
Let’s start by adding some keyboard controls to the Update method. When the user presses the left arrow key, let’s make the player move left, and when they press the right arrow key, let’s make the player move right:



To do this, we use Input.GetKey(keycode) to detect when the left and right arrow keys are pressed, and then “translate” (a fancy word for “move”) our game object’s “transform”, which is an object attached to every game object that stores information about it’s position, scale, rotation, and a bunch of other useful information. When we call a transform’s “Translate” method, it changes the position of that transform by however much we indicate in the parameter for the method.

For example, transform.Translate(new Vector3(0 , 1 , 2)); would shift the position of our game object by 0 on the x-axis, 1 on the y-axis, and 2 on the z-axis. A Vector3 is a data structure that is useful for storing 3D position information – such as the x, y, and z coordinates of our object in the game world.

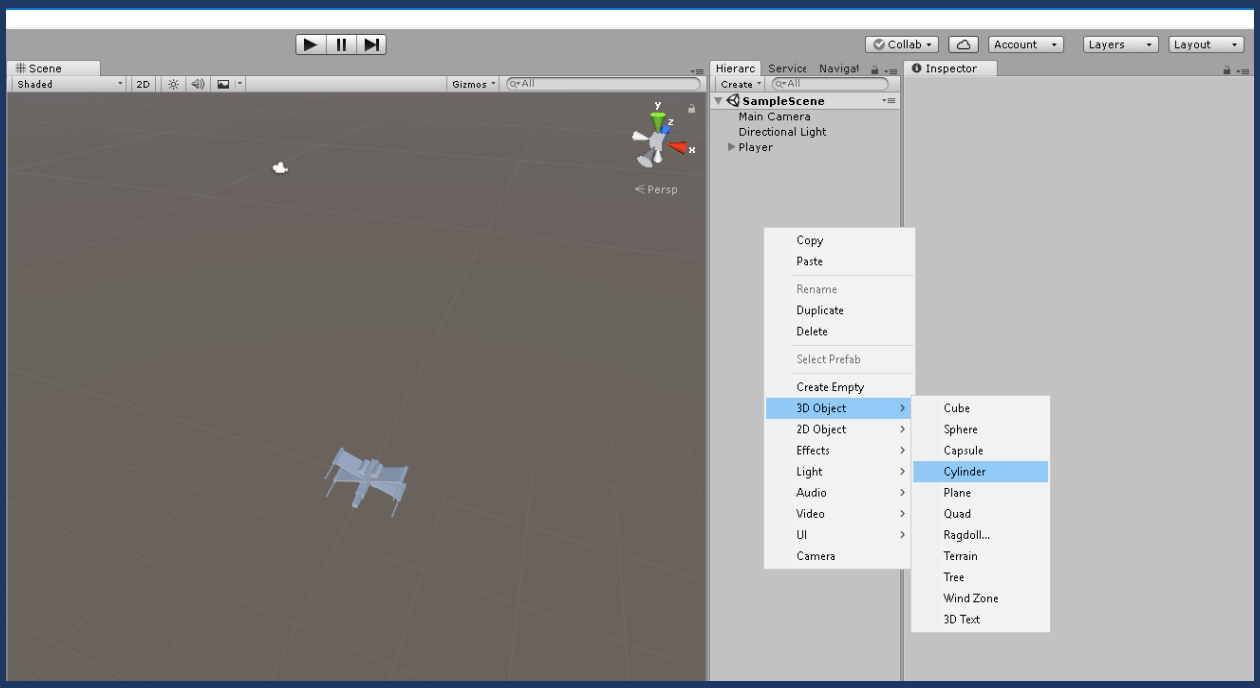
If we save the script (ctrl + S on windows, command + S on mac) and go back to Unity and run the game now, we should see that we can now move our x-wing starfighter using the left and right arrow keys.

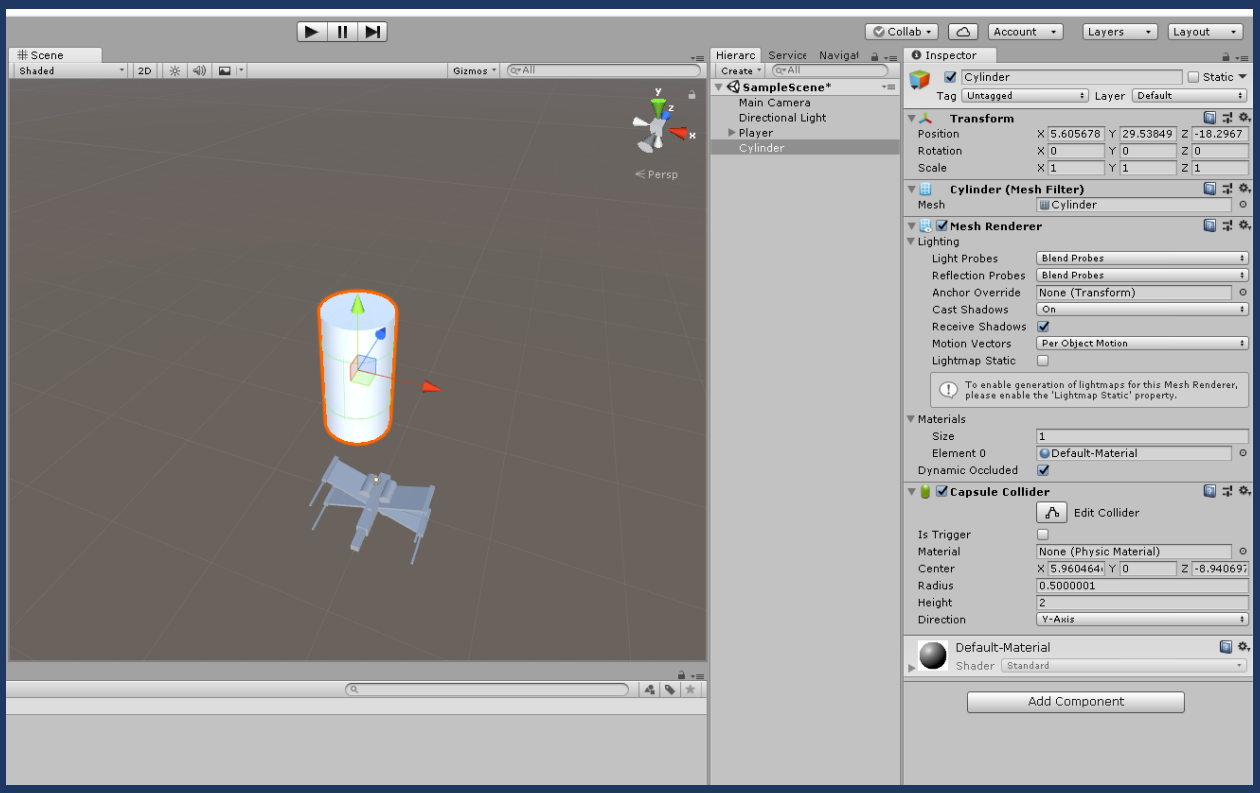


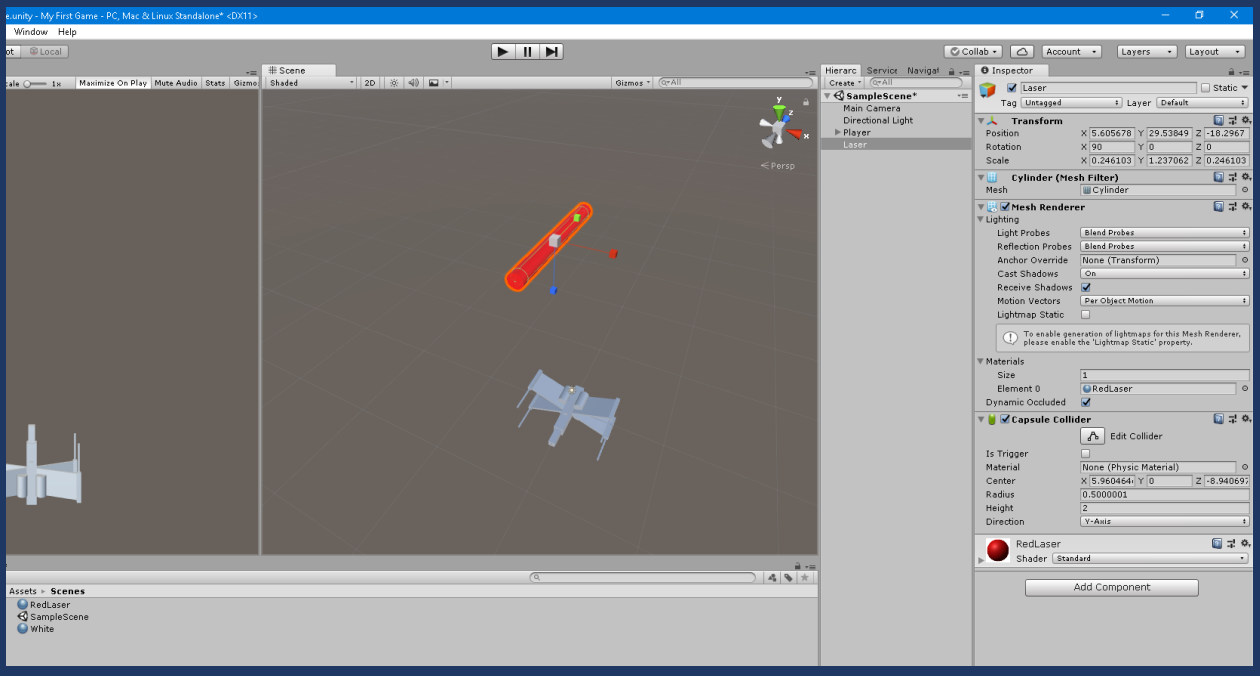
Now that we can move our x-wing, we want to make it shoot lasers.

To do this, first we will have to go back to the Unity Editor and create a “Laser” game object.

Create a cylinder and apply a red material to it, and make it shaped like a laser blast:



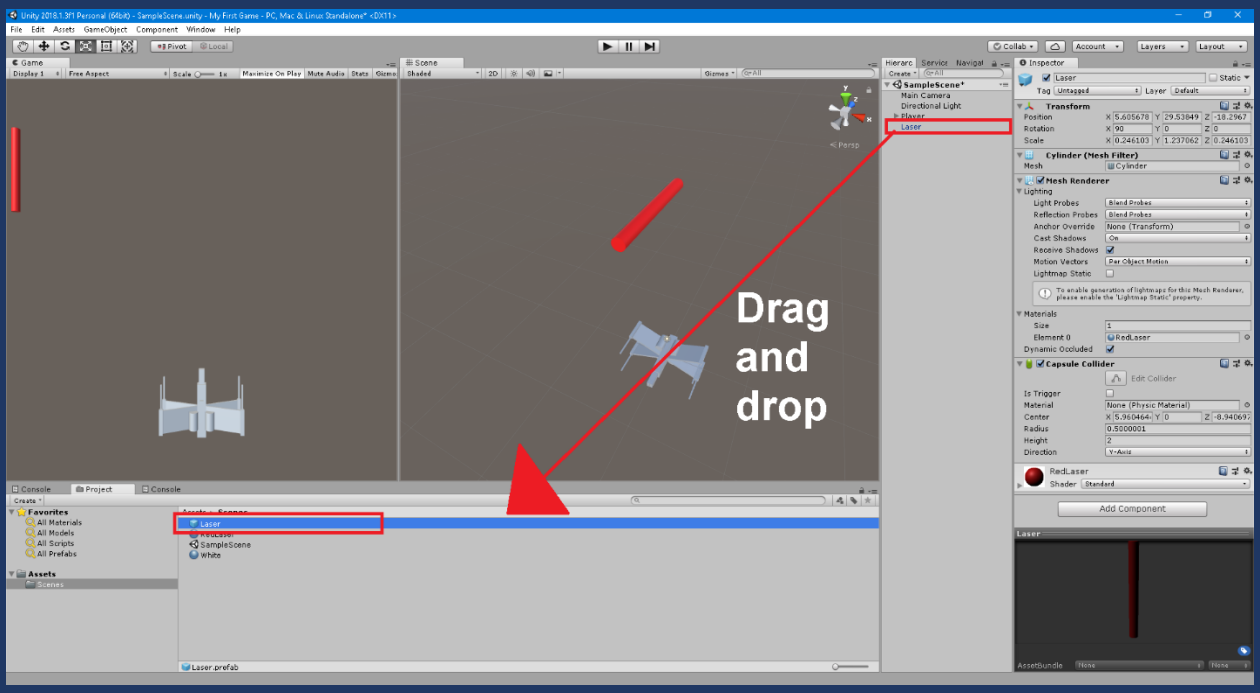




Name this new game object “Laser”.

Now, our laser game object is slightly different than our player object, in that we don’t want it to exist when the scene starts, and we don’t want it to be created until the player presses the “shoot” button.

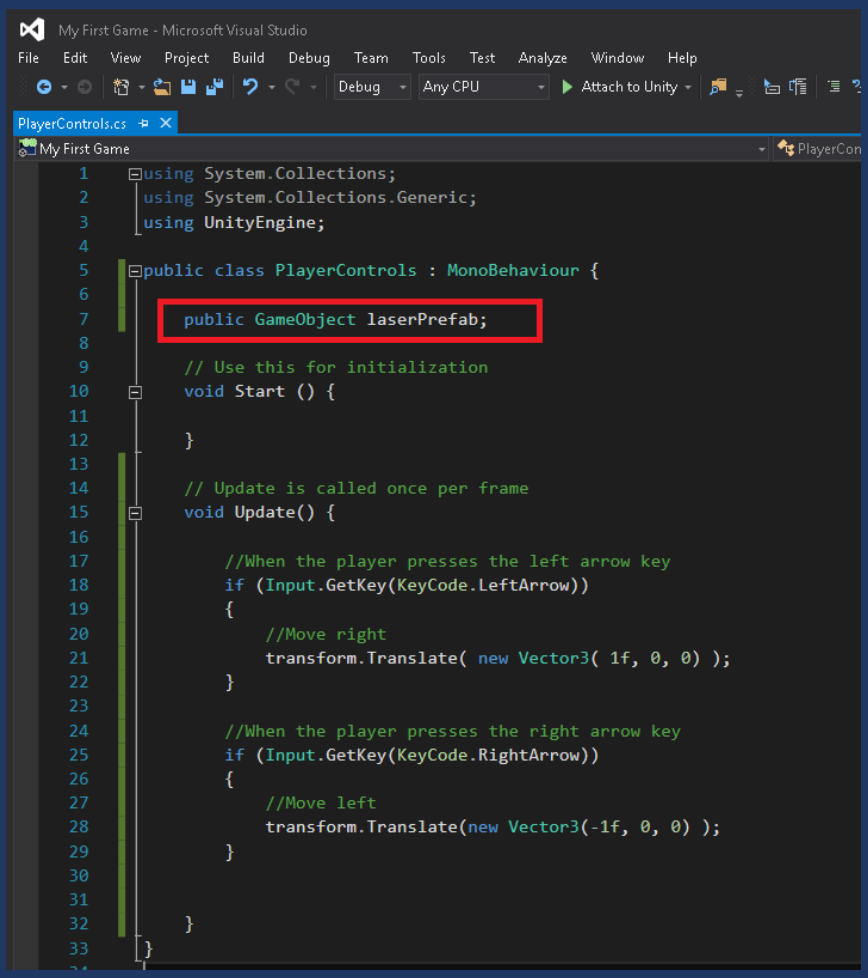
To do this, we don't want to have the game object in the scene when the game starts, so instead, we can use what are called "Prefabs". A prefab is a saved game object that we can create at any point during the game and doesn't need to be in the scene when the game starts. To turn the laser game object into a prefab, drag and drop it into the Assets tab like so:



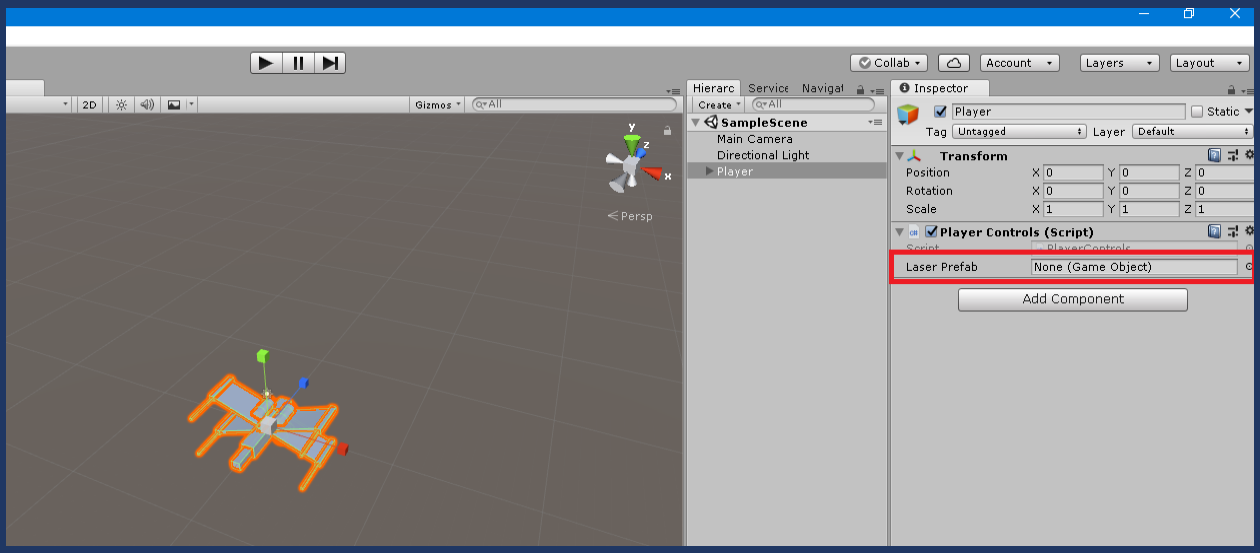
You will now see a "Laser" prefab show up in the Assets folder. You can now delete the laser game object from the scene, as we don't want it to be there when the game starts.

In order to make it so the player can shoot, let's go back to our PlayerControls script.

