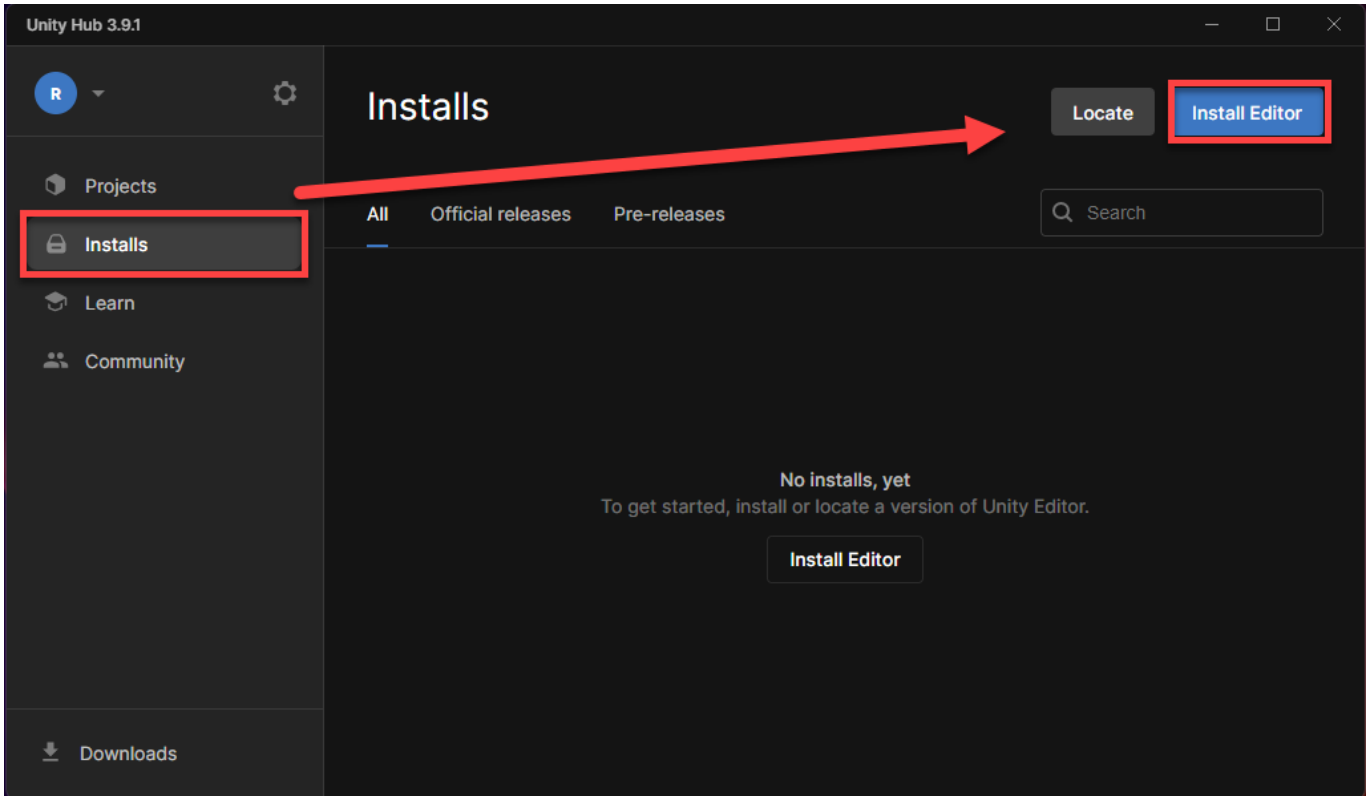
Course Updated to the Unity 6 LTS

We've updated the project files to Unity version **6 LTS** (released October 2024) for this course. This is a long-term support version, which Unity recommends.

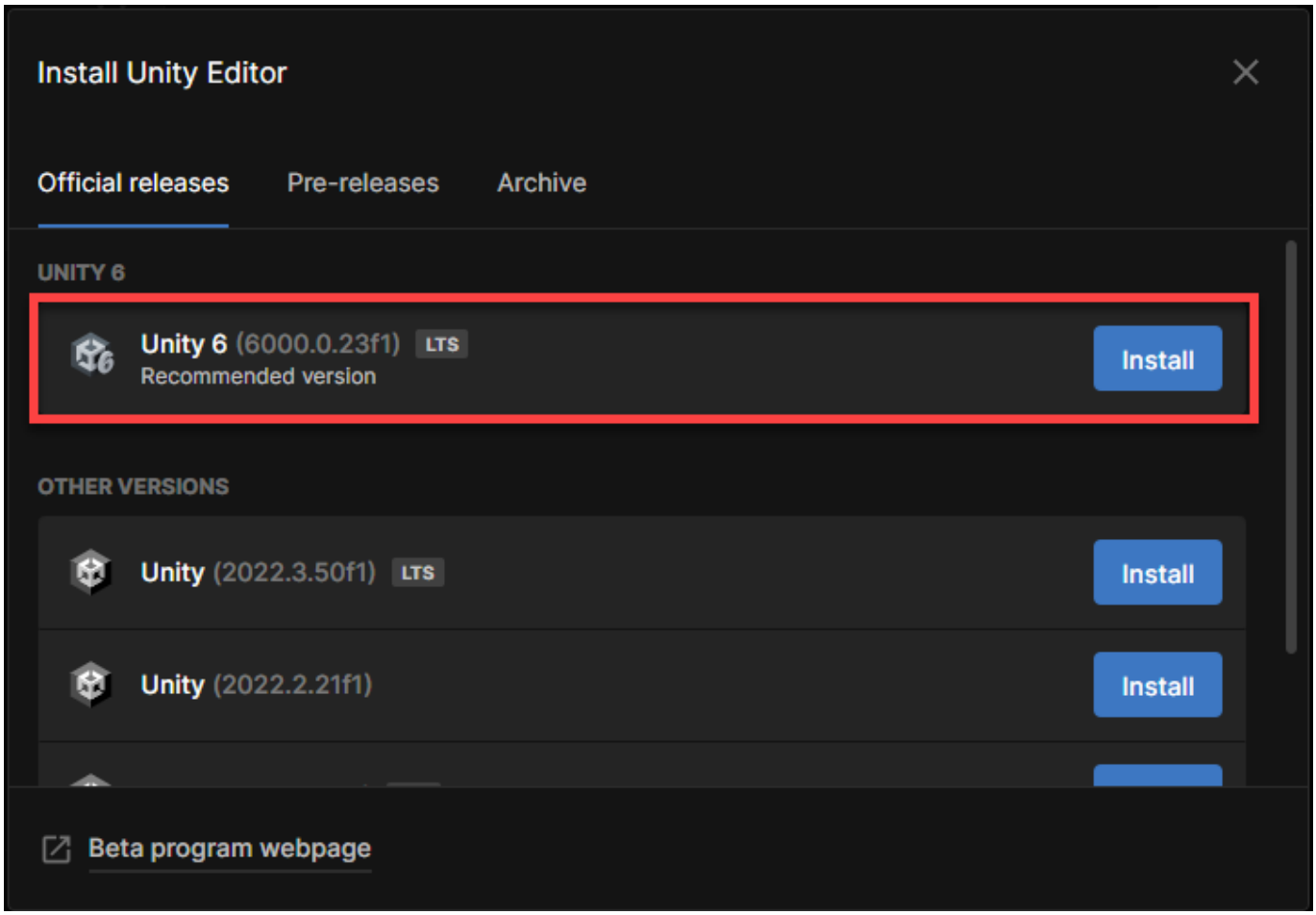
LTS versions are released each year and are a stable foundation for building your projects. For students, this means consistent project files across all courses – no matter when created. So, no more downloading different Unity versions or opening old projects filled with errors!

How to Install the Unity 6 LTS

First, open up your **Unity Hub** and navigate to the **Installs** screen. Then, click on the blue **Install Editor** button.