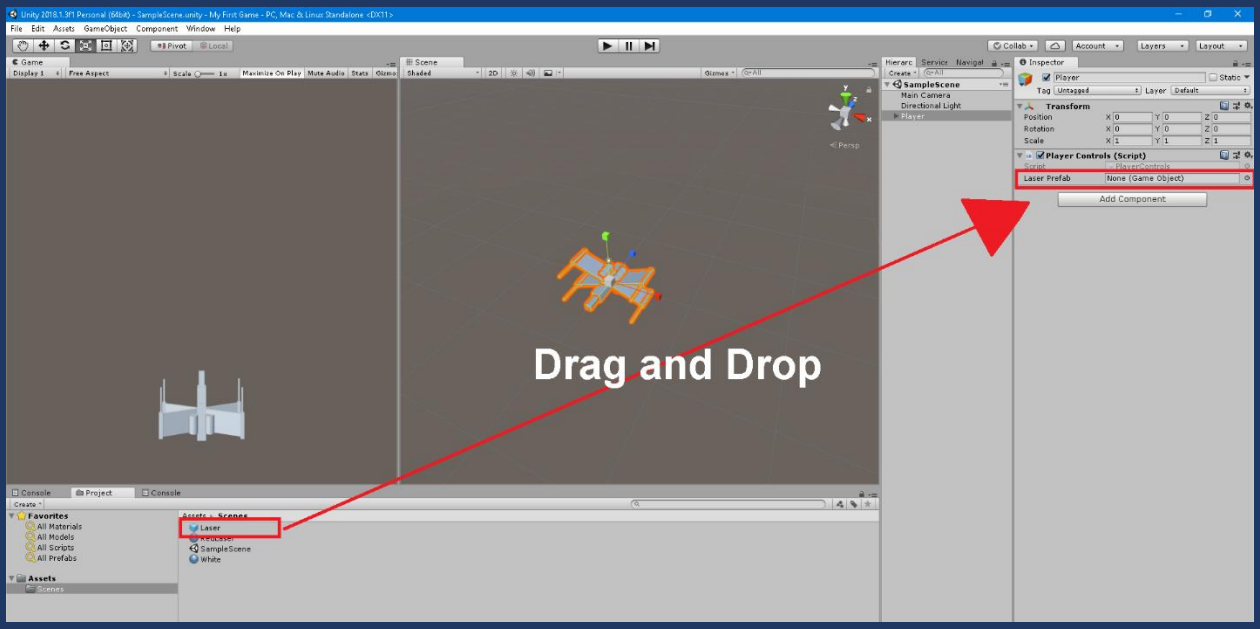
First, we need to add a variable to store the laser prefab.



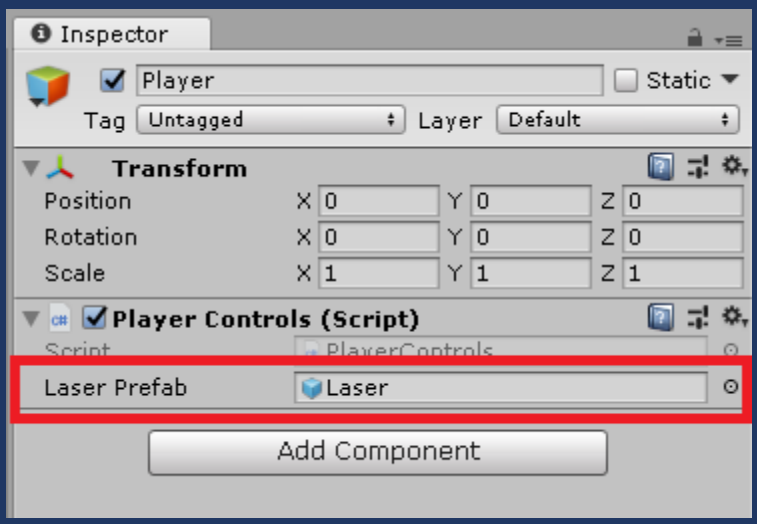
If we save this and go back to Unity, we should now see a slot appear under the Player Controls component in our player game object that says “Laser Prefab” like so:



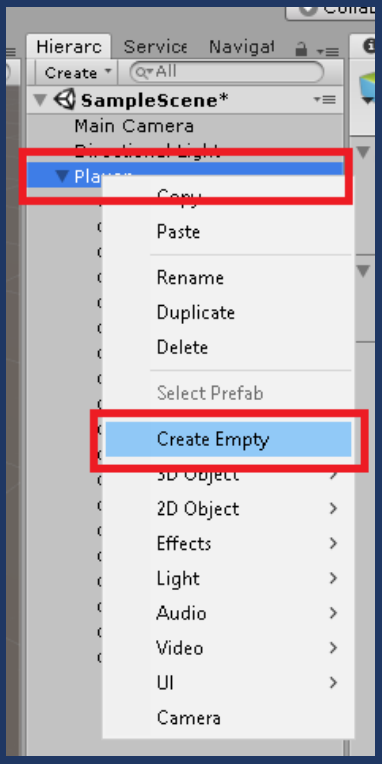
We can now drag and drop the laser prefab we created earlier into this slot so that we can use it in the script.



You should now see the Laser prefab linked to in the Player Controls component like so:

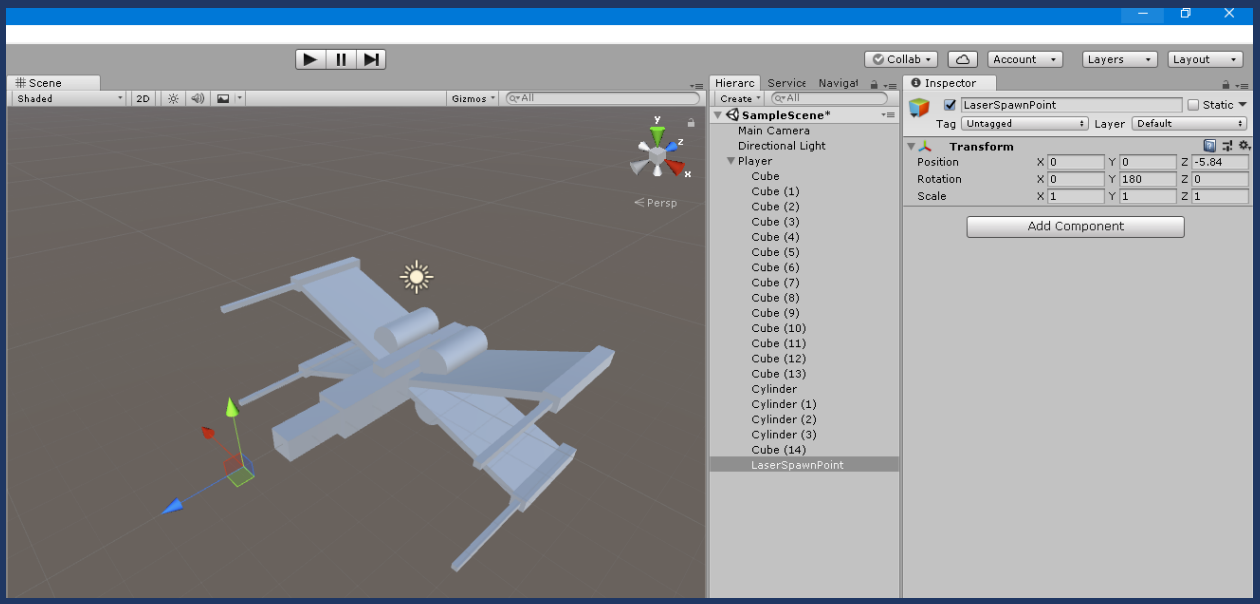


Next, let’s create an empty game object to represent the position where the lasers should spawn from when we shoot. Right click on your Player object and click “Create Empty”

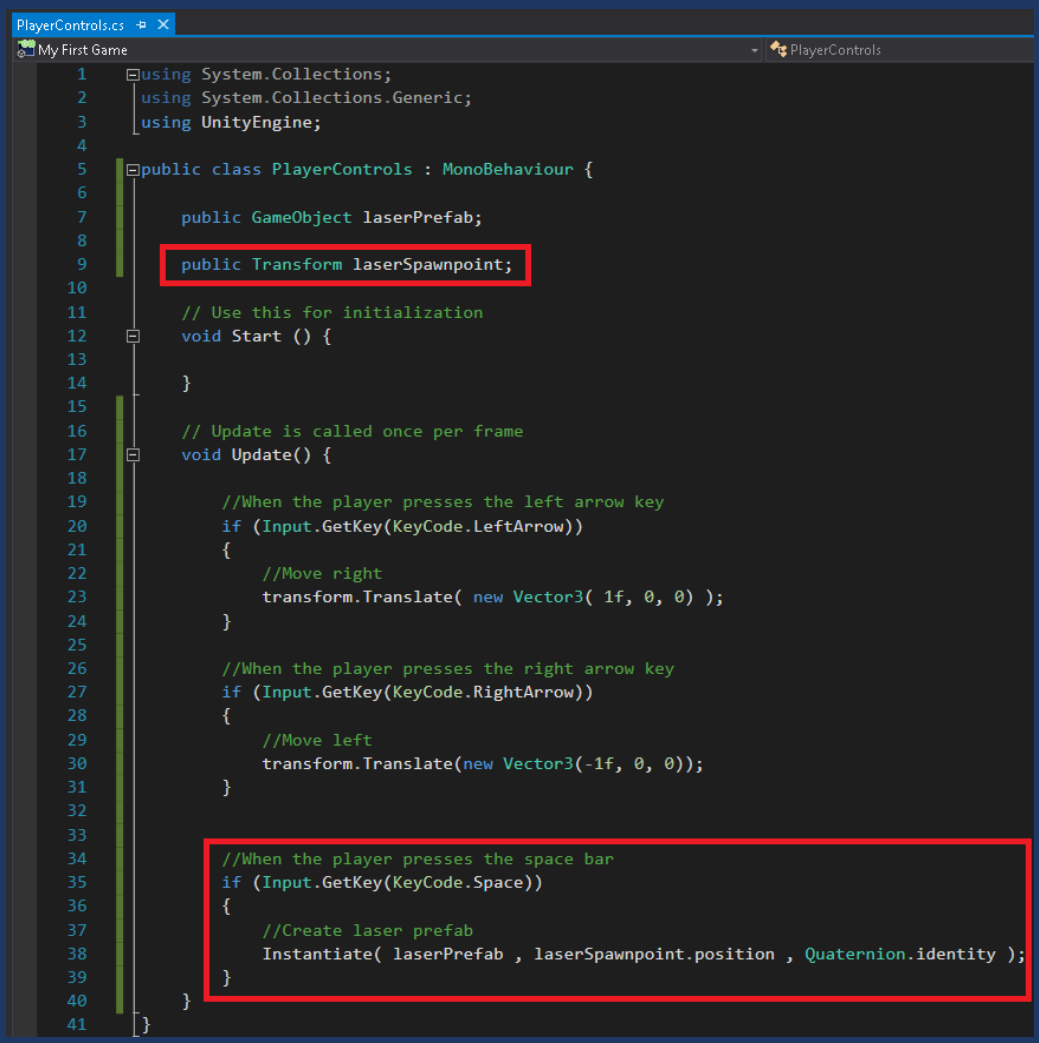


Position this object relative to your player where you want your lasers to spawn from. For now, we will just place it out in front of the x-wing. Later, if we want to make it more like the movies, we can have multiple of these objects, one for each wing cannon, but for now we will keep it simple and just have one out in front of the space craft.

We can name this object “LaserSpawnPoint”.



Now let's go back to our Player Controls script and add the logic for when you press the "shoot" button. First, let's add another public variable slot for our new LaserSpawnPoint game object. We will specify it as a transform, because that is all we really need. Unity allows us to specify any script that inherits from MonoBehaviour as a public variable that can be seen in the editor, which means we can drag any game object into that slot and it will automatically link the correct component from that game object to that slot so that we can access its variables and methods from that script. We want the transform component, because all we really need is the position information of the laser spawn point, which is stored in the transform component.

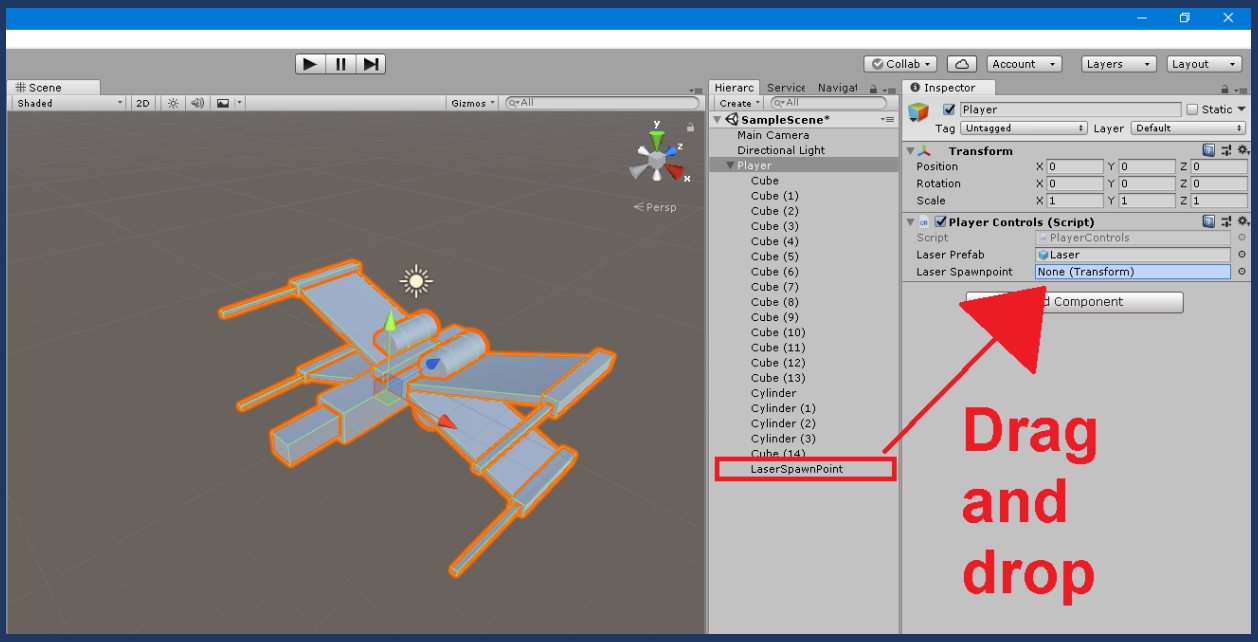


To create our laser, we will use the MonoBehaviour "Instantiate" method, which allows us to create a prefab in the scene from script and specify it's starting position and rotation. We will set the laser's starting position to the position of the laser spawn point, and set the rotation to Quaternion.identity, which is just the default zero rotation.

For those who don't know what a quaternion is, it is a data structure that stores the rotation of an object on 4 different axes, (x, y, z, w) allowing you to avoid "gimbal lock", a problem that occurs with 3 axis rotations, and therefore is extremely useful for game development. Unity

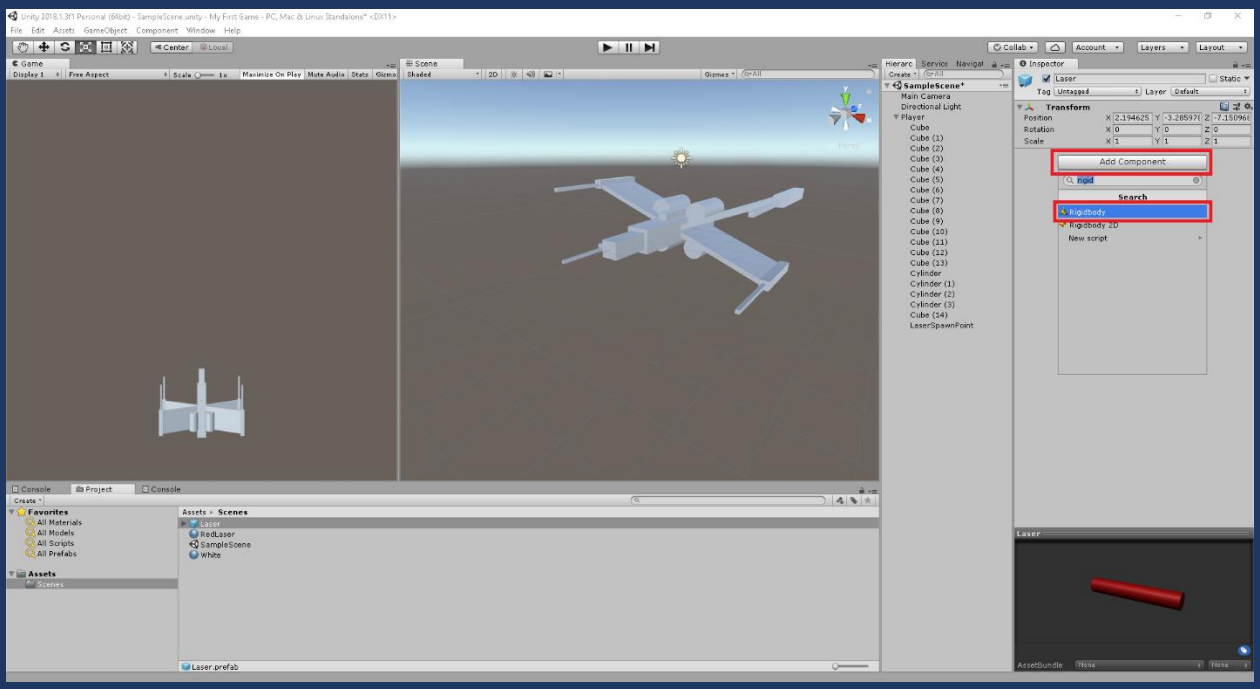
conveniently handles a lot of the quaternion math for you, so you only really need to know the basics in order to use it, however if you are interested in understanding it more deeply it can be an interesting Google search and there are some good YouTube videos explaining how they work.

Next, we will go back to Unity, and drag and drop our Laser Spawn Point into the Laser Spawn Point slot we just created, and then try running the game.



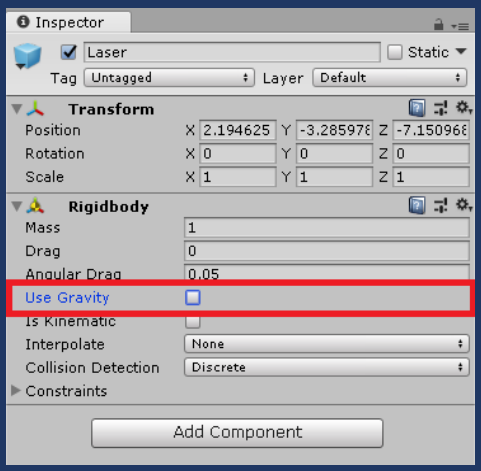
We should now see if we press the space bar, a laser prefab spawns at the laser spawn point position in front of our space ship, however does not move yet.

In order to make our laser move, lets exit the game and click on our laser prefab in the assets tab. In the Inspector tab, Click “Add Component” and start typing “RigidBody” and then select it from the dropdown that appears.



Unity’s RigidBody component is a powerful behavior that adds physics simulation to our object. This will allow us to assign the object a velocity, and therefore allow us to tell it to start moving when we spawn it.

Uncheck the “Use Gravity” checkbox, as we do not want our lasers to be affected by gravity.



Now let's go back to our PlayerControls script and assign a velocity to the spawned laser's RigidBody like so:

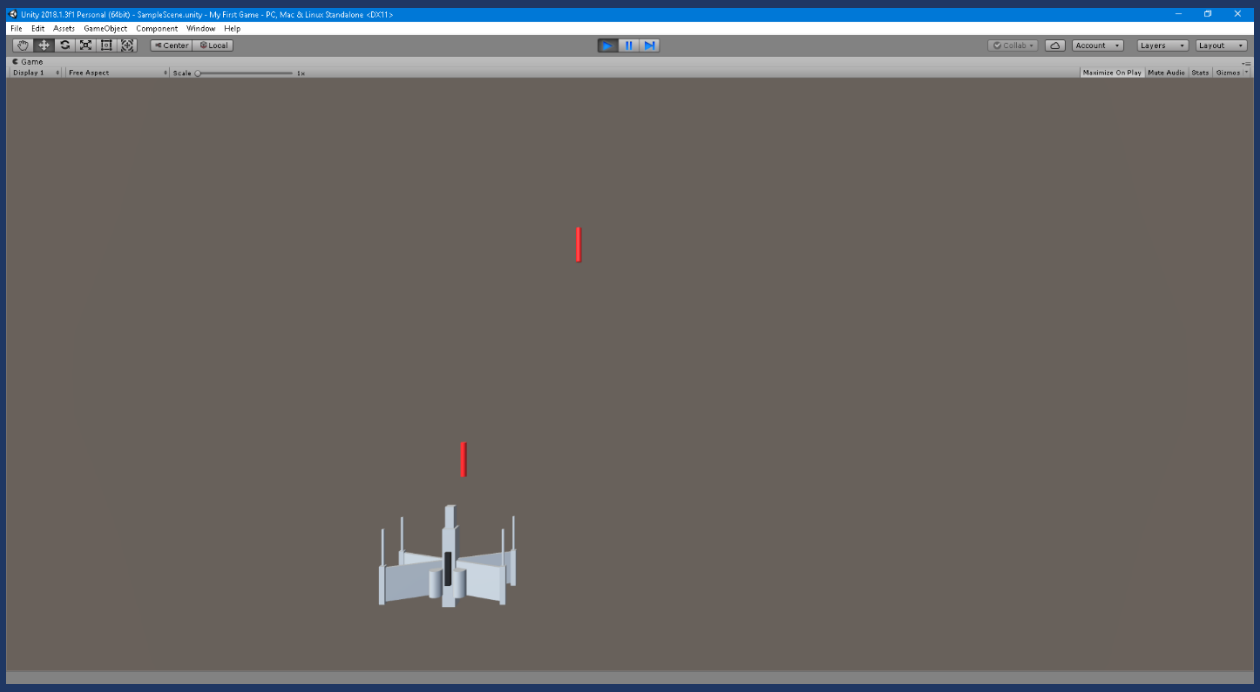
```
16      // Update is called once per frame
17      void Update() {
18
19          //When the player presses the left arrow key
20          if (Input.GetKey(KeyCode.LeftArrow))
21          {
22              //Move right
23              transform.Translate( new Vector3( 1f, 0, 0) );
24          }
25
26          //When the player presses the right arrow key
27          if (Input.GetKey(KeyCode.RightArrow))
28          {
29              //Move left
30              transform.Translate(new Vector3(-1f, 0, 0));
31          }
32
33          //When the player presses the space bar
34          if (Input.GetKeyDown(KeyCode.Space))
35          {
36              //Create laser prefab
37
38              GameObject newLaser = Instantiate( laserPrefab , laserSpawnpoint.position , Quaternion.identity );
39              Rigidbody laserRb = newLaser.GetComponent<Rigidbody>();
40              laserRb.velocity = laserSpawnpoint.forward * 100;
41          }
42      }
43  }
```

119 %

Few notes here:

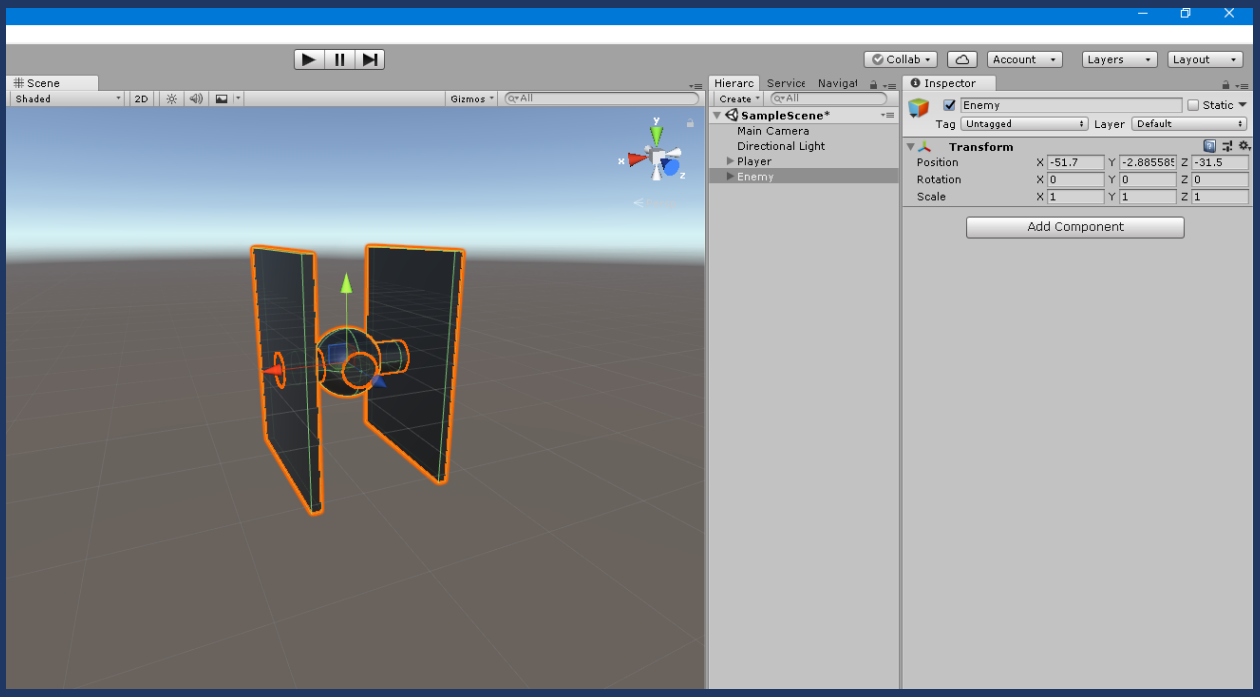
1. We replaced "GetKey" with "GetKeyDown", so that instead of spawning laser prefabs continuously when the space bar is held down, it only does it the instant the space bar is pressed.
2. We added "GameObject newLaser = " before the "Instantiate" call, so that we can store it for later use when we need to get the laser's RigidBody component. The Instantiate method will return the ID of the new game object after creating it and store it in the newLaser variable.
3. We get the laser's RigidBody component by calling newLaser.GetComponent<RigidBody>();
4. Once we have the laser's RigidBody component, we can set its velocity, which we set to a Vector equal to the laserSpawnpoint transform's "forward" direction times 100, to make it move faster (if we didn't multiply by 100 the laser would move really slowly). Best coding practices wouldn't use "magic numbers" like this, but rather instead have a variable there like "laserMoveSpeed", but we will just do it like this for now for simplicity. The RigidBody will automatically add this velocity to the position of the object every physics frame. Using the laserSpawnpoint's transform.forward also allows us to rotate the laserSpawnpoint directly from the Unity editor to face whatever direction we want the lasers to travel in when they are spawned.

Now if we run the game, we should observe that the lasers spawn with rapid forward movement:

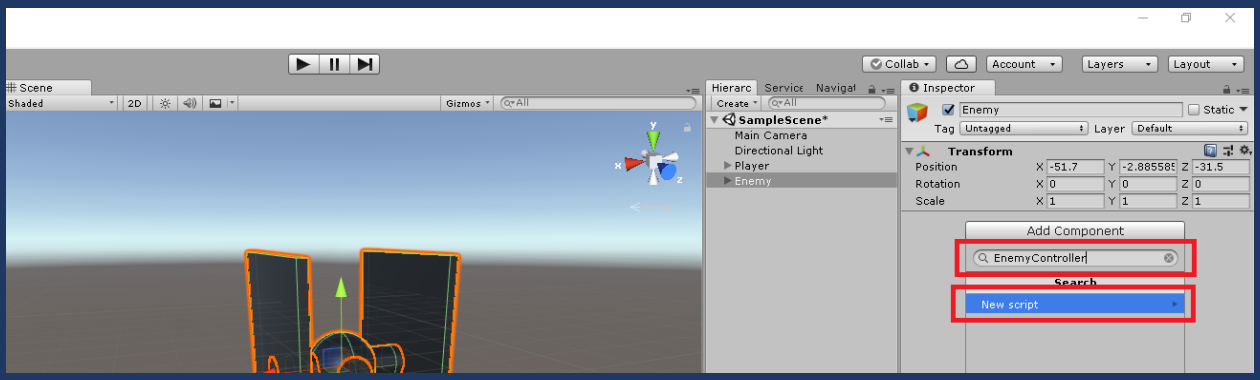


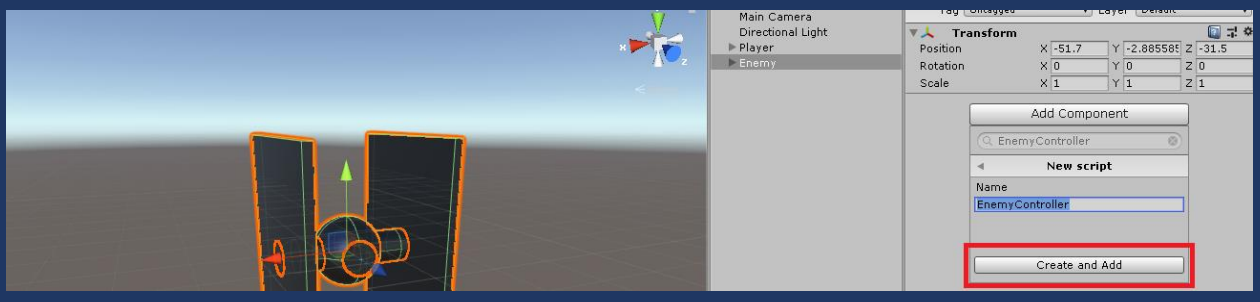
Now that we can both move and shoot, we need some enemies to dodge and shoot at!

Let's make an imperial tie fighter, the natural nemesis of our x-wing starfighter, (or you could make some other kind of enemy space ship / character of your choice)

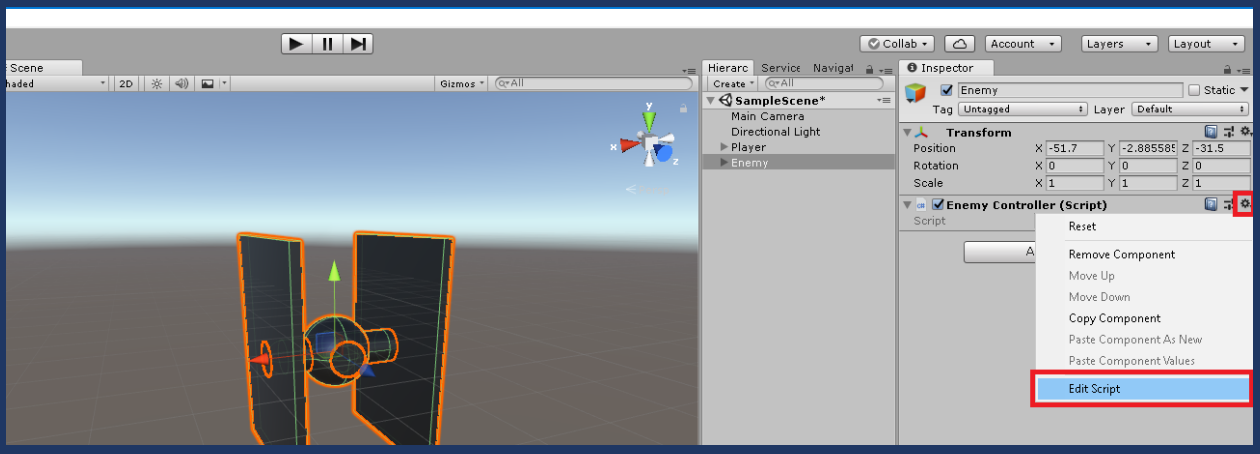


Once you finish creating the model for the enemy object, go ahead and name it “Enemy”, and then click “Add Component” and create a new script called “EnemyController”

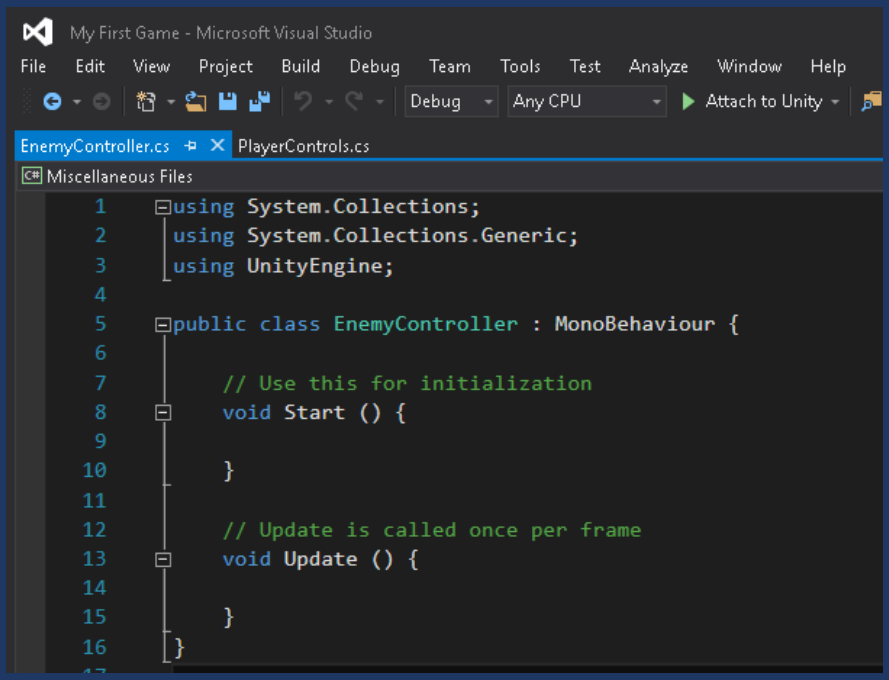




Then open the new script to edit it by clicking the gear icon and then “Edit Script”.



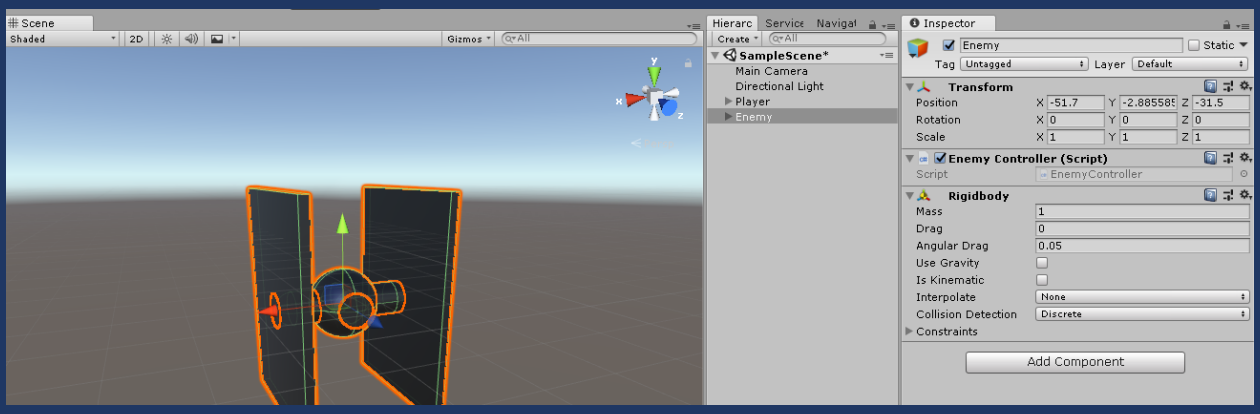
As you can see, the script looks exactly the same as our PlayerControls script was when we created it, with an Update and Start method.



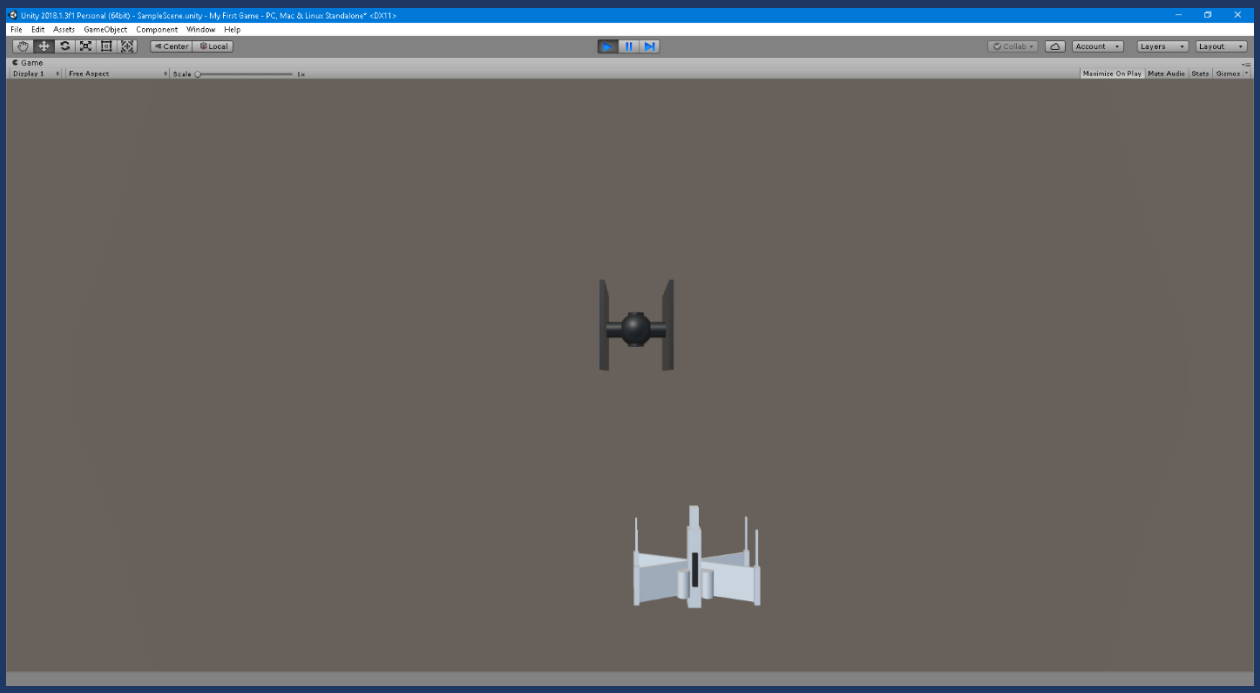
By default, we want our tie fighters / enemies to be moving towards us. We can set the velocity of the enemy similar to how we did it with our lasers earlier, by adding a RigidBody component to the enemy object, and then accessing that component with GetComponent<RigidBody>() and setting it's velocity, like so:

```
Assets\EnemyController.cs   PlayerControls.cs
My First Game                EnemyController                Update()
 1  using System.Collections;
 2  using System.Collections.Generic;
 3  using UnityEngine;
 4
 5  public class EnemyController : MonoBehaviour {
 6
 7      Rigidbody rb;
 8
 9      // Use this for initialization
10      void Start () {
11          rb = GetComponent<Rigidbody>();
12      }
13
14      // Update is called once per frame
15      void Update () {
16          rb.velocity = new Vector3(0, 0, 10);
17
18          if (transform.position.z > 20)
19          {
20              transform.position = new Vector3( transform.position.x, transform.position.y, -50 );
21          }
22      }
23  }
```

Go back to the editor and add a Rigidbody component to your enemy object, the same way we did it for the laser prefab. Make sure to turn "Use Gravity" off.