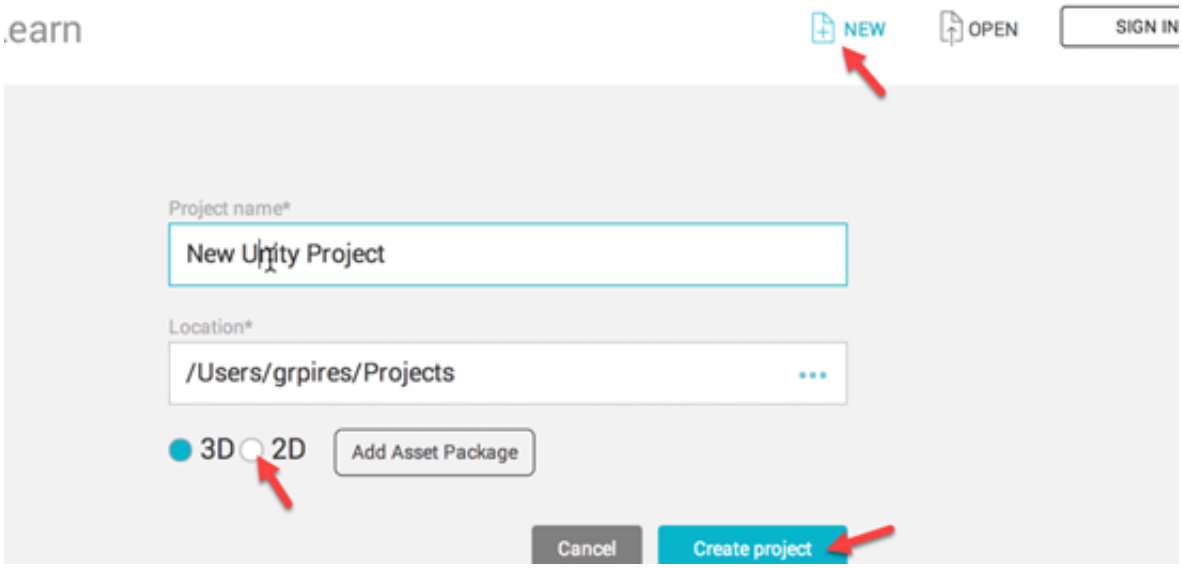
A smaller window with a list of Unity versions will open. At the top of the **Official releases** list, in the 'LONG TERM SUPPORT (LTS)' section, we want to select **Unity 6 [LTS]** (Recommended version). From here, click **Install** and follow the install process.



Setting up the 2D Pong Project

We will be creating a classic Pong game in this course. Pong was one of the first games ever created. The game is very simple mechanically, but there can be some interesting challenges made if you play against a human opponent or even against the computer (Artificial Intelligence).

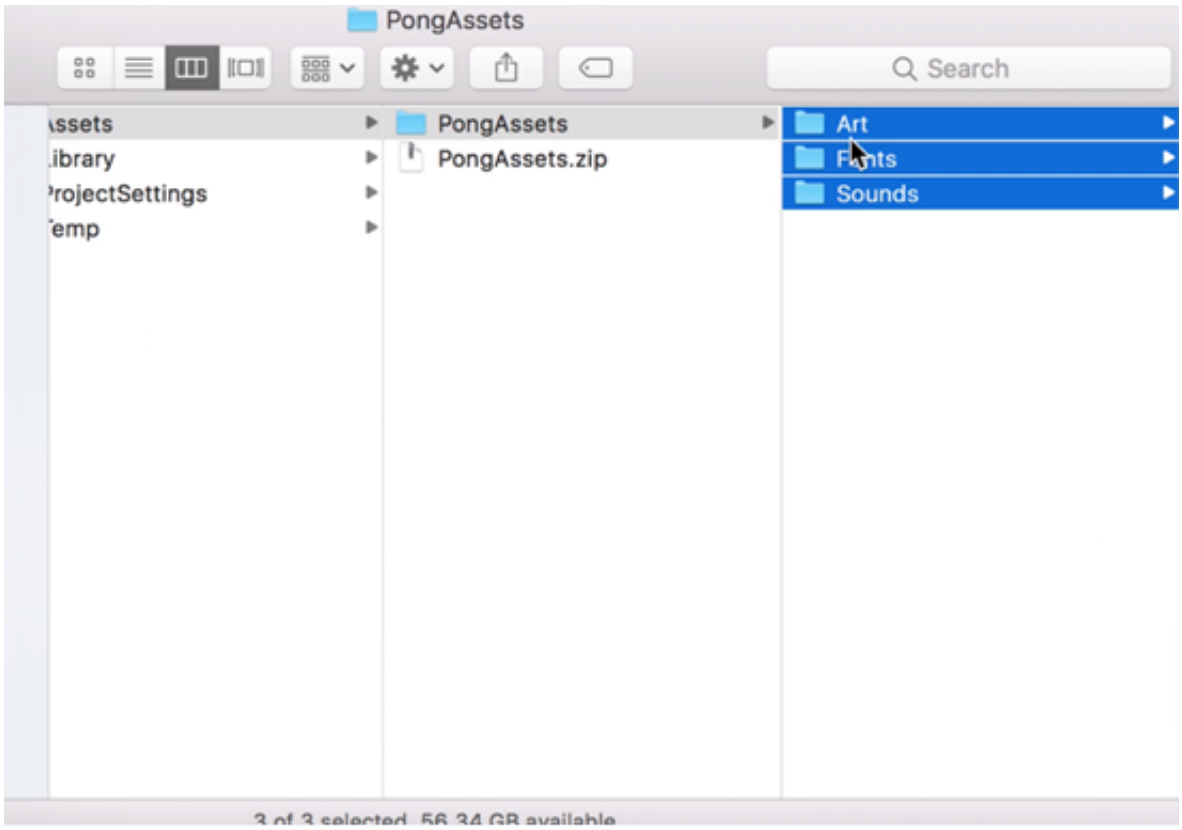
Open Unity and create a new 2D project and name it “Pong.”



Once Unity loads up and you are greeted at the Unity Editor select the **Assets** folder form the **Project** window.

We need to import the Pong Assets that are included for download for this course. Once you download the assets, unzip them and import them into the Assets folder.

Once the assets have been unzipped you will see three folders: Art, Fonts, and Sound.



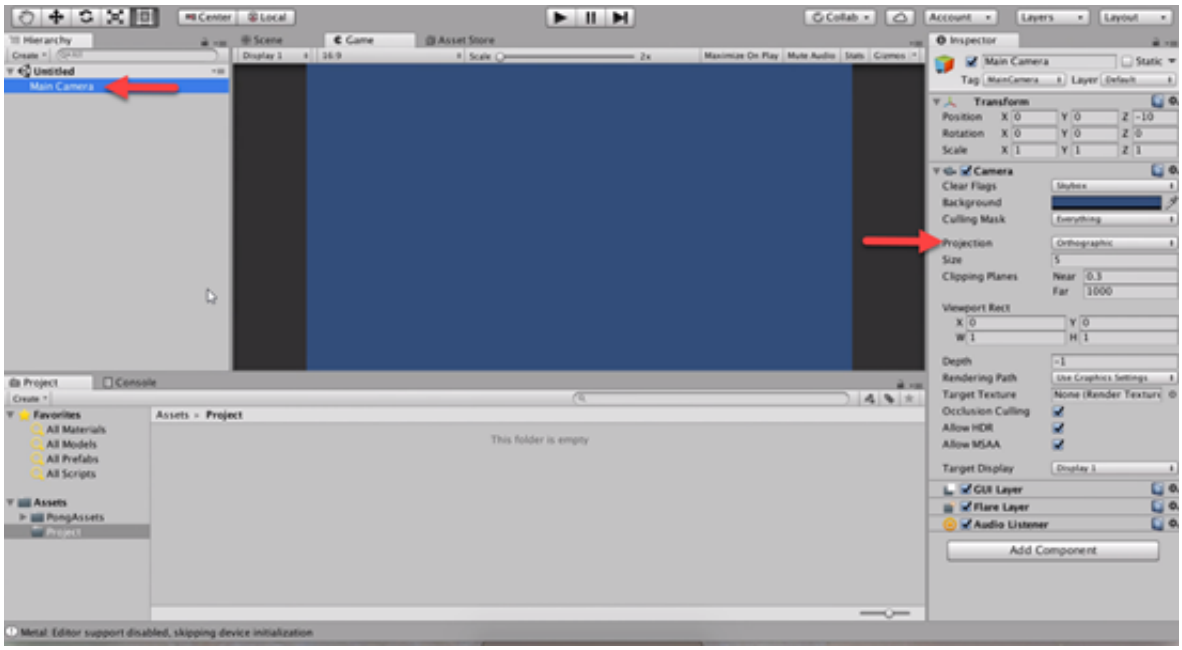
Now back in Unity you should see all the files in the Pong Assets folder. The folder structure should look like this:

It’s important to keep the project organized. This is so its easier to find a particular file without wasting too much time.

Select the Assets folder and right click>Create>Folder. Name the new folder “Project.” This folder can be used to keep any files you might download from the Unity Asset store or any external files made by other people out of your game that you are actually making.

The 2D Camera

Select the **Main Camera** object and look at the Inspector, here you will see that projection mode is set to **Orthographic**.



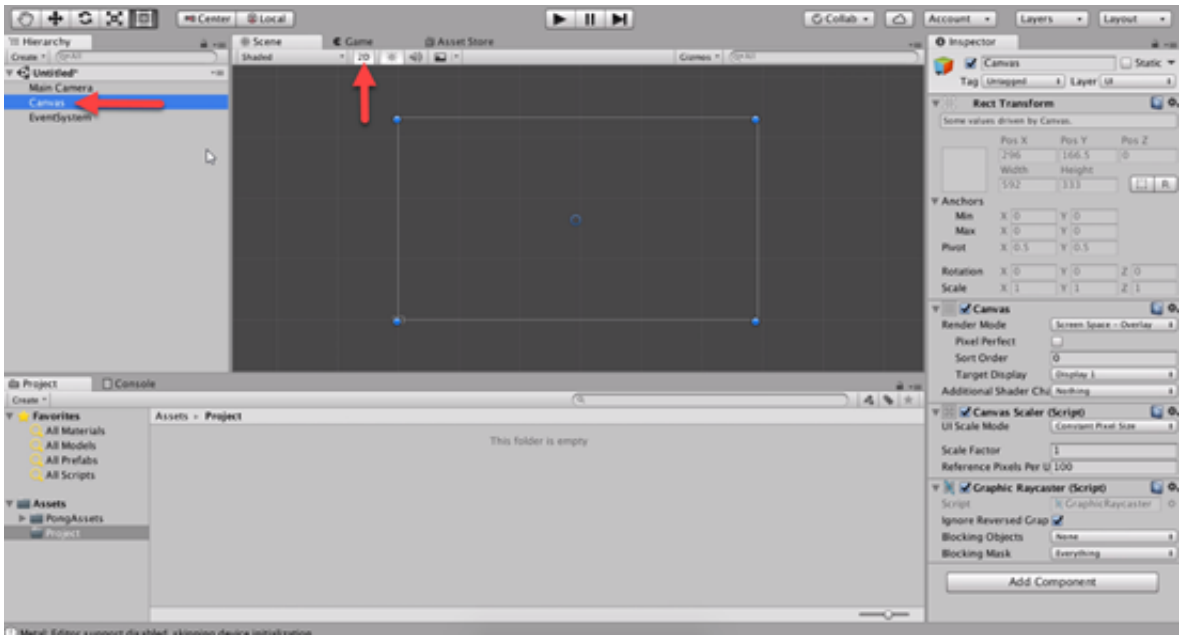
An Orthographic camera projection is used for two dimensional games, there is no perspective applied here in this type of projection.

The **Size** value field can be increased or decreased, right now its currently set to five and we will leave it like that for now. All this value does is increases or decreases whats displayed by the camera.

Canvas Elements

Create a **Canvas** element by right clicking in the Hierarchy>Create>Canvas. This will add a canvas element to the scene. Canvas elements are needed whenever you are going to have user interface elements in your project.

If you select 2D mode at the top of the Scene window you will now see the field of view for the Canvas in the scene. This represents exactly where the user interface elements will be displayed to the user. Since our Pong game will display a score for player one and player two we will be using a Canvas element for this.



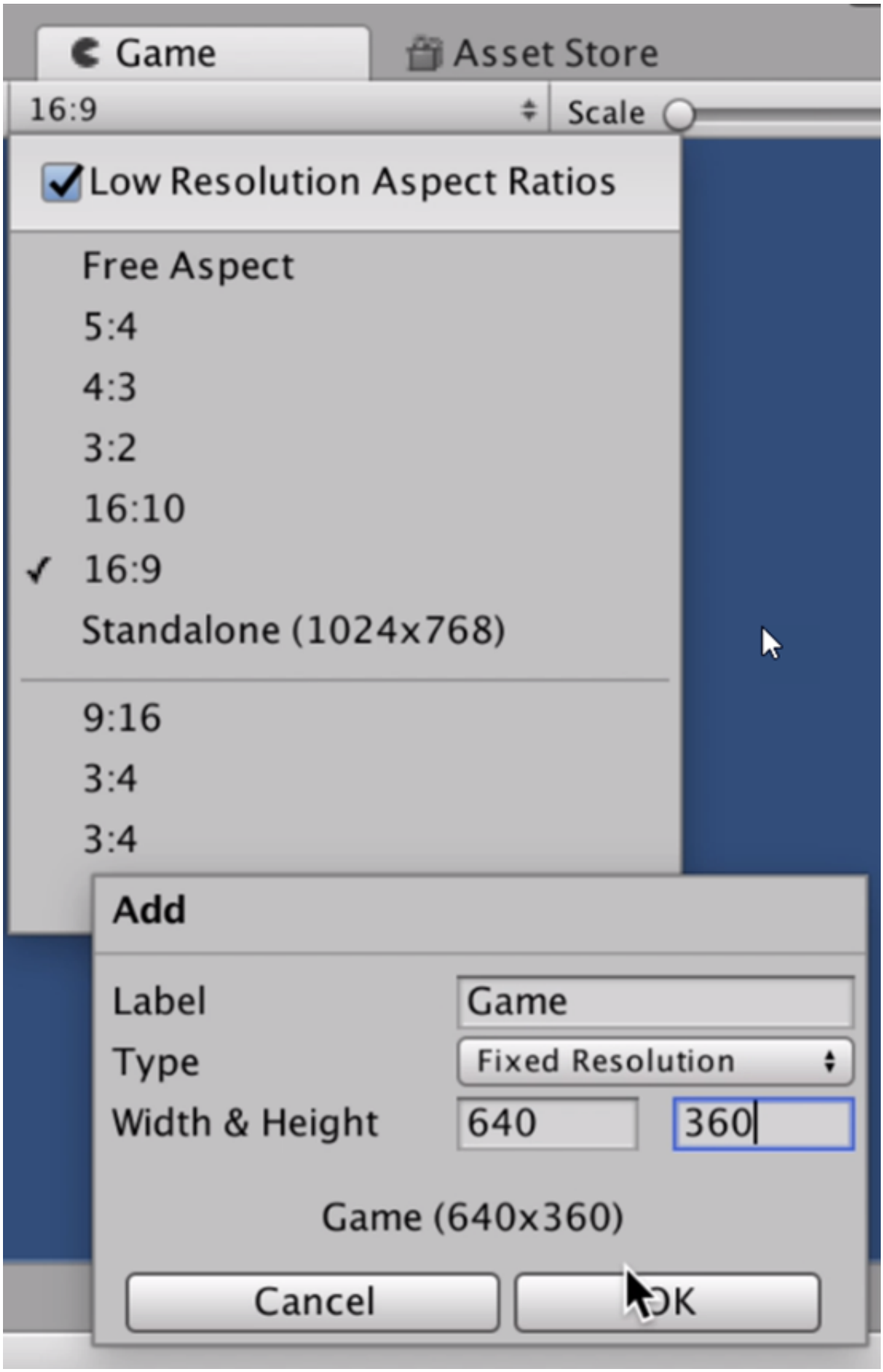
Aspect Ratios

We are going to add a custom aspect ratio. Select the **Game** window and click on the drop down menu for **Aspect Ratio's** right below the Game tab. Click the plus sign icon at the very bottom to add new Aspect Ratio.

In the **Label** field type the word "Game."