Try running the game. You should now see that the tie fighter flies towards the player object.



You will notice that when you shoot at the tie fighter, it bounces off the lasers and starts spinning, like the lasers are solid objects that are hitting it. What we want to happen, is for the tie fighter to blow up in a fantastic explosion and then disappear. We also want the laser to disappear when it hits the tie fighter.

We also want enemies to continually spawn from above and fly towards the player, so really we don't need to delete the enemy object from the scene when we destroy it, we can just have it reset it's position to somewhere above the screen so that it is continually looping. We should have it reset its position whenever it goes out the bottom side of the screen, and whenever we hit it with a laser.

We also want to display a "game over" screen when the tie fighter hits the player.

To do this from script, go back to the EnemyController script and add these changes:

Assets\EnemyController.cs | PlayerControls.cs

My First Game | EnemyController | Update()

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

public class EnemyController : MonoBehaviour {

    //Initialize Rigidbody variable
    Rigidbody rb;

    //Initialize start position variable
    Vector3 startPos;

    // Use this for initialization
    void Start () {
        //Get the Rigidbody component attached to this game object
        rb = GetComponent<Rigidbody>();

        //Set the velocity of this game object's Rigidbody component
        rb.velocity = new Vector3(0, 0, 10);

        //Store the start position of the object
        startPos = transform.position;
    }

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

        //Check if the enemy is outside the bottom side of the play area
        if (transform.position.z > 30)
        {
            //If it is, reset it's position
            Respawn();
        }
    }

    //This event is triggered whenever the Enemy object collides with something
    void OnCollisionEnter(Collision col)
    {
        //If it hit the player
        if (col.transform.name.Contains("Player"))
        {
            //Get the playerTransform component
            GameObject playerObj = col.gameObject;

            //Get the playercontrols component from the transform
            PlayerControls playerControls = playerObj.GetComponent<PlayerControls>();

            //Display the game over screen
            playerControls.GameOver();
        }

        Respawn();

    }

    void Respawn()
    {
        //Reset it's position to somewhere above the play area, randomizing it's x position
        transform.position = new Vector3(startPos.x + Random.Range(-20, 20), startPos.y, startPos.z);
        rb.velocity = new Vector3(0, 0, 10);
    }
}
```

Now go to the PlayerControls script and add these changes:

Assets\EnemyController.cs | PlayerControls.cs

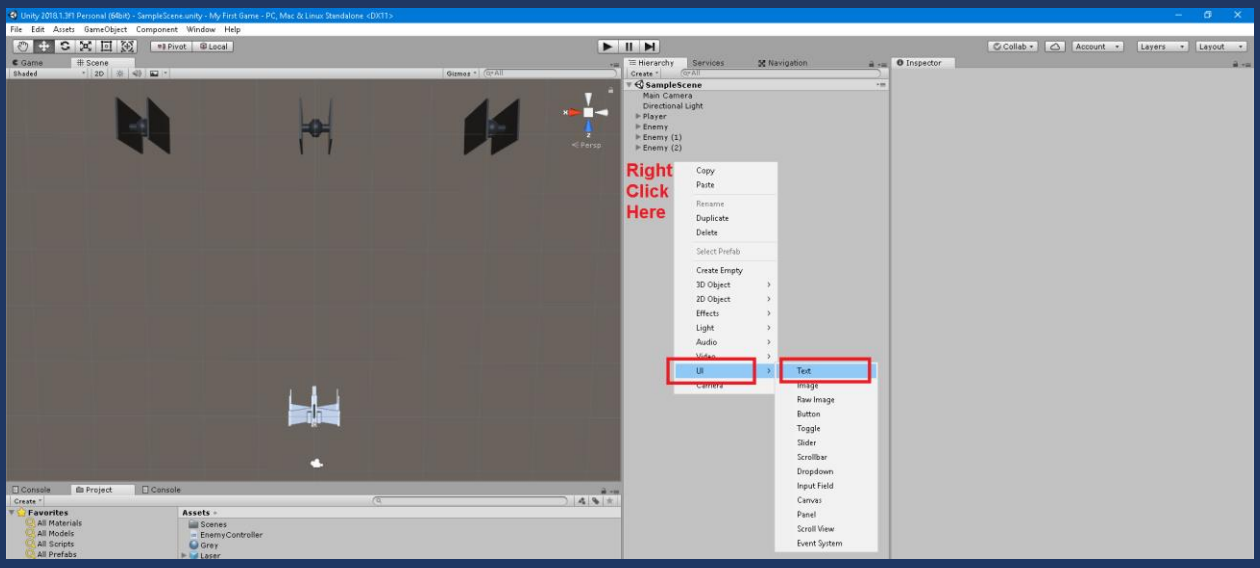
My First Game | PlayerControls | Update()

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

public class PlayerControls : MonoBehaviour {

    public GameObject laserPrefab;

    public Transform laserSpawnpoint;

    public GameObject gameOverScreen;

    // Use this for initialization
    void Start () {

    }

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

        //When the player presses the left arrow key
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            //Move right
            transform.Translate( new Vector3( 1f, 0, 0) );
        }

        //When the player presses the right arrow key
        if (Input.GetKey(KeyCode.RightArrow))
        {
            //Move left
            transform.Translate(new Vector3(-1f, 0, 0));
        }

        //When the player presses the space bar
        if (Input.GetKeyDown(KeyCode.Space))
        {
            //Create laser prefab

            GameObject newLaser = Instantiate( laserPrefab , laserSpawnpoint.position , Quaternion.identity );
            Rigidbody laserRb = newLaser.GetComponent<Rigidbody>();
            laserRb.velocity = laserSpawnpoint.forward * 100;
        }
    }

    public void GameOver()
    {
        gameOverScreen.SetActive(true);
    }
}
```

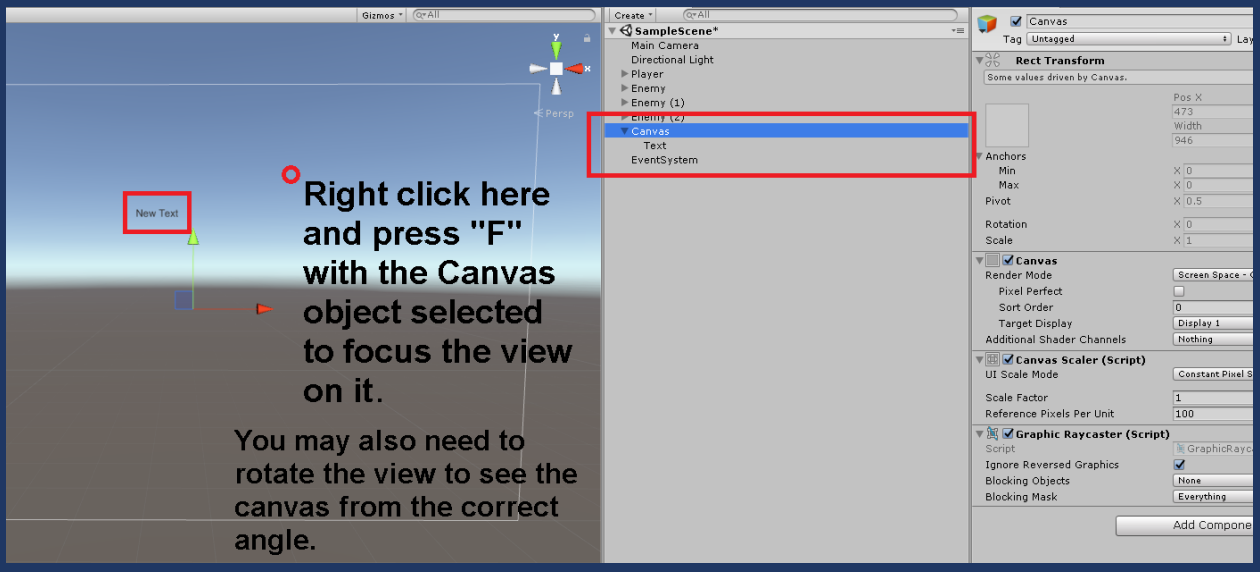
We don’t yet have a game over screen, so let’s create one.

A simple way to create 2D user interface (UI) in Unity is using Unity's built in UI Canvas system.

To create some game over text on the screen, we can right click in the scene hierarchy tab and click "UI > Text"

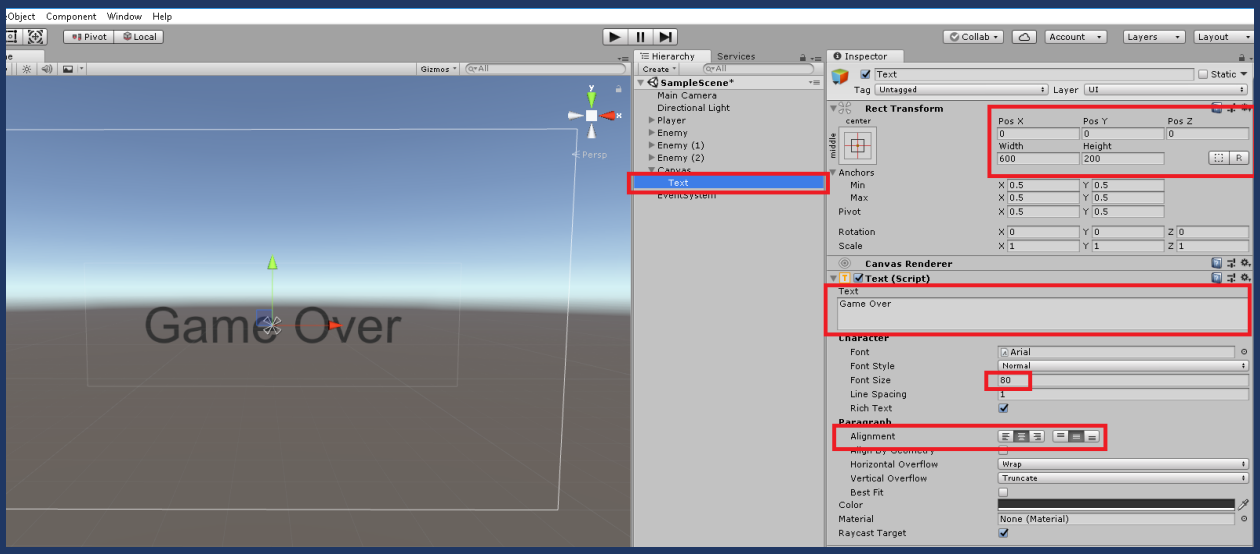


Right click inside the Scene tab and then press "F" on your keyboard to zoom out to where the text was created. It should look something like this:



Notice when we created the Text, Unity automatically created it under a parent object called "Canvas", and also created another game object called an "EventSystem". The EventSystem object is used later when we want to add interactive UI such as buttons and sliders, and works in the background and there isn't much you will need to do with it. The Canvas object is the object that all of our 2D UI will be grouped under, and does special rendering specifically for 2D UI to make it look good on many different displays and screen resolutions.

With the Canvas now in view, you can see the new text label object which we created, which says “New Text”. Click on the Text object in the Hierarchy tab and then change the parameters in the inspector to match those below:

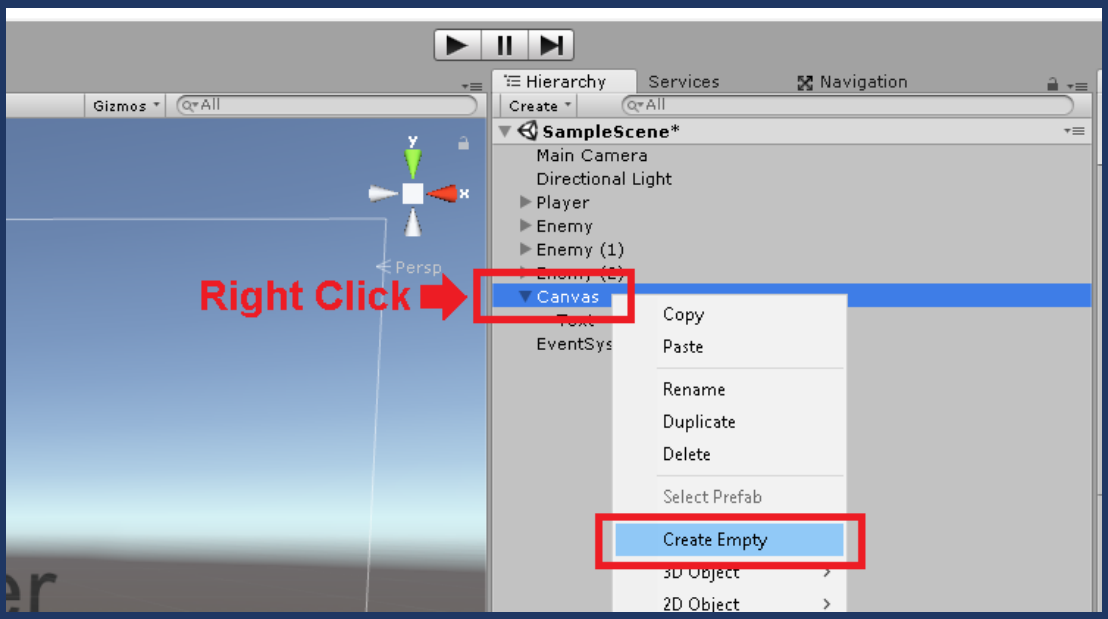


To list the changes we made:

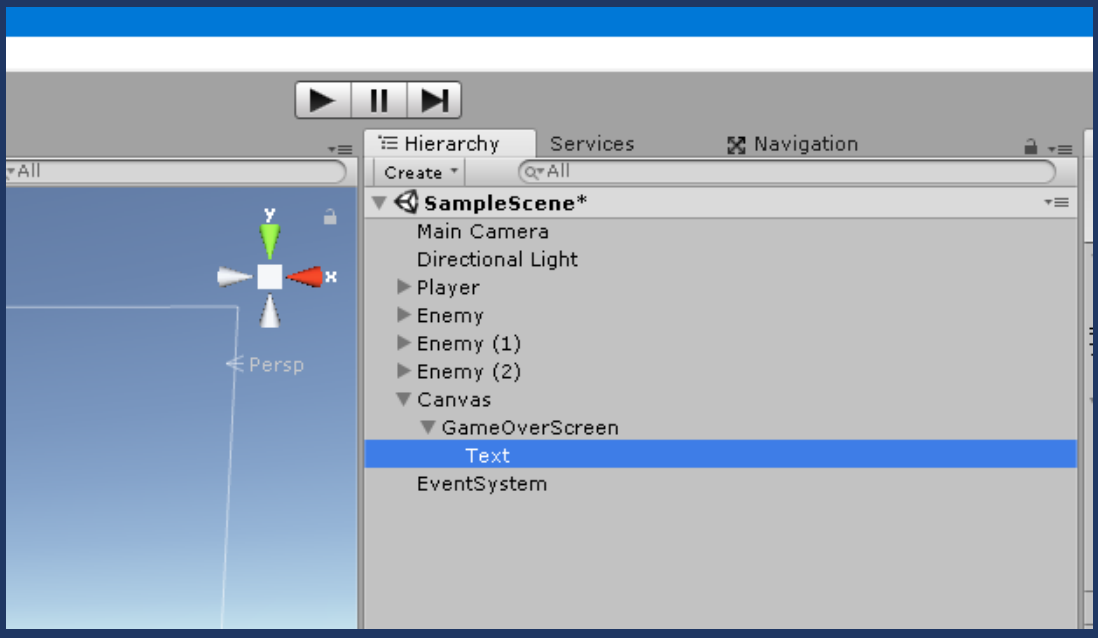
1. Change the Pos X and Pos Y to both be 0, and change the width to be 600 and the height to be 100.
2. Set the text to say "Game Over”
3. Set the font size to 80
4. Set the text alignment to be centered both horizontally and vertically.

Now, the “Game Over” text isn’t the only thing we want displayed on the Game Over screen, we will also probably want a “Try Again” button so the player can continue playing / try again after losing the game. Later, we might also want to add things like a high score label, a “return to main menu” button, images, animations, ect… and so it’s useful for us to create a parent object that all of these game over screen objects will be stored under, so that the only object we need to enable is that one parent object when the game ends instead of having to enable each of the objects individually.

To create a parent object for the game over screen, right click on Canvas and click “Create Empty”. We can rename this new object to “Game Over Screen”. Then drag and drop the Text object we created earlier onto this object to set it as a child object.



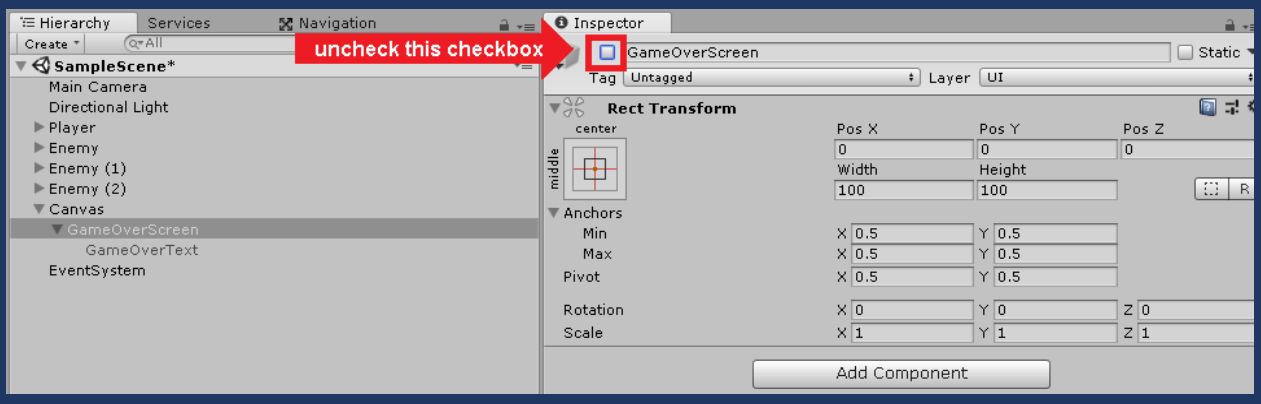
When you're done it should look like this:



As you can see, the GameOverScreen is a child object of the Canvas object, and the "Game Over" Text label is a child object of the GameOverScreen. (We can rename this Text object to GameOverText for clarity)

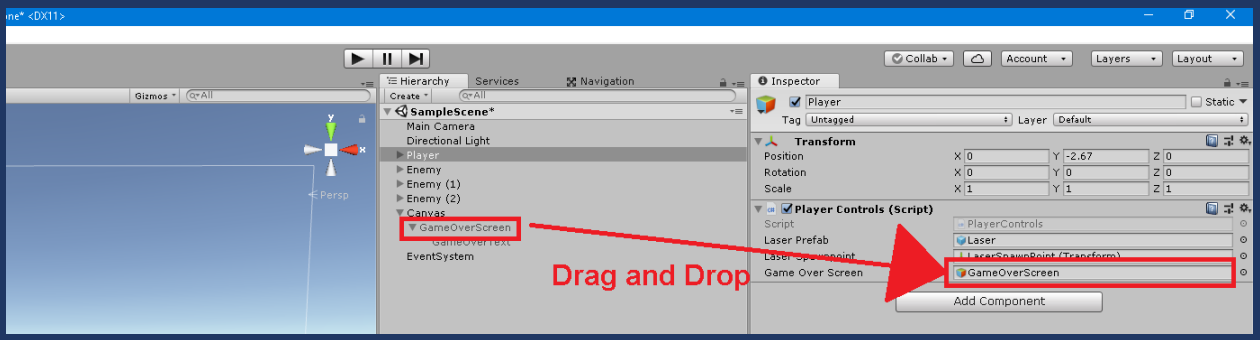
Now, if we run the game, we see that we instantly lose:

To make the game over screen not appear instantly when we start the game, we can exit the game and set the GameOverScreen object's "Enabled" property to false:



You can now see the GameOverScreen and it's child object are greyed out in the Hierarchy tab view, meaning the objects will be invisible in game and their scripts won't be executed.