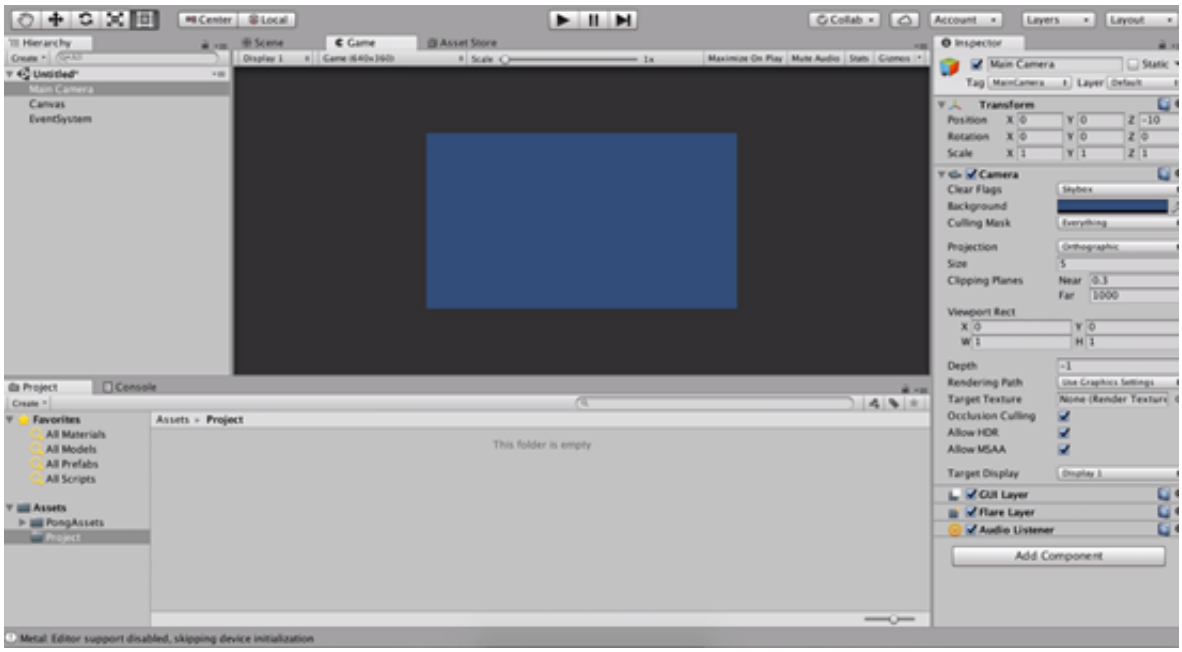
For Type select **Fixed Resolution**.

For the **Width** = 640 and for the **Height** = 360.

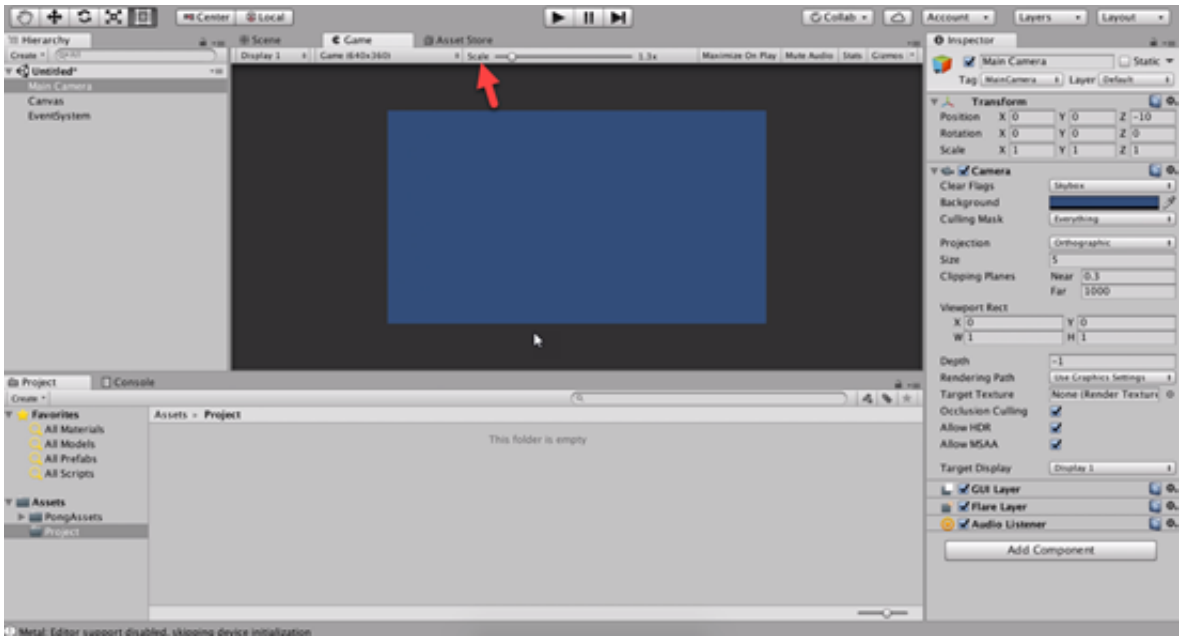


Then press the **OK** button.

The Game window will now look like this:



You can zoom in and out with the Scale slider at the top of the Game window.



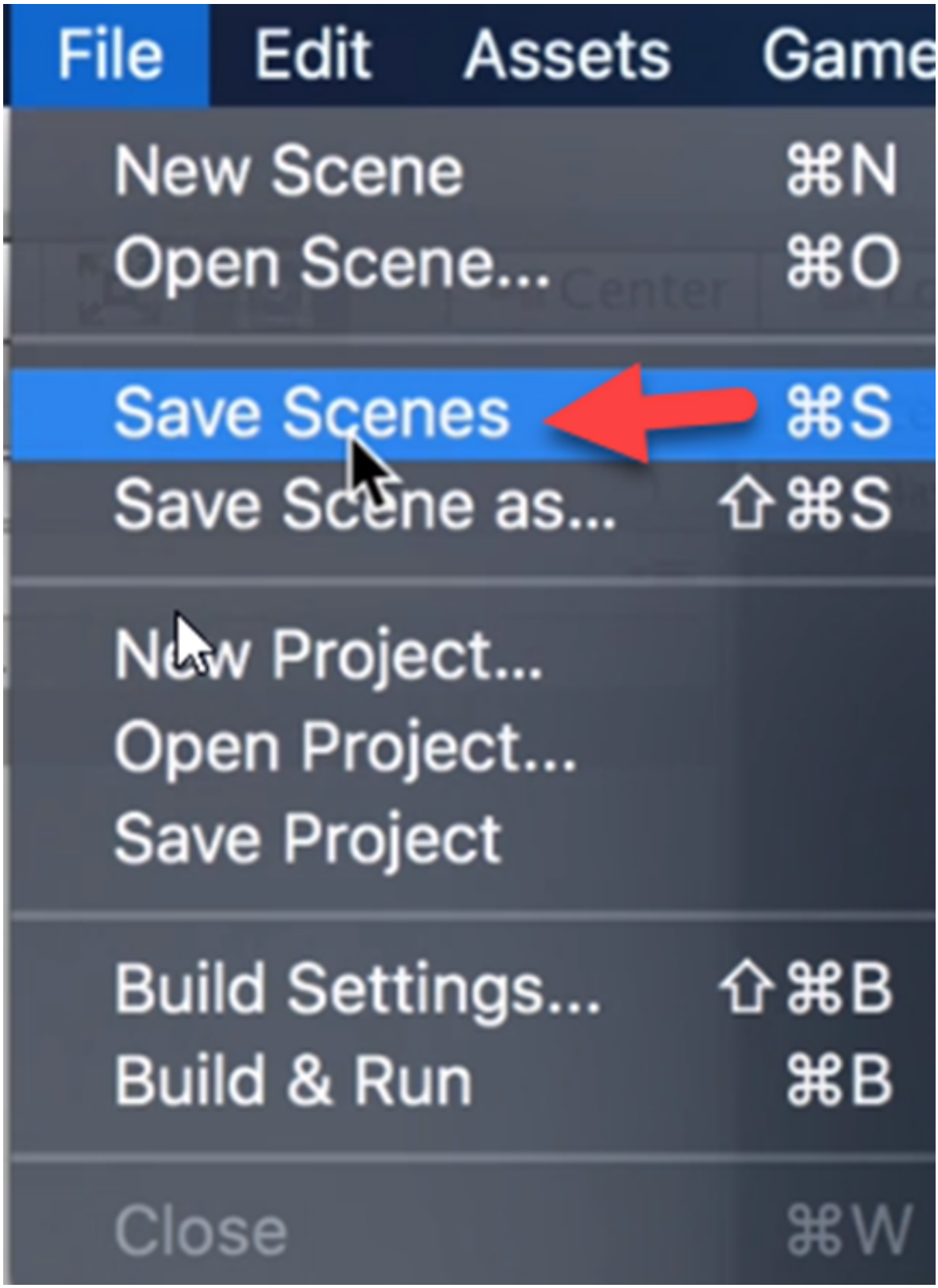
The Aspect Ratio is the ratio between the horizontal and vertical measurements of your screen. The Resolution is the amount of pixels that are going to be rendered.

Make sure your Game window is set to the 640X360 fixed Resolution we setup earlier and you can set the Scale slider to 1.8.

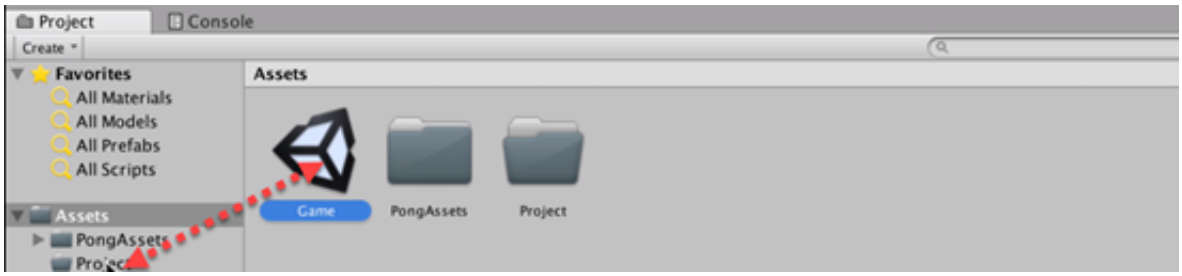
Select the canvas object in the Hierarchy and look at the Inspector, find the component called the Canvas Scaler (Script) and make sure that the UI Scale Mode field is set to **Scale With Screen Size.** Change the X value and Y values under **Reference Resolution** to 640X360. **Screen**

Match Mode needs to be set to **Match Width** or **Height** and **Match** needs to be slid over to match with **Height** since this Pong game is in **landscape** mode. The value can be 1.

Left on File>Save Scenes and name this Scene “Game.”



Select the Assets folder in the Project window and you will see the scene we just saved and named Game there. Select it and drag and drop it into the Project folder.

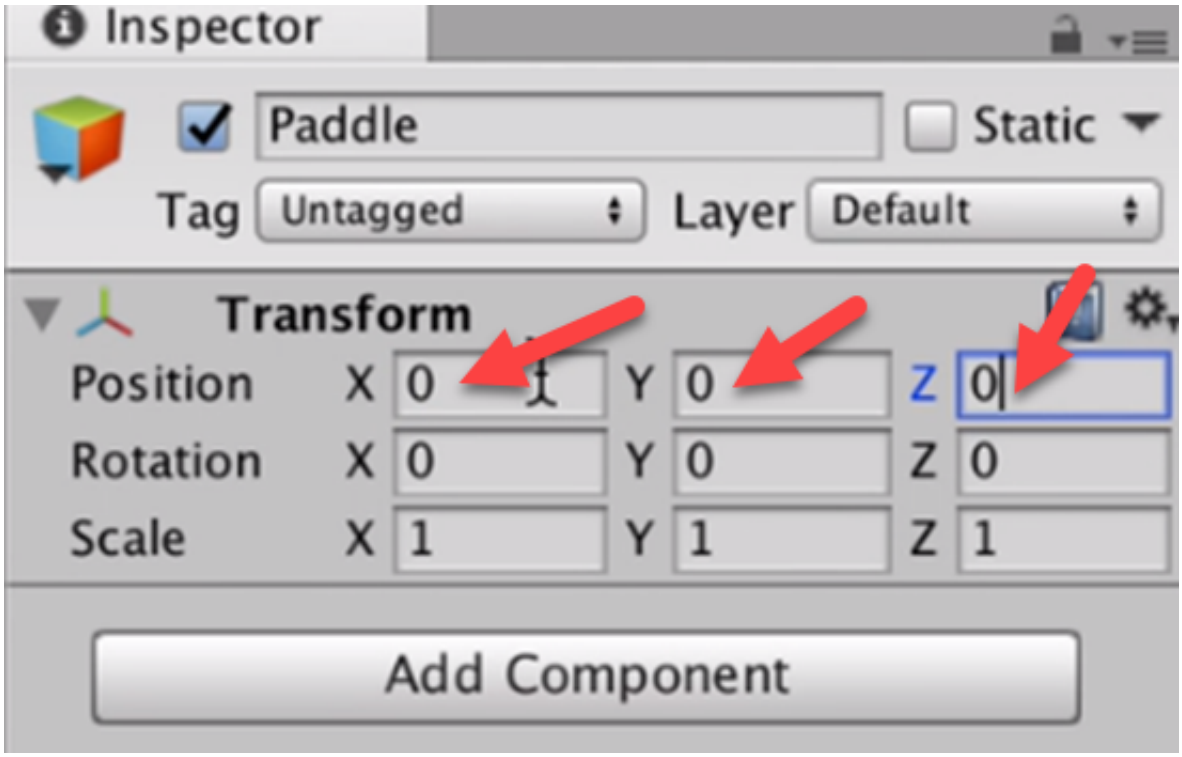


Select the Project folder and you will see the Game scene in this folder now we need to create a new folder and name it “Scenes” we will then move the Game scene into this folder.

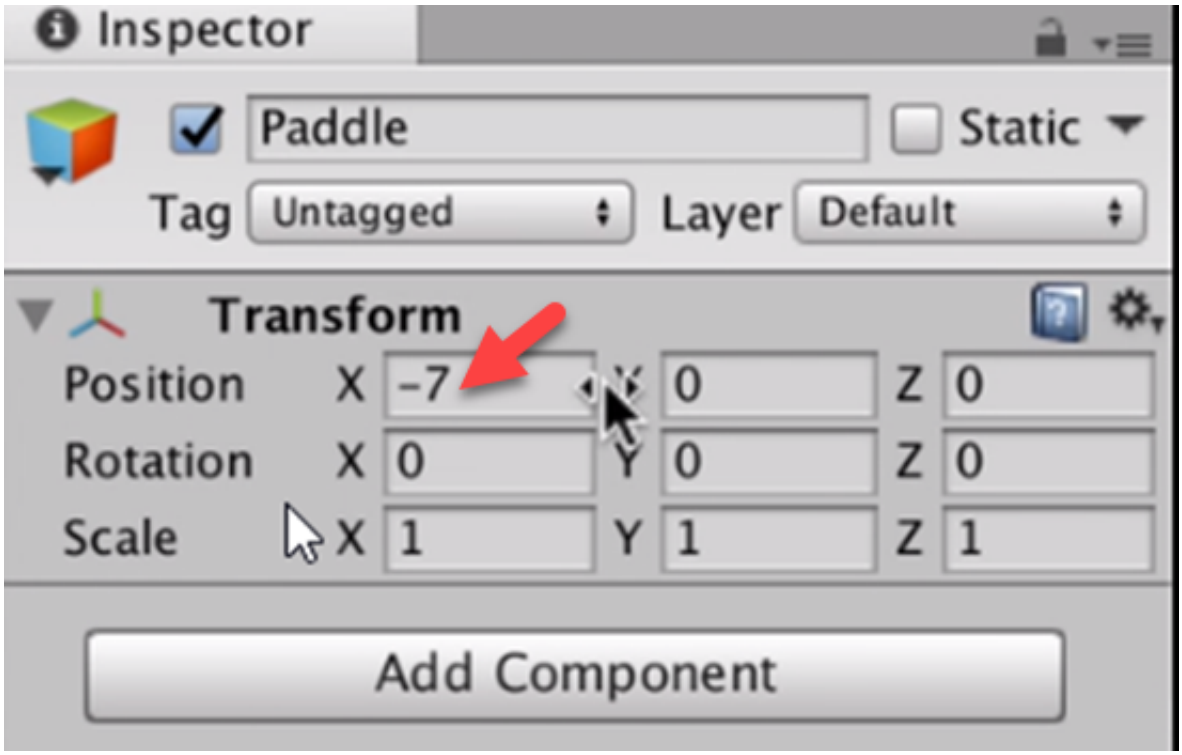
We are now setup and ready to go. In the next lesson we will be working with the Pong paddles. Save the Scene and Project.

Creating the Paddle Game Object

In the **Hierarchy** Right Click>Create Empty and rename this object to **“Paddle.”** Reset the position of the Paddle on the Transform component to 0,0,0.