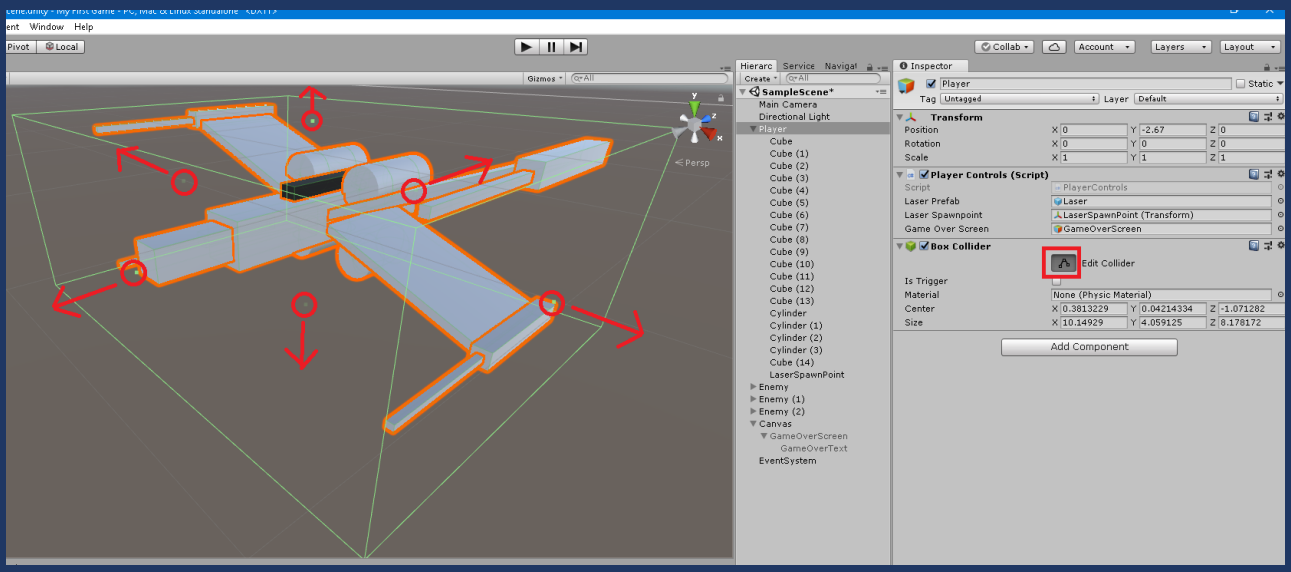
We have already written the code to enable the GameOverScreen object when the game ends, we now just need to click on our player object, and then drag and drop the GameOverScreen object into the GameOverScreen slot in the PlayerControls component:



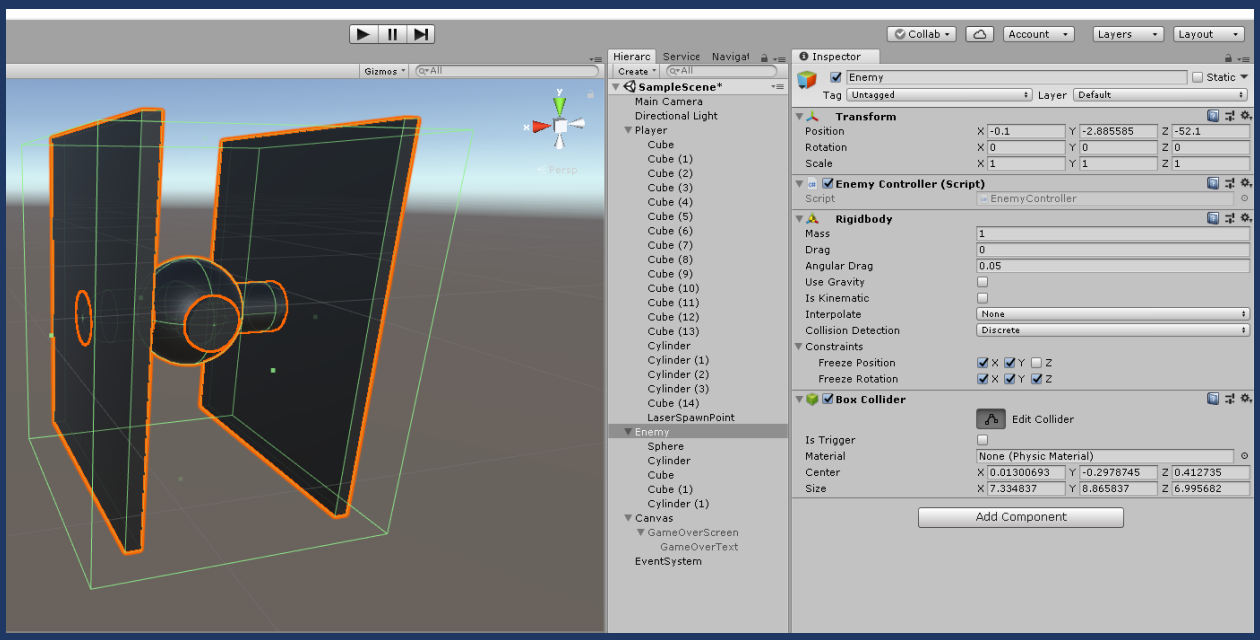
Another thing we need to do is add collider components to both the player and the enemy objects. Their child objects already have them, but this won't work with our collision code, so let's disable those and add some to the parent objects. A collider component defines a volume in which if another collider enters that volume, it will be considered a collision, and execute the OnCollision event which we coded earlier into our EnemyController script.

First click on the Player, and go Add Component > Box Collider.

Then click the button labeled "Edit Collider" and use the dots around the green box that appears around the player to click and drag the collider box to encompass the full player object.



Repeat the same steps for your enemy object:



Now if we run the game, the Game Over screen should not appear until a tie fighter crashes into our x-wing.

One final thing we need to do to make it more complete, is make it so the player object gets disabled when the game is over, because right now when you get hit by a tie fighter and lose the game, you can still move around afterwards.

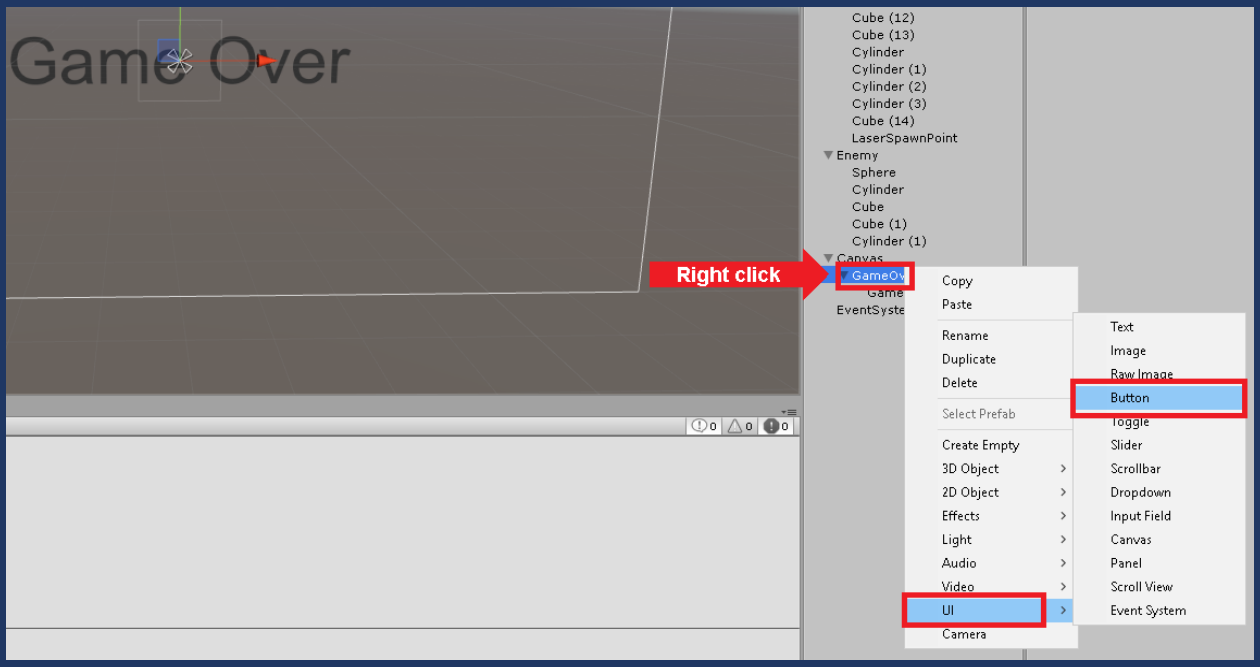
To do this, let's go back to our PlayerControls script, and add a line that disables the game object in the GameOver() method:

```
46      public void GameOver()
47      {
48          //Activate the game over screen
49          gameOverScreen.SetActive(true);
50
51          //Disable the player object
52          gameObject.SetActive(false);
53      }
54  }
```

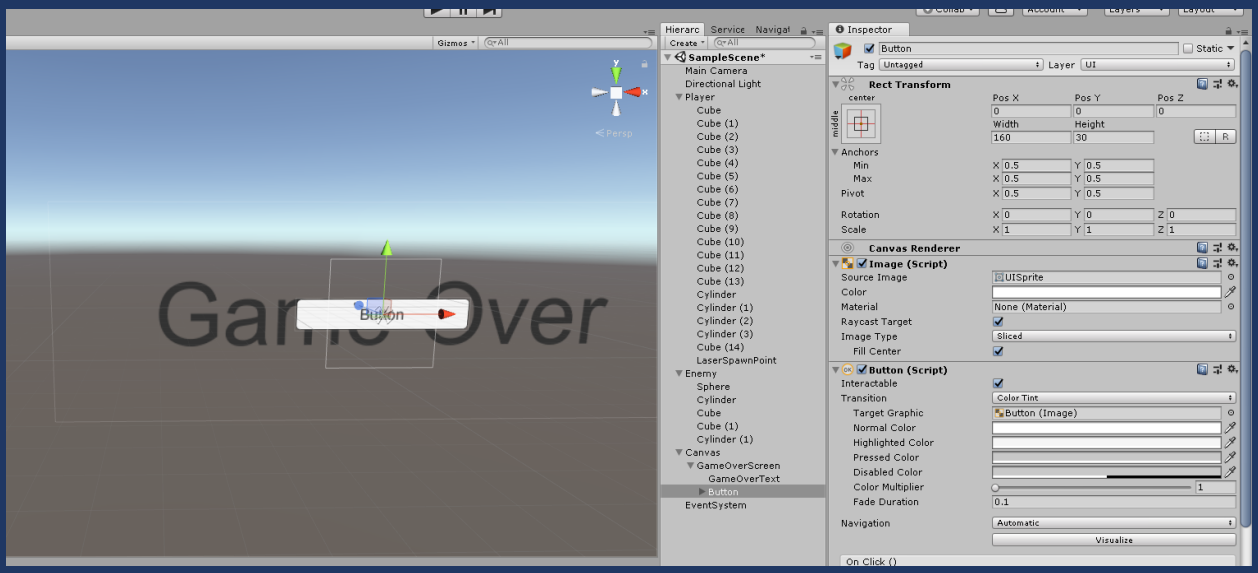
Now if we run the game and get hit by an enemy, it should disable the player object and enable the game over screen:

Now, what if we wanted to add some interactive UI elements, like a button, to our game? Let's try adding a "Restart Game" button to the game over screen.

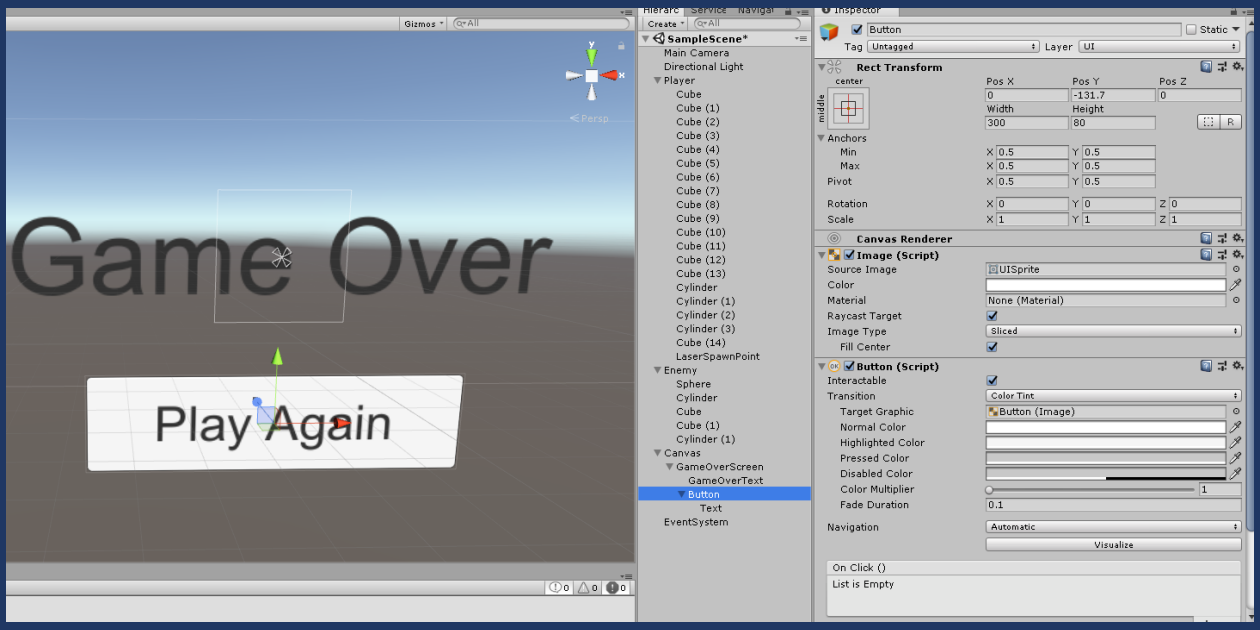
To do this, re-enable the game over screen object in the scene editor so we can see it and edit it, and then right click on it and click "UI > Button" to add a button.



You should see a new button appear:

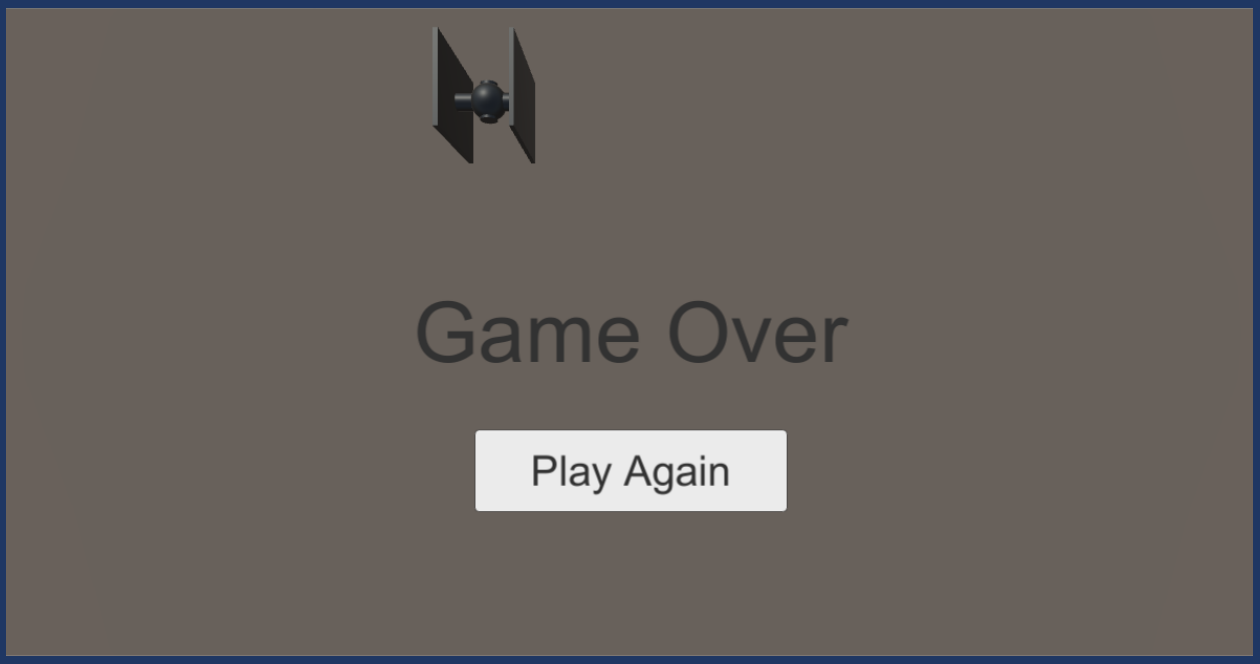


Move the button below the Game Over text, change it's text to say "Play Again", and make it bigger by changing the width, height, and font parameters. Note to change the button's text, you will need to change it in the Button's child Text object.