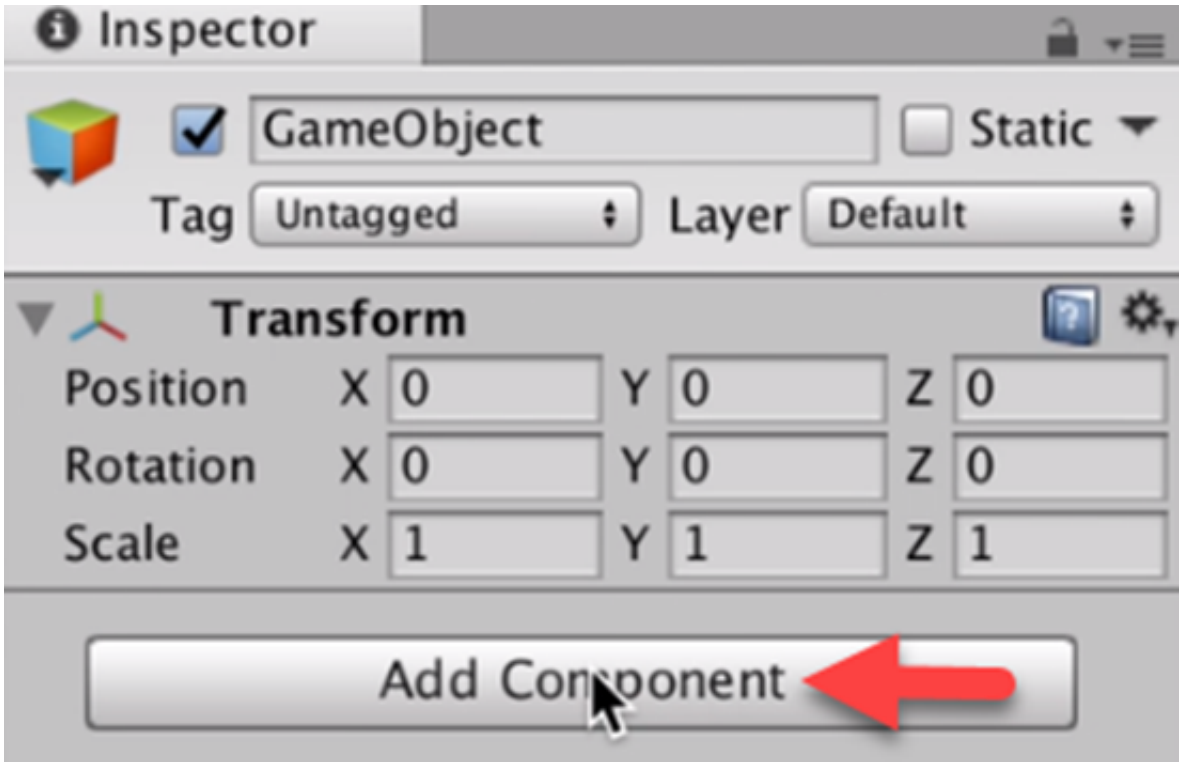


Double click on the **Paddle** game object and the camera will zoom in on the object. We are going to move it over to the left on the X Axis at -7.

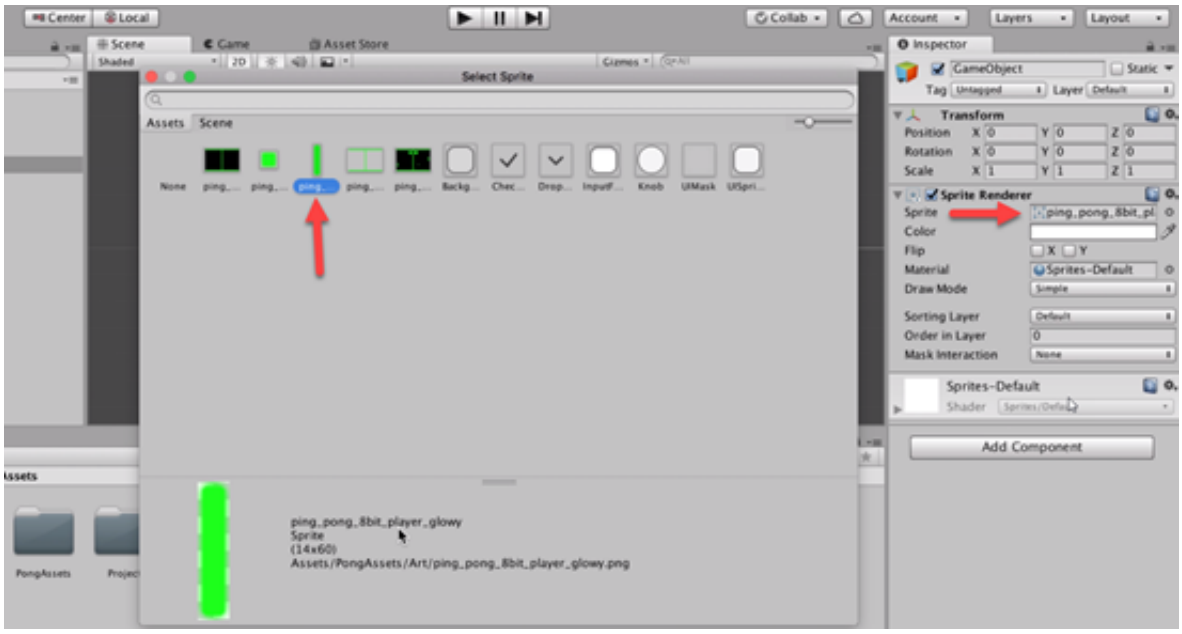


Right Click on the **Paddle** game object and choose create empty.

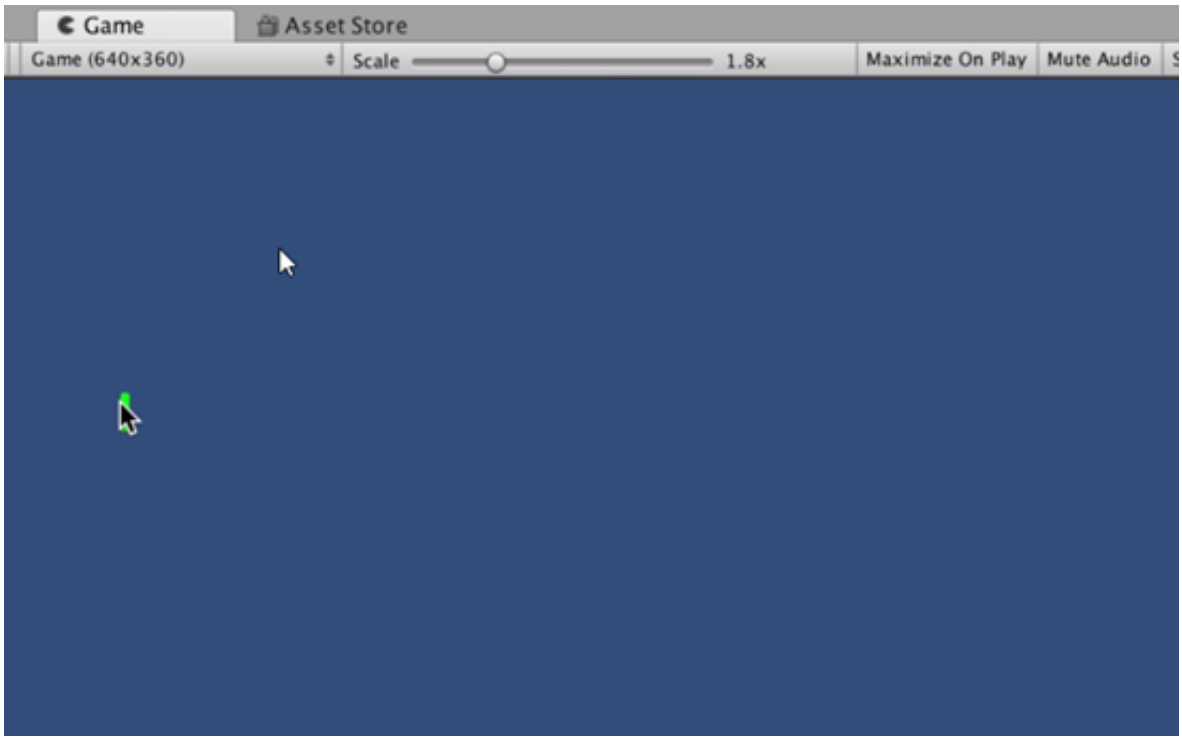
In the **Inspector** window click on the **Add Component** button.