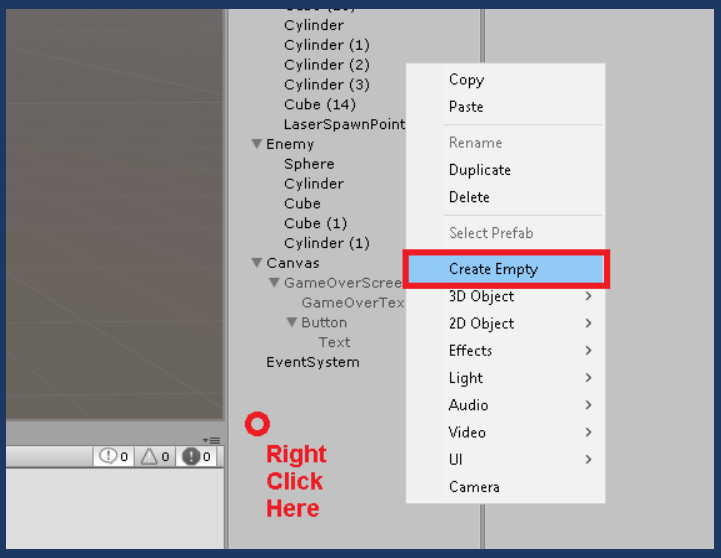


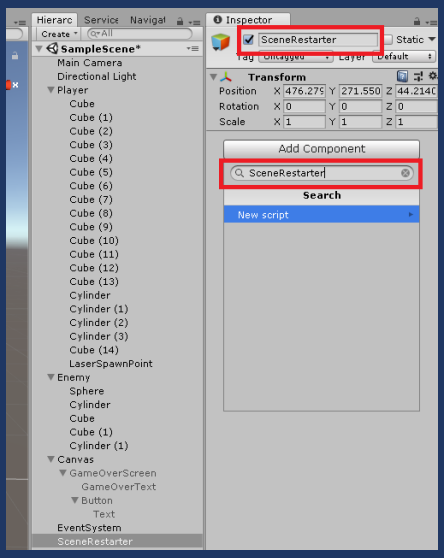
When you're done, disable the GameOverScreen object again and then run the game.

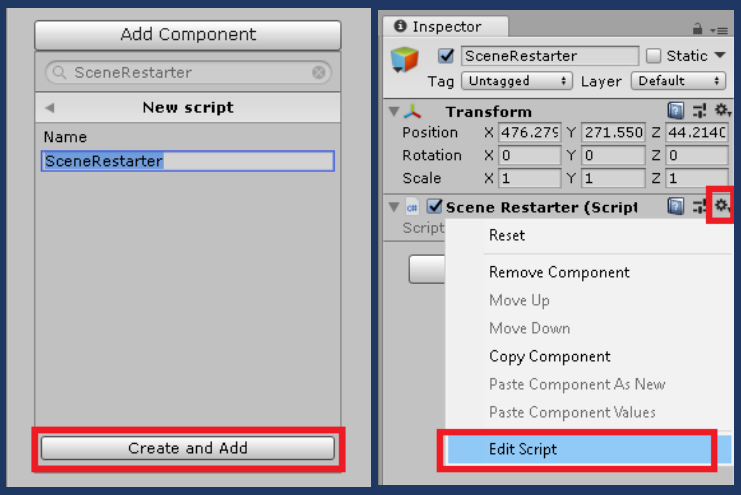
When you lose, you should now see the new button appear in the game over screen, however when you click it, you will notice nothing happens, because we haven't told it to do anything yet when we click it.



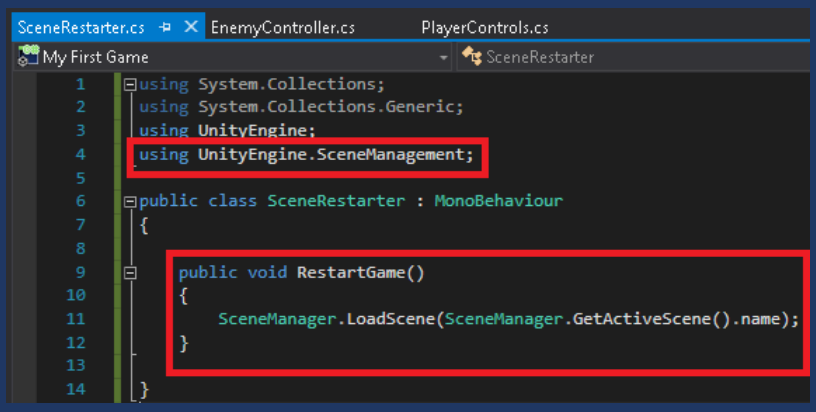
To make the "Play Again" button actually restart the game, lets create a new object called "SceneRestarter" and then create a new component script for it also called "SceneRestarter", and then edit this script.





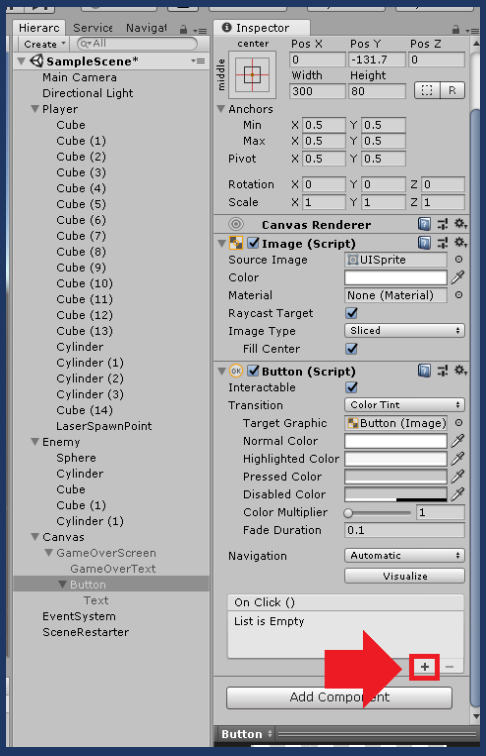


You can delete the default “Start” and “Update” methods of this script, as we won’t be needing them. Instead, add a public “RestartGame” method, and also add an import statement at the top to use Unity’s Scene Management library:

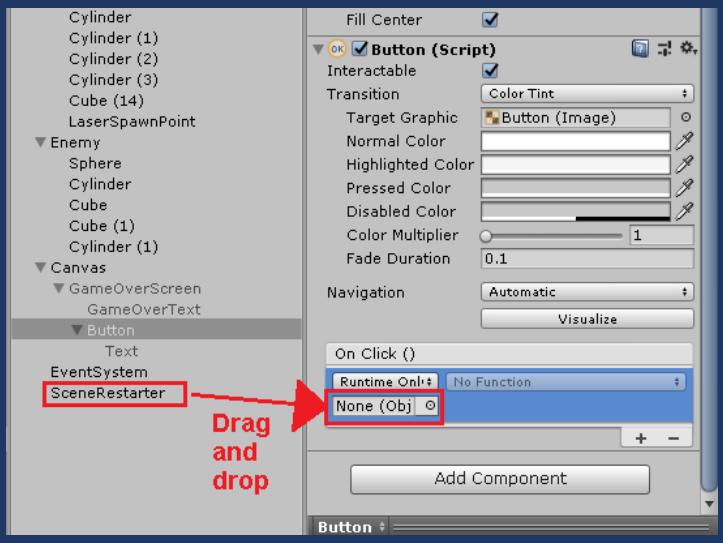


Now go back to the Unity editor, and click on your Restart Game button in the game over screen.

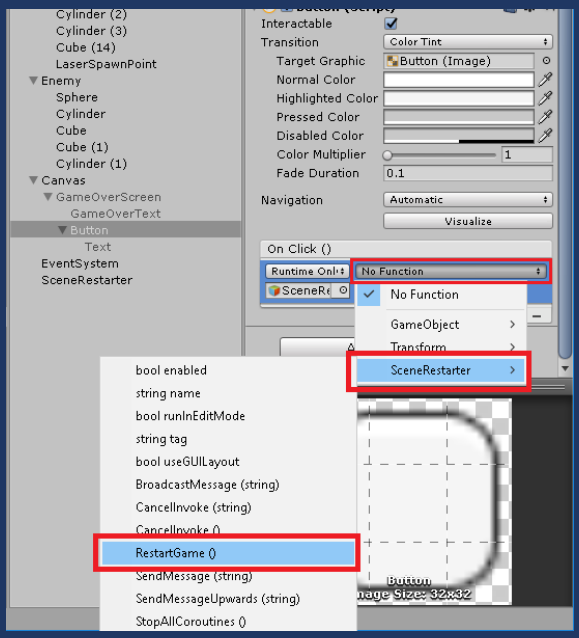
In the button component, click the “+” button under OnClick() to add a new function to execute when this button is clicked.



Then drag and drop your SceneRestarter object into the object slot of the new OnClick() function.



Then click where it says “No Function” and select SceneRestarter > RestartGame()

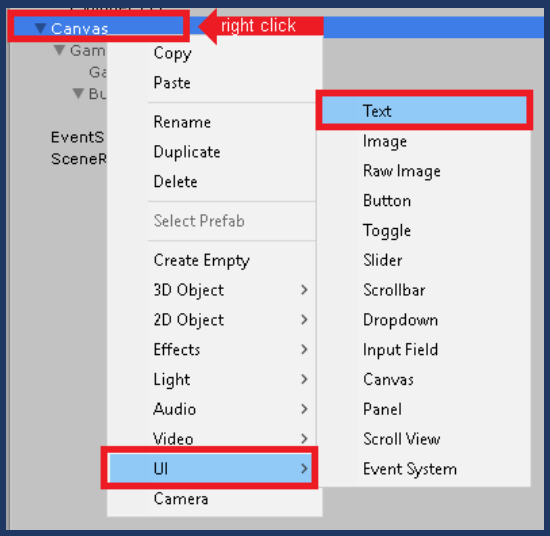


Now if you run the game and try losing, and then press the “Play Again” button, it should restart the game.

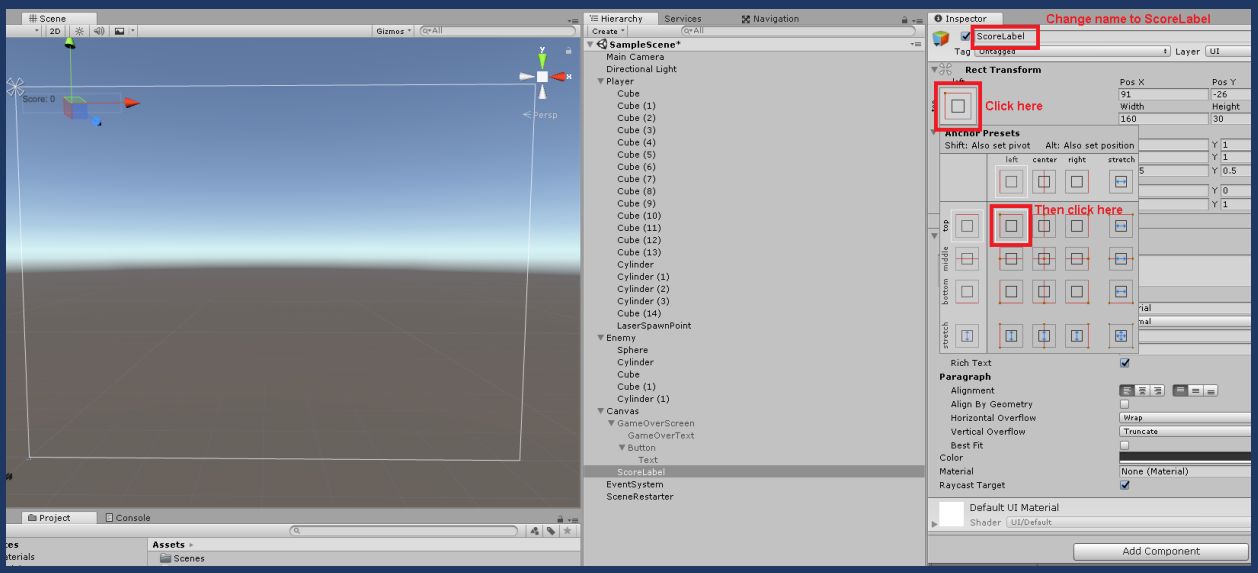
So now we have a game where we play as an x-wing fighter, we can shoot at incoming enemy tie fighters, we can lose by hitting an enemy, and we can see a game over screen with a restart button. One thing we might want to add is a way to keep score, so the player knows how well they are doing. So let’s add a score label.

To do this, similar to how we added the game over text, right click on Canvas and click

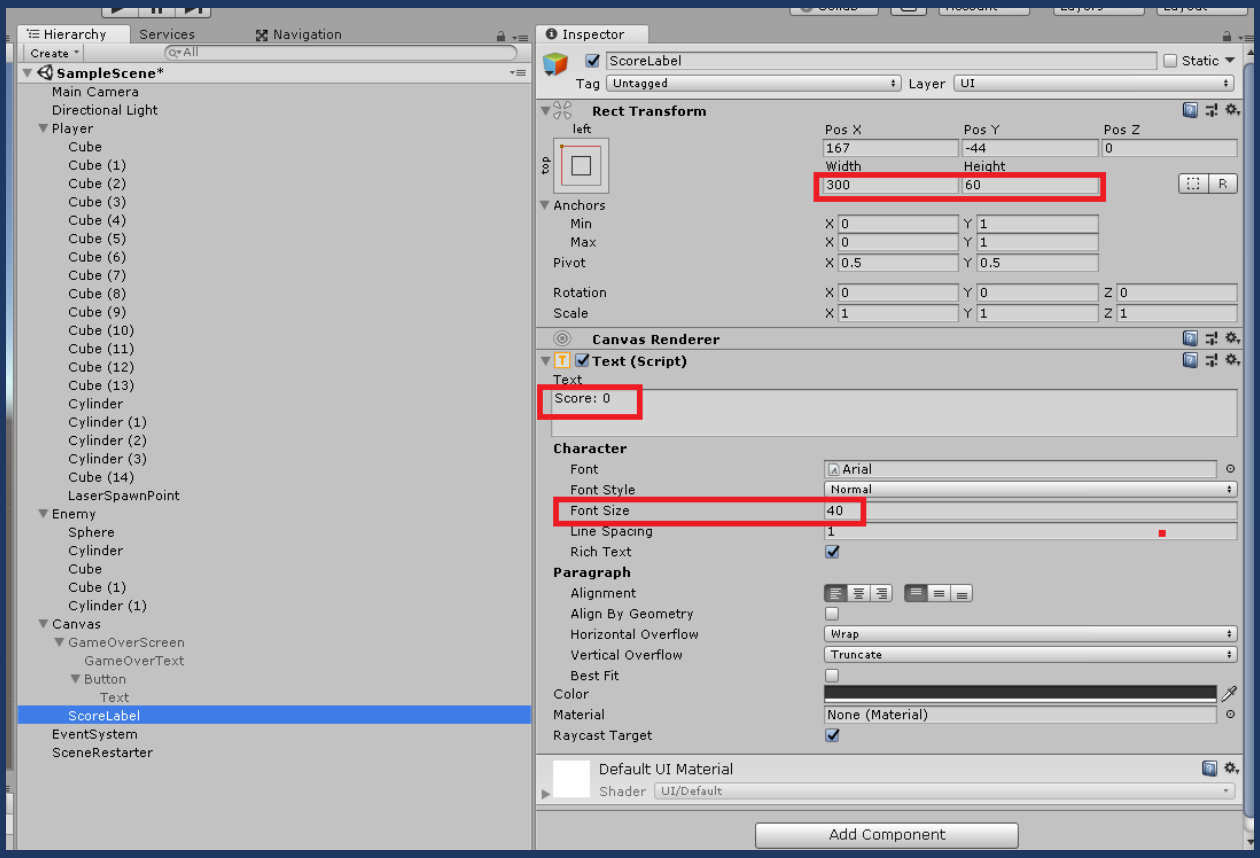
“UI > Text”



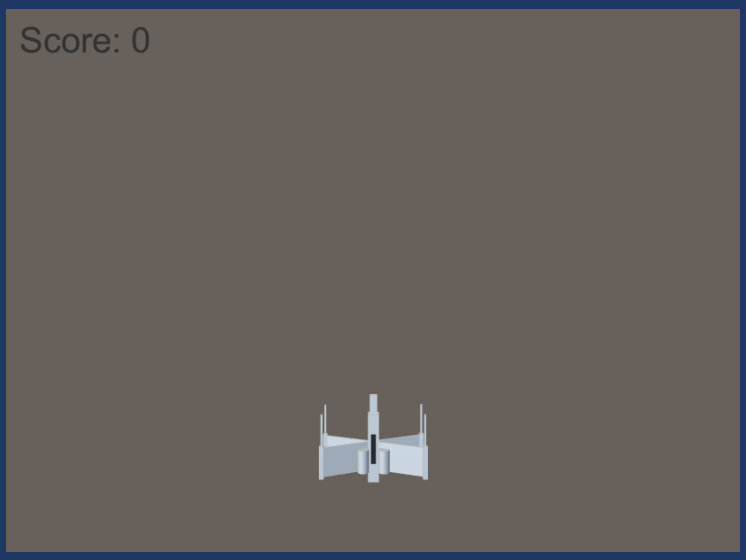
Name this new Text object “ScoreLabel”, then click on the “Anchor Presets” button and click the one to align the text to the top left corner of the screen. Then move the text to be in the top left corner. This will make sure we always preserve the position of the text in that corner of the screen for when we port the game to different resolution displays.



After that, change the width of the score label to 300 and the height to 60, change the text to say “Score: 0”, and change the font size to 40. You might have to move the text again in the scene after resizing to make sure it is still within the canvas.



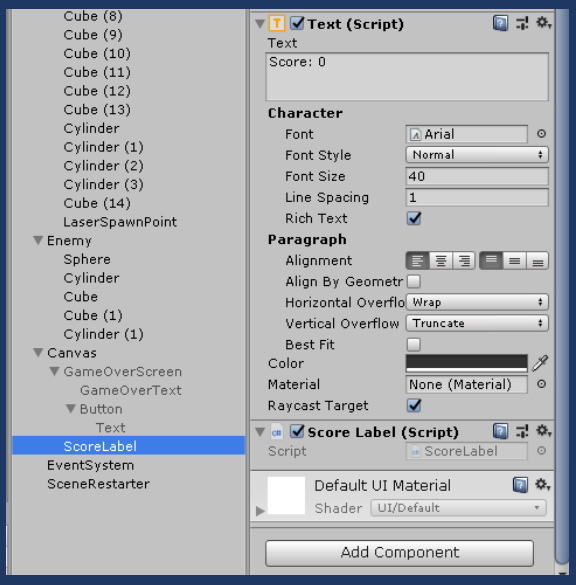
When you're done it should look like this:

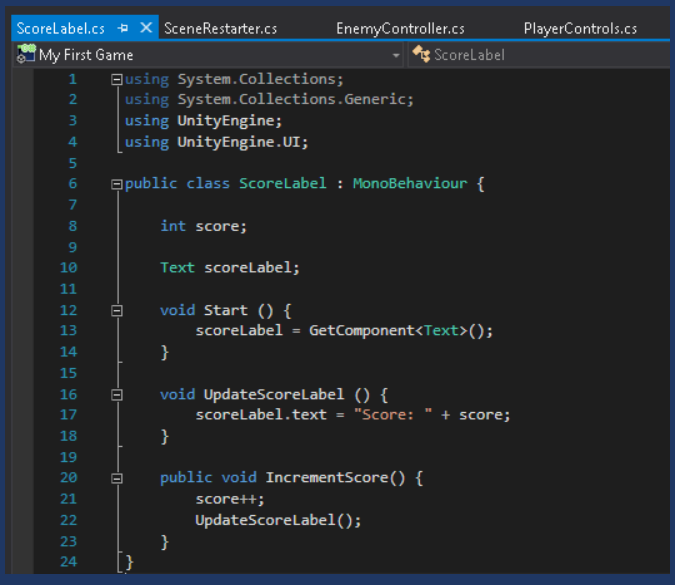


Now if we play the game, we will see that the score always stays 0, because we never told it to increment whenever we hit an enemy.