Then type **“Sprite Renderer”** into the search field. This will add a Sprite Renderer component to the empty Game Object. As for the **Sprite**, we want to choose the Paddle sprite we imported from the course materials. Click on the circle button next to the Sprite field. and choose the paddle.



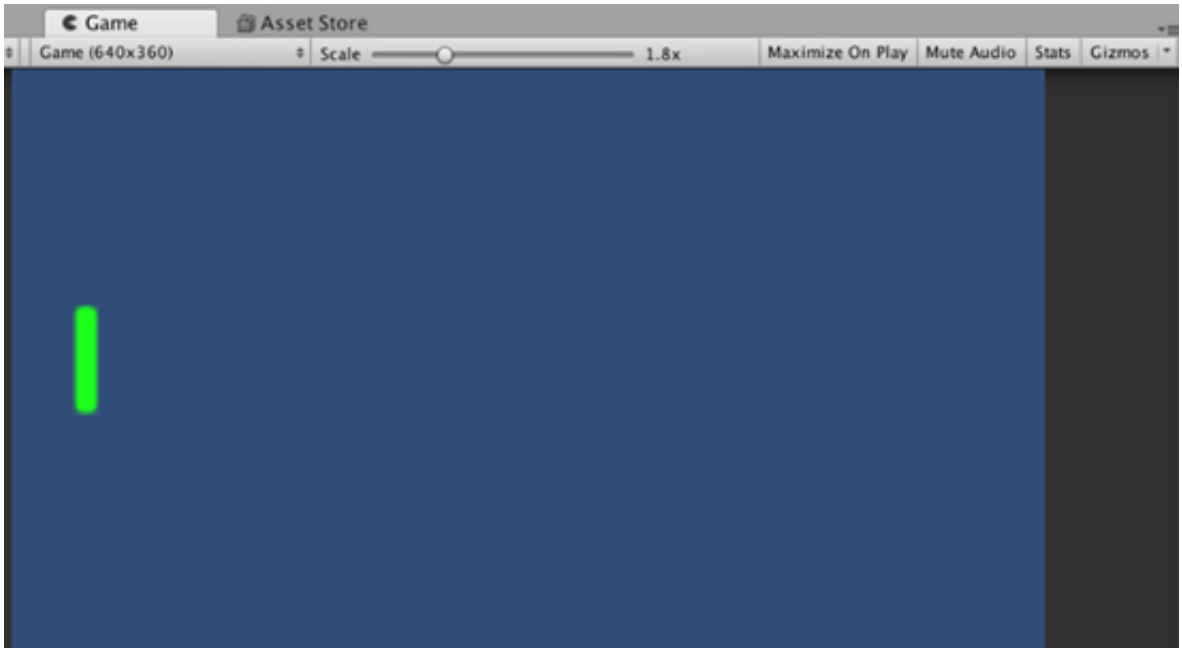
The paddle should now look this in the Game view:



Adjust the **Scale** of the Paddle on the **Transform** component to 2 for X and 2 for Y.

The paddle still looks small so what we can do now to adjust the size of the object is select the Main Camera object and set the Size of the camera to 3.2 instead of 5. This will move the paddle object, but we can just position the paddle at X=-4.97 now.

Take a look at the **Game** view and you will see the paddle object is now bigger and looks better:



Rename the Game Object to **“Sprite.”** So we have the Paddle as the Parent to the Sprite object in the Hierarchy.

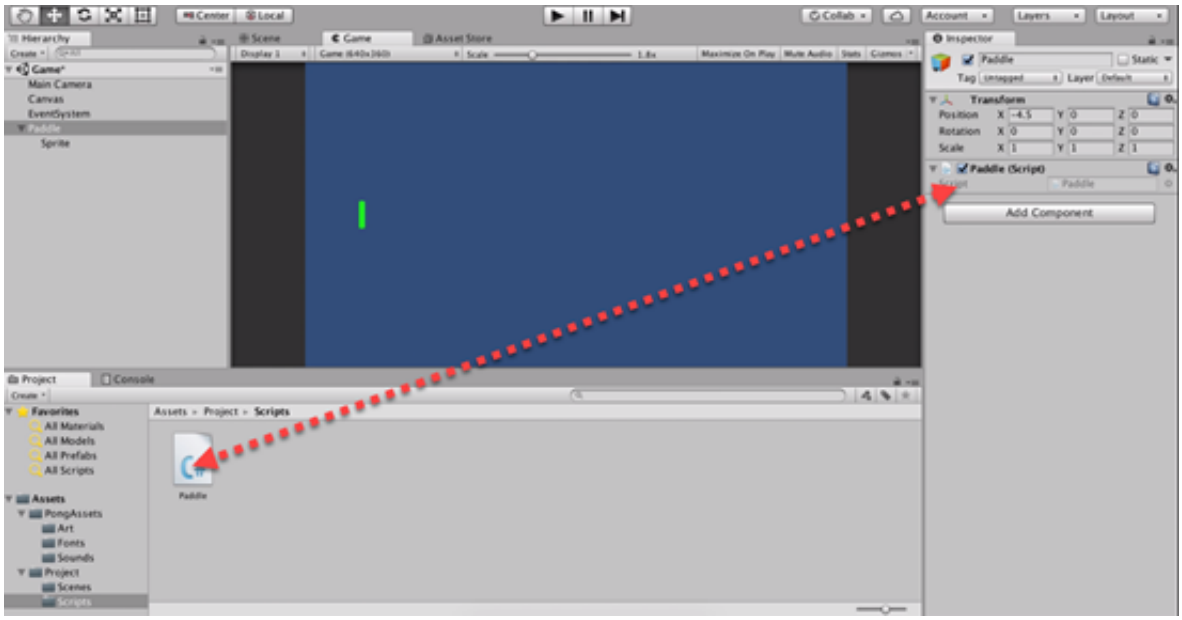
If we control the Paddle object on the Y Axis the Sprite will follow because its a child of the Paddle

object.

The Paddle Script

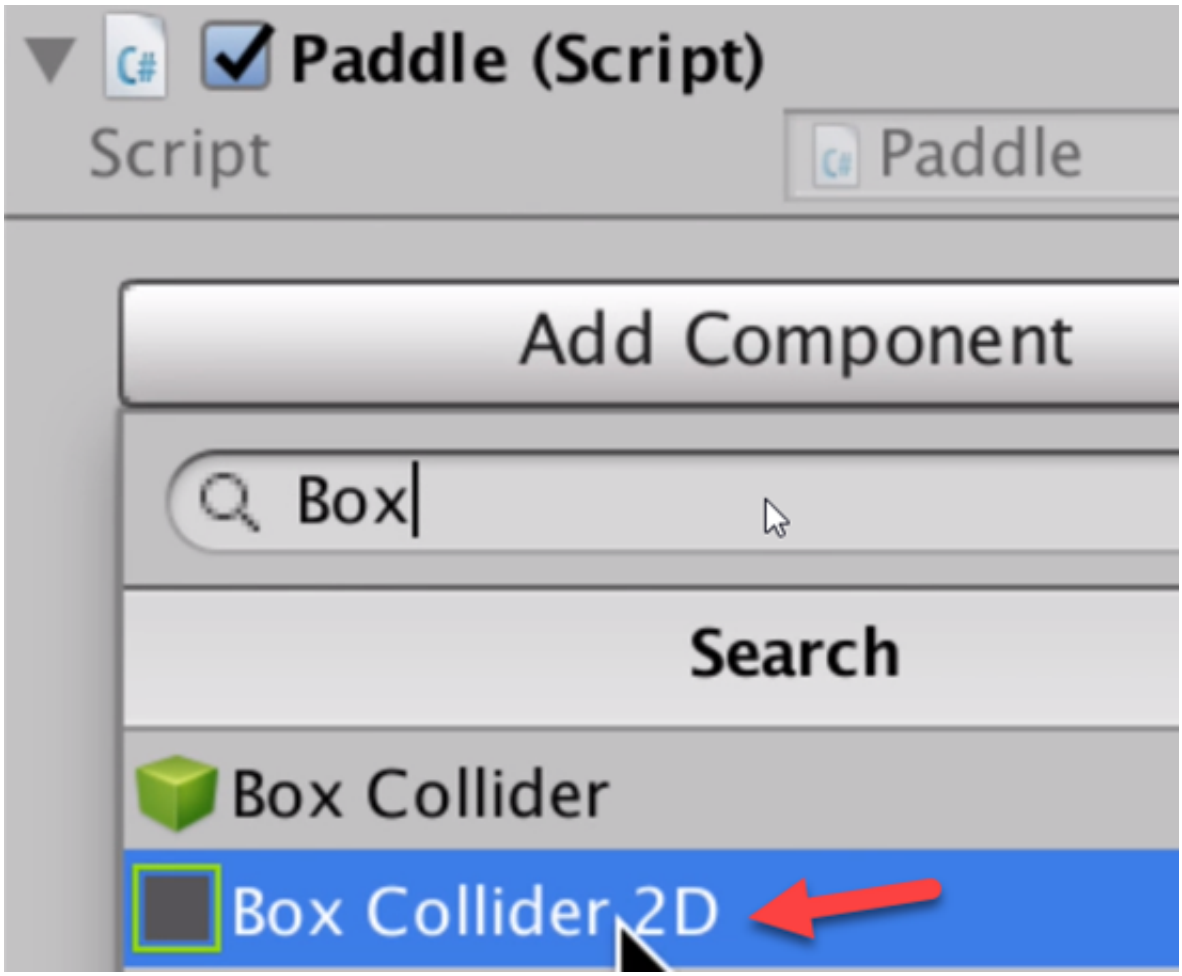
Navigate to the Project folder and create a new folder and name it **“Scripts.”** Double left click on this folder and Right Click>Create>C# Script and name the script “Paddle.”