In order to do this, we will need to create a new script on the score label object to update the score label whenever we hit an enemy.

Let's create a new component script attached to the ScoreLabel object and call it ScoreLabel. (Add Component > New script > name it "ScoreLabel" and then click the gear icon and edit the script)





In this script, we will have a counter variable to keep score, a public method to increment the score, and a method to update the score label with the new score. We will get the Text component attached to the game object in the start method and use that to set the text whenever the score is updated.

Now we just need to have our enemy objects call this script's IncrementScore() method whenever they are hit by a laser.

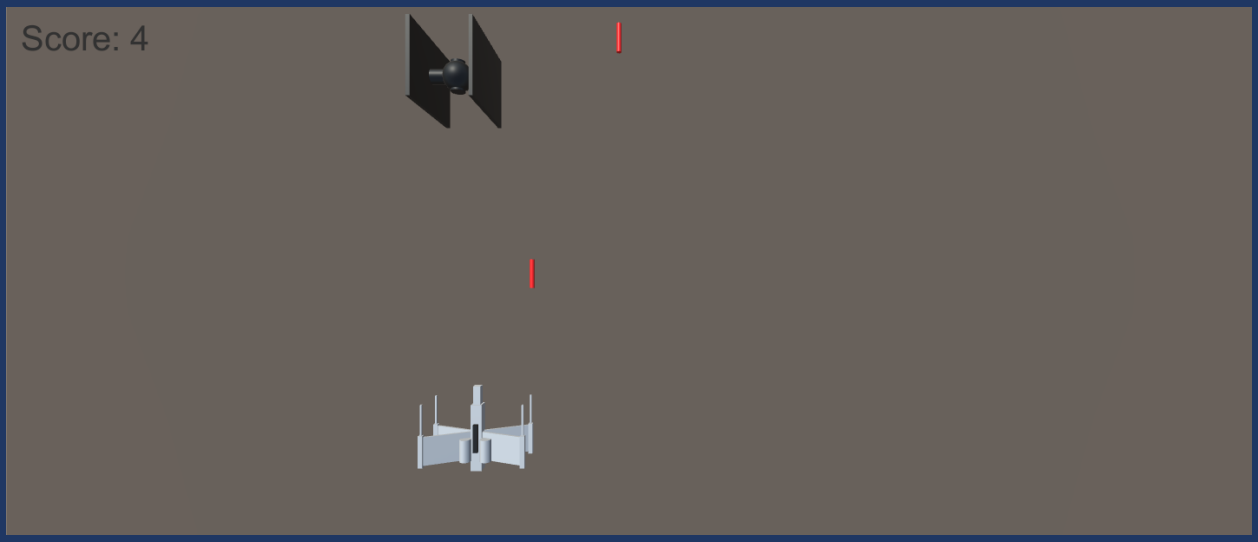
To do this, let's go back to the EnemyController script and add the following changes:

ScoreLabel.cs | SceneRestarter.cs | EnemyController.cs | PlayerControls.cs

My First Game | EnemyController | Update()

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class EnemyController : MonoBehaviour {
6
7      //Initialize Rigidbody variable
8      Rigidbody rb;
9
10     //Initialize start position variable
11     Vector3 startPos;
12
13     //Initialize ScoreLabel variable
14     ScoreLabel scoreLabel;
15
16     // Use this for initialization
17     void Start () {
18         //Get the Rigidbody component attached to this game object
19         rb = GetComponent<Rigidbody>();
20
21         //Set the velocity of this game object's Rigidbody component
22         rb.velocity = new Vector3(0, 0, 10);
23
24         //Store the start position of the object
25         startPos = transform.position;
26
27         //Find the ScoreLabel component in the scene
28         scoreLabel = GameObject.FindObjectOfType<ScoreLabel>();
29     }
30
31     // Update is called once per frame
32     void Update () {
33
34         //Check if the enemy is outside the bottom side of the play area
35         if (transform.position.z > 30)
36         {
37             //If it is, reset it's position
38             Respawn();
39         }
40     }
41
42     //This event is triggered whenever the Enemy object collides with something
43     void OnCollisionEnter(Collision col)
44     {
45         //If it hit the player
46         if (col.transform.name.Contains("Player"))
47         {
48             //Get the playerTransform component
49             GameObject playerObj = col.gameObject;
50
51             //Get the playercontrols component from the transform
52             PlayerControls playerControls = playerObj.GetComponent<PlayerControls>();
53
54             //Display the game over screen
55             playerControls.GameOver();
56         }
57
58         //If it was hit by a laser
59         if (col.transform.name.Contains("Laser"))
60         {
61             //Increment score
62             scoreLabel.IncrementScore();
63
64             //Destroy the laser object
65             Destroy(col.gameObject);
66         }
67
68         Respawn();
69     }
70
71     void Respawn()
72     {
73         //Reset it's position to somewhere above the play area, randomizing it's x position
74         transform.position = new Vector3(startPos.x + Random.Range(-20, 20), startPos.y, startPos.z);
75         rb.velocity = new Vector3(0, 0, 10);
76     }
77 }
```

Now if we run the game, we should see that whenever we hit an enemy with a laser, the score should increase by 1.



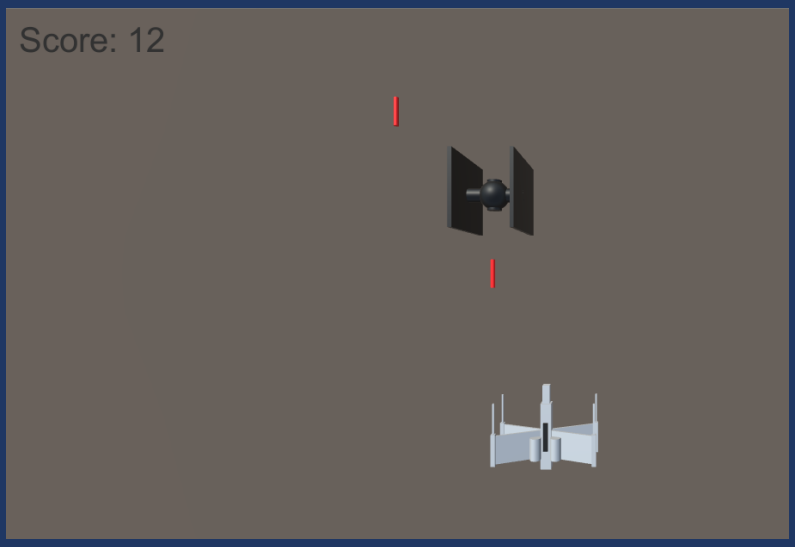
Now it seems we almost have a completely playable game, but it's still a very easy game. We can make the game harder by making the enemies gradually increase in speed over the course of the game. To do this, let's go back to our EnemyController script and add the following changes:

ScoreLabel.cs SceneRestarter.cs EnemyController.cs PlayerControls.cs

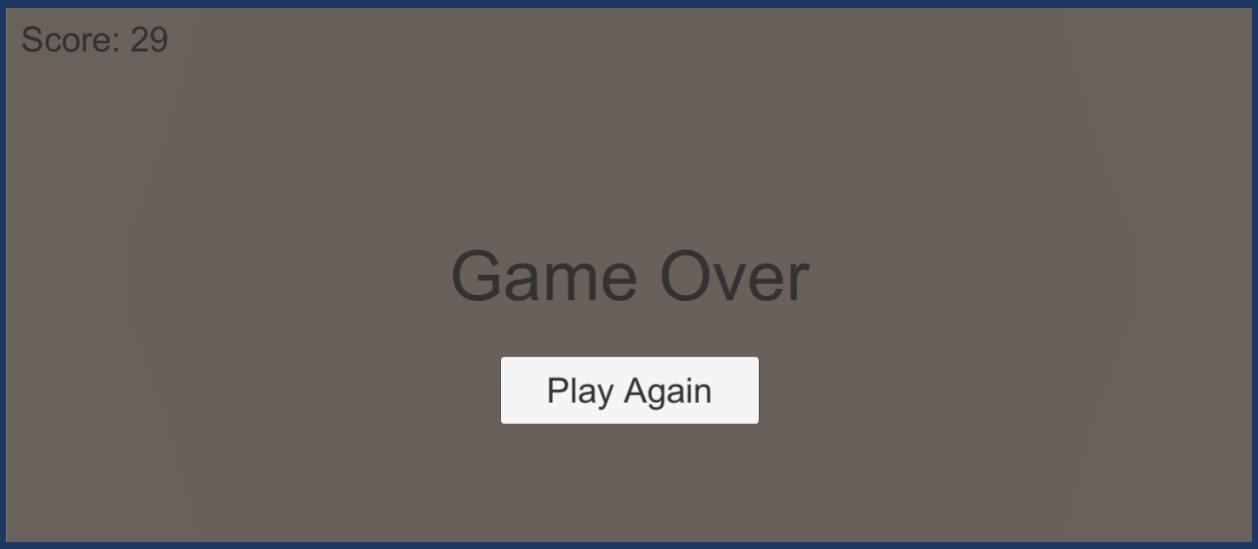
My First Game EnemyController

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

public class EnemyController : MonoBehaviour {

    //Initialize Rigidbody variable
    Rigidbody rb;

    //Initialize start position variable
    Vector3 startPos;

    //Initialize ScoreLabel variable
    ScoreLabel scoreLabel;

    //Initialize speed boost variable
    int speedBoost = 0;

    // Use this for initialization
    void Start () {
        //Get the Rigidbody component attached to this game object
        rb = GetComponent<Rigidbody>();

        //Set the velocity of this game object's Rigidbody component
        rb.velocity = new Vector3(0, 0, 10);

        //Store the start position of the object
        startPos = transform.position;

        //Find the ScoreLabel component in the scene
        scoreLabel = GameObject.FindObjectOfType<ScoreLabel>();
    }

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

        //Check if the enemy is outside the bottom side of the play area
        if (transform.position.z > 30)
        {
            //If it is, reset it's position
            Respawn();
        }
    }

    //This event is triggered whenever the Enemy object collides with something
    void OnCollisionEnter(Collision col)
    {
        //If it hit the player
        if (col.transform.name.Contains("Player"))
        {
            //Get the playerTransform component
            GameObject playerObj = col.gameObject;

            //Get the playercontrols component from the transform
            PlayerControls playerControls = playerObj.GetComponent<PlayerControls>();

            //Display the game over screen
            playerControls.GameOver();
        }

        //If it was hit by a laser
        if (col.transform.name.Contains("Laser"))
        {
            //Increment score
            scoreLabel.IncrementScore();

            //Destroy the laser object
            Destroy(col.gameObject);
        }

        Respawn();
    }

    void Respawn()
    {
        //Reset it's position to somewhere above the play area, randomizing it's x position
        transform.position = new Vector3(startPos.x + Random.Range(-20, 20), startPos.y, startPos.z);

        //Set the velocity
        rb.velocity = new Vector3(0, 0, 10 + speedBoost);

        //Increment speed boost so that every time the enemy respawns, it will go a little bit faster
        speedBoost++;
    }
}
```

Now if we play the game, we will see that the enemies gradually increase in speed over time, and it is much more challenging!



What high score can you get?

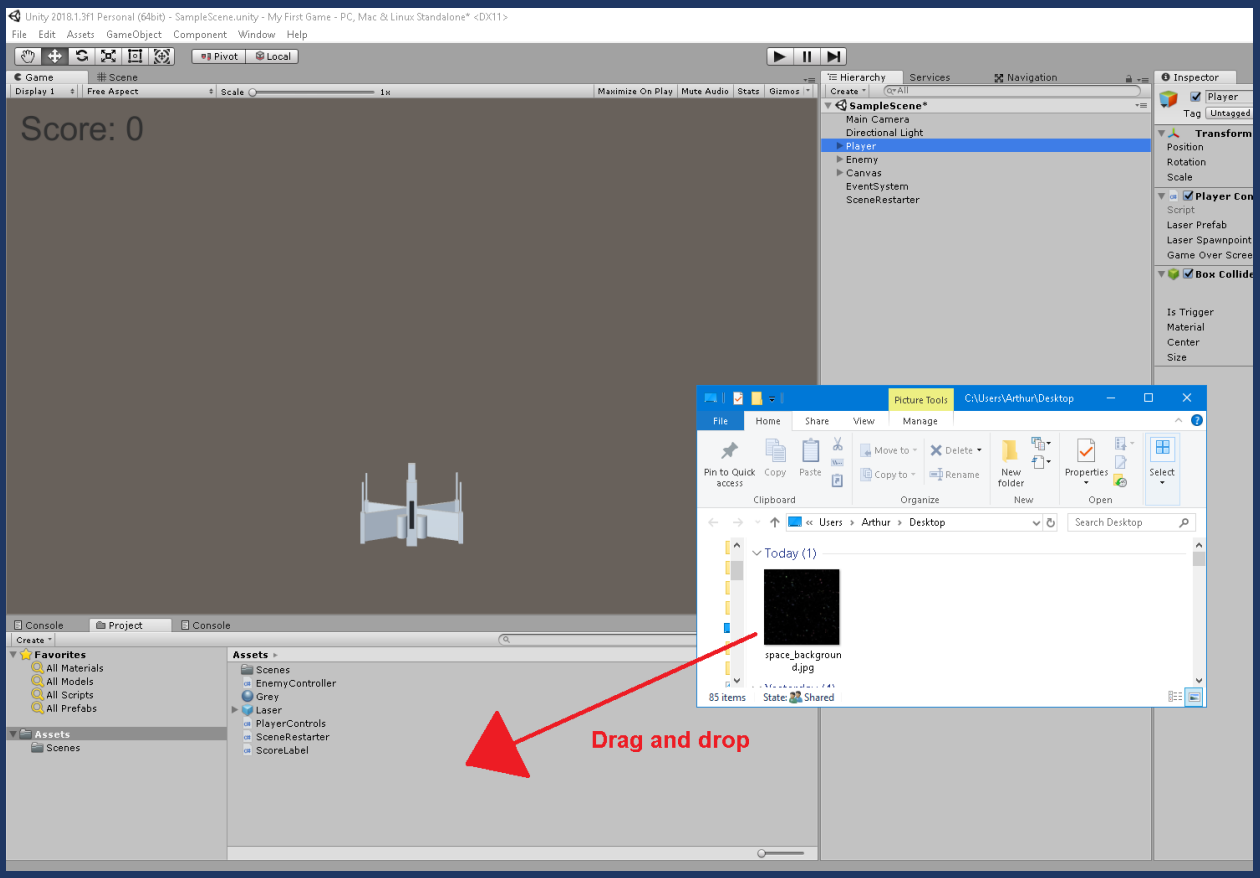


Our game still looks somewhat ugly, because it only has a grey background. (Among a host of other reasons)

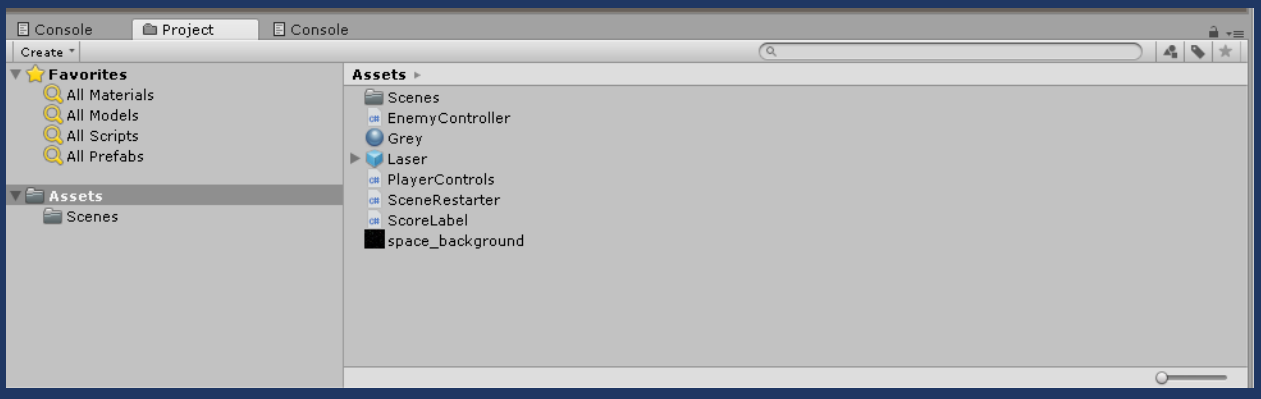
To make it look slightly better, let's make the game have a space background.

You can create your own space background in MS Paint or Adobe Photoshop or the image editing tool of your choice, or for now you could just go to Google images and search "space background" and download one that you'd like to use as a placeholder.

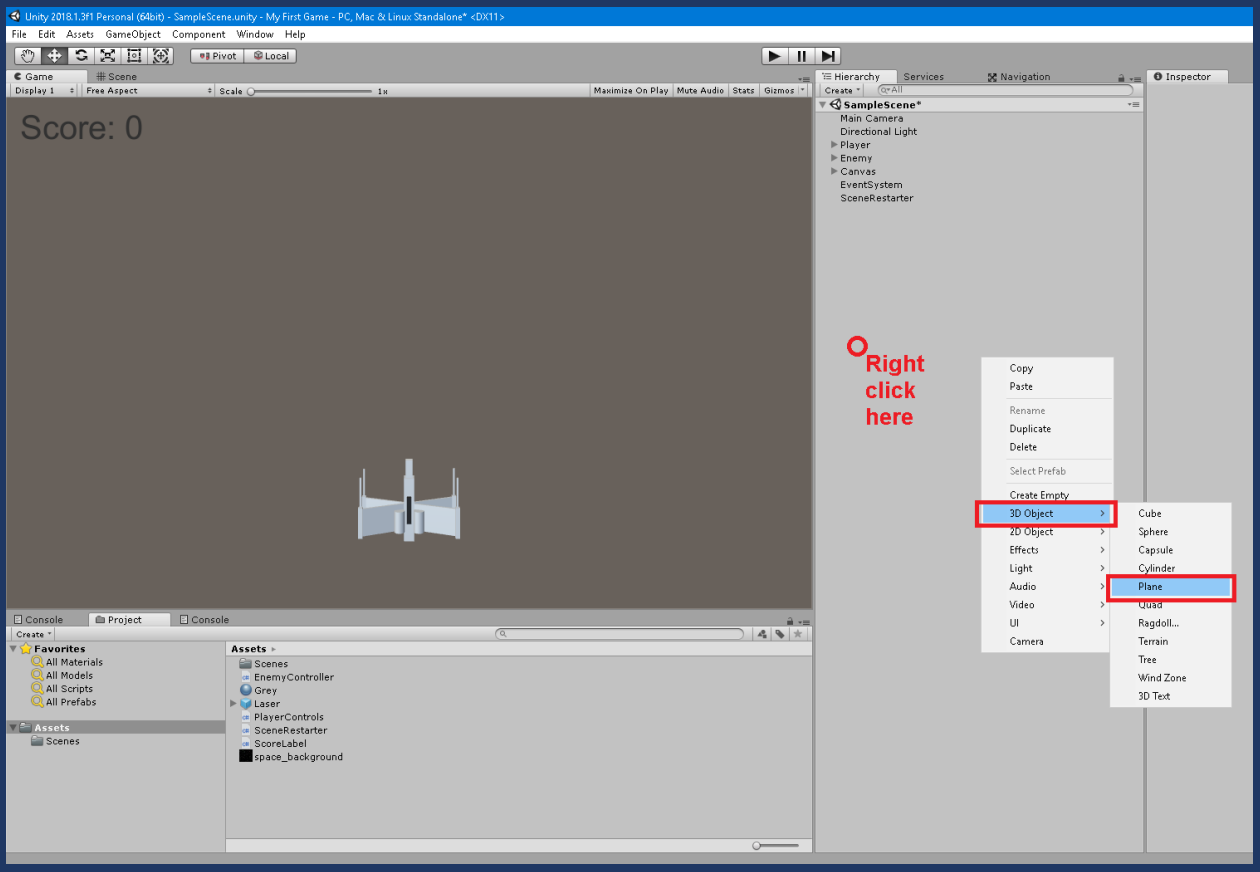
When you find a space background image that you like, you can drag and drop the image file into the Unity Project tab to import it into your project.



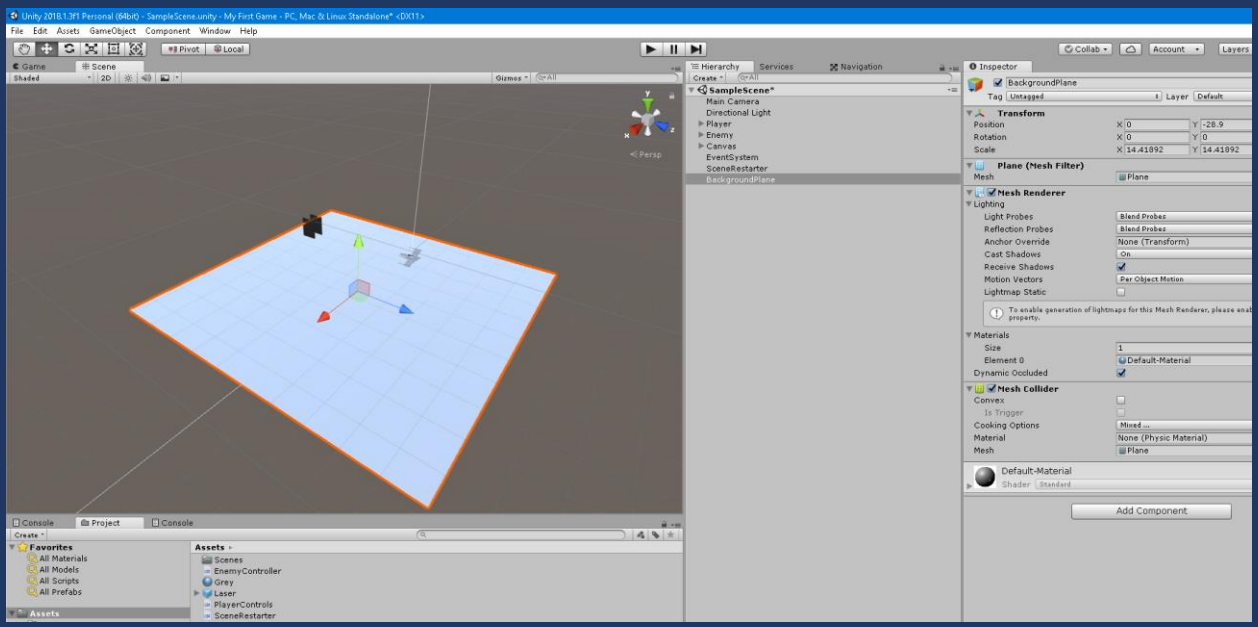
You should now see the space background image appear in your assets folder under the project tab.



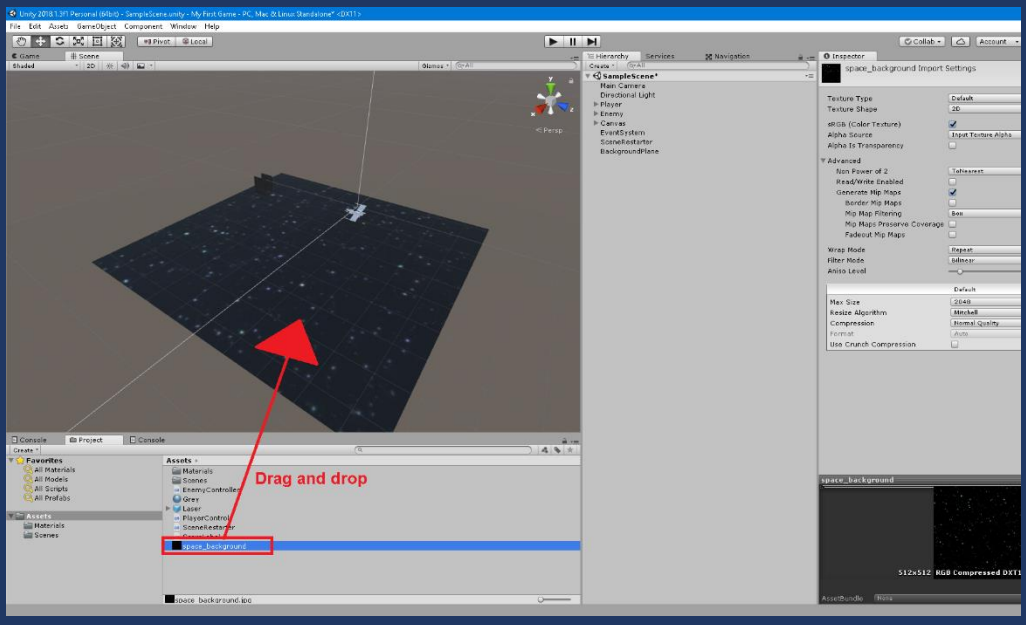
To add this background to the scene, let's create a new flat plane object to display the background. Right click under the hierarchy tab and then click "3D Object > Plane".



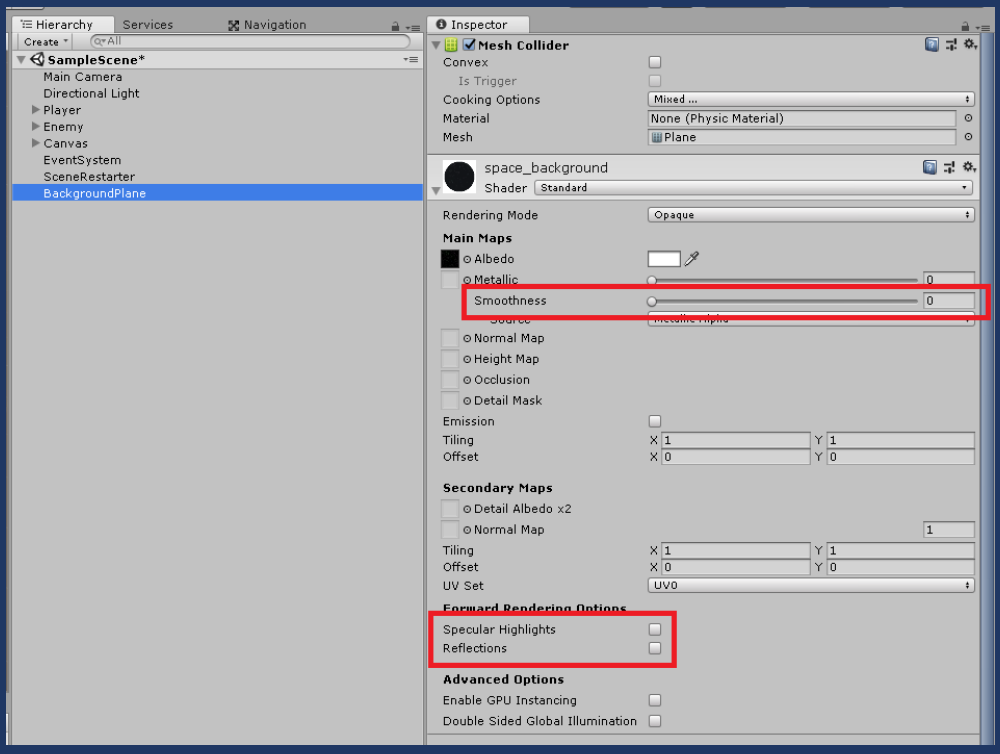
Rename the plane to "BackgroundPlane" and then position it in your scene so that it fills up the background of your game space:



Now drag and drop the space background you imported onto the plane in the scene to apply it as a material.



By default, Unity applies some glossiness to the material which makes it not really look like outer space. To fix this, we can click on the BackgroundPlane and edit the material settings:

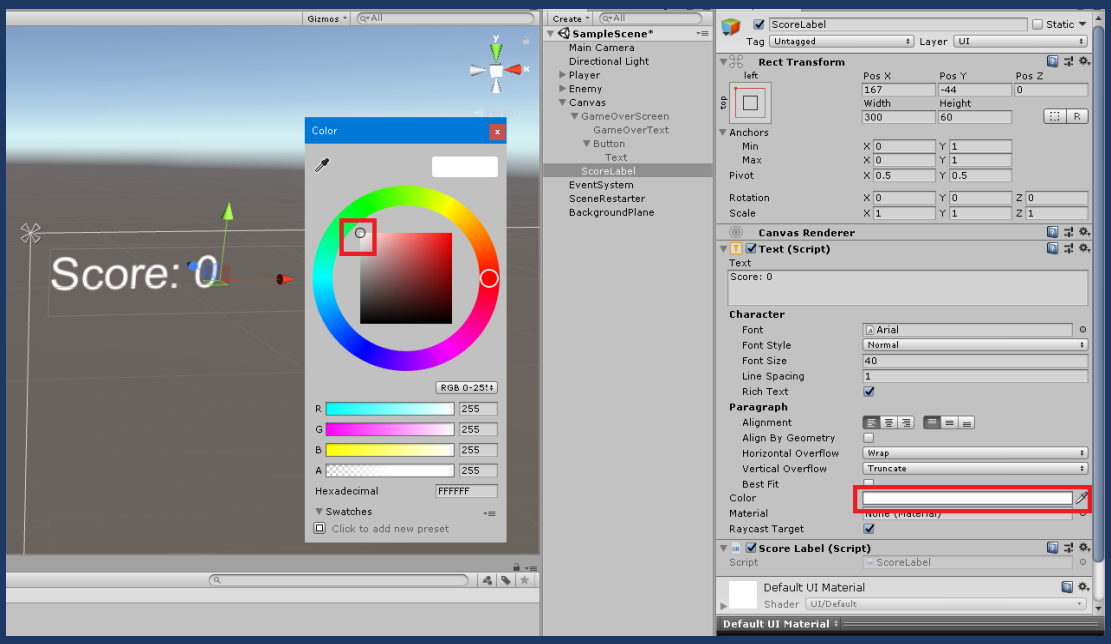


Set “Smoothness” to 0 and disable Specular Highlights and Reflections.

Now if we run the game, we see we have a nice outer space background.

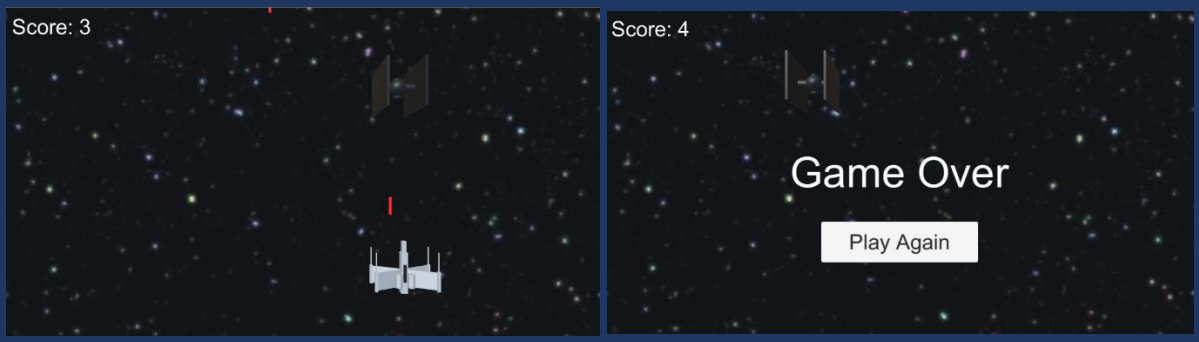
But now both the score text and the game over text are hard to read, because they are so dark.

To fix this, we can go back to our GameOverText and ScoreLabel text objects and change the Text components Color property to be a lighter color:



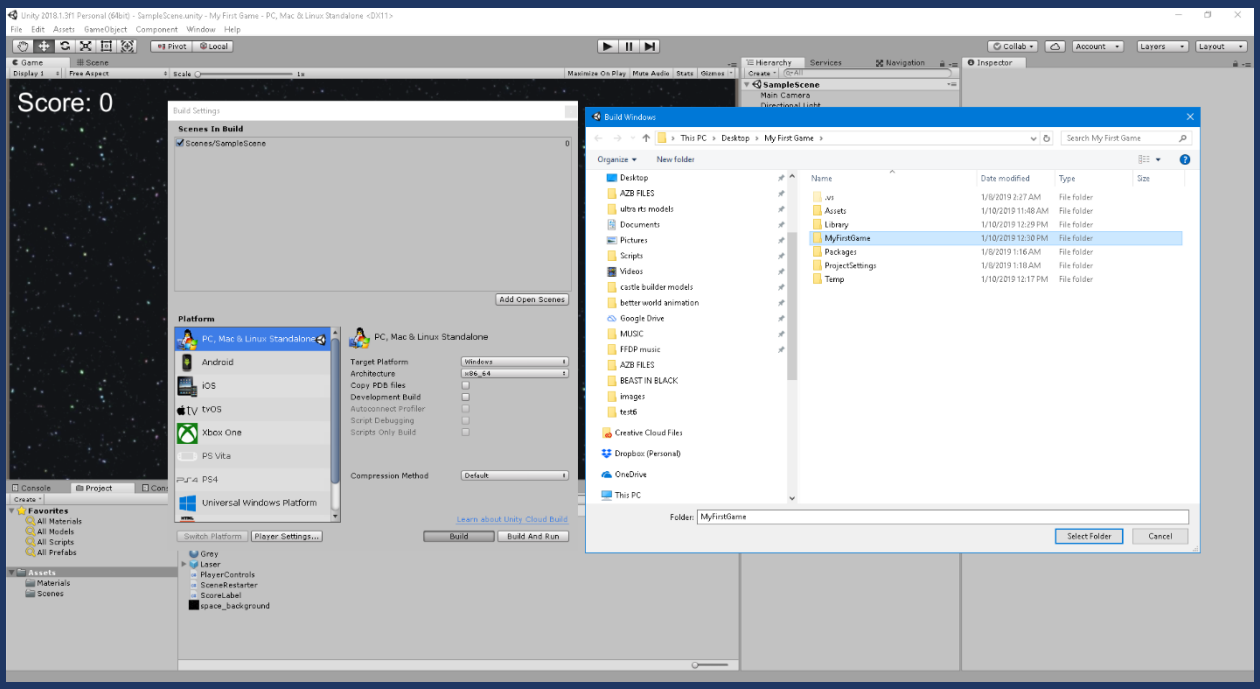
Now if we run the game, it should be much easier to read:

(You can tweak the colors to be whatever you want them to be, doesn't have to be white.)

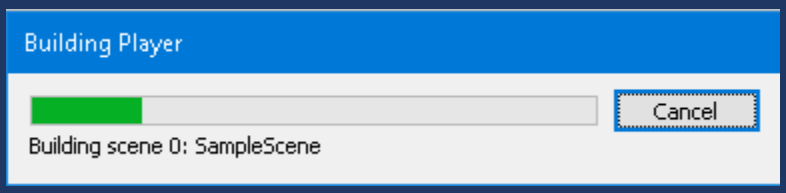


The final thing we need to do to “finish” our game, is build the executable, that is, the file that you can share with people so that they can play your game on their devices.

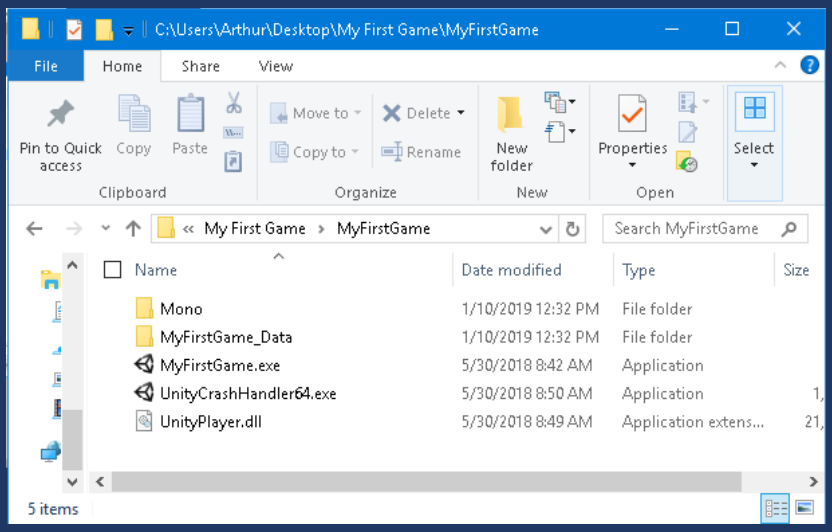
To build the executable file for your game in Unity, click File > Build Settings, and then in the window that pops up, click “Build”. Then create a new folder for the executable of your game, and click “Select Folder”.



The game will take a minute to build…



When it's done, you will have a folder that you can share with people and they can play your game!



You can now put your game on a CD, upload it to the internet, publish it to an app store, or share it however you want! Unity also allows you to build for different platforms, so you can switch platforms and build your game for Android, iOS, Xbox, PlayStation, Virtual Reality headsets, the Hydrogen One, other future LitByLeia devices, and many other platforms!

Congratulations!

You have completed the tutorial and created your first game using Unity!

Thanks for attending my tutorial and I look forward to seeing what cool Unity apps and games you come up with!