Select the **Paddle** game object in the Scene and Drag and Drop the Paddle script into the **Inspector** window.



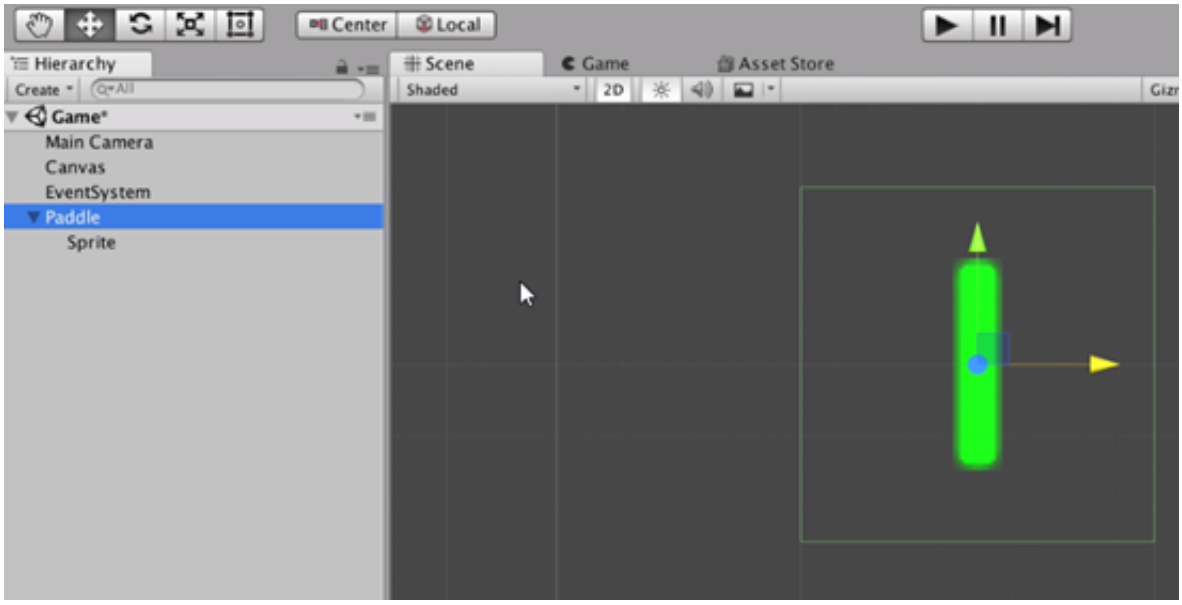
Now the Paddle script is now a component of the Paddle object and whatever we write in this script will work with the Paddle object in our game.

Left click the **Add Component** button and in the search field type the word **“Box Collider 2D”** and select this component to add it to the paddle object.



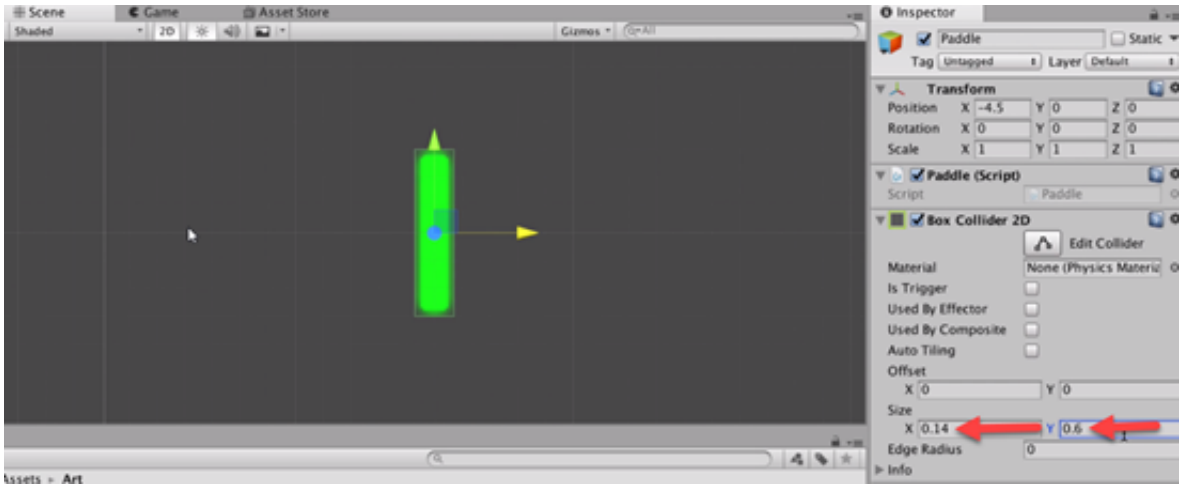
Because were making a 2D game we need to use the 2D version of the **Box Collider**.

Now if you look in the Scene window you will see the green box that now surrounds the paddle sprite and its too big for what we are doing.



We can reduce the size of the **Box Collider 2D** by setting the X value in the size to 0.14 and the Y

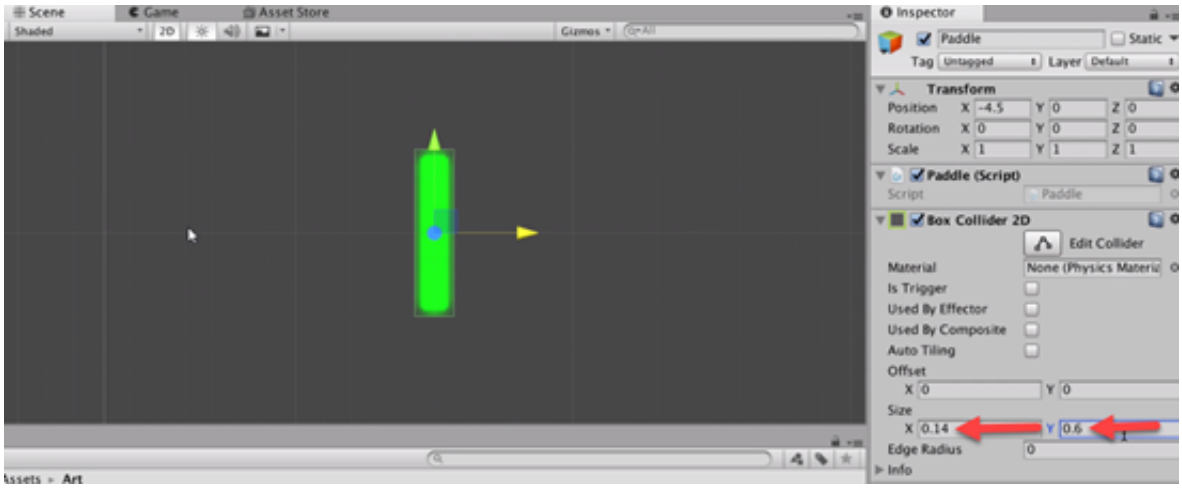
value to 0.60. Now you will notice the box collider covers the sprite perfectly.



Now we need left click on the **Add Component** button again and search for **Rigidbody 2D** and add that component to the paddle object.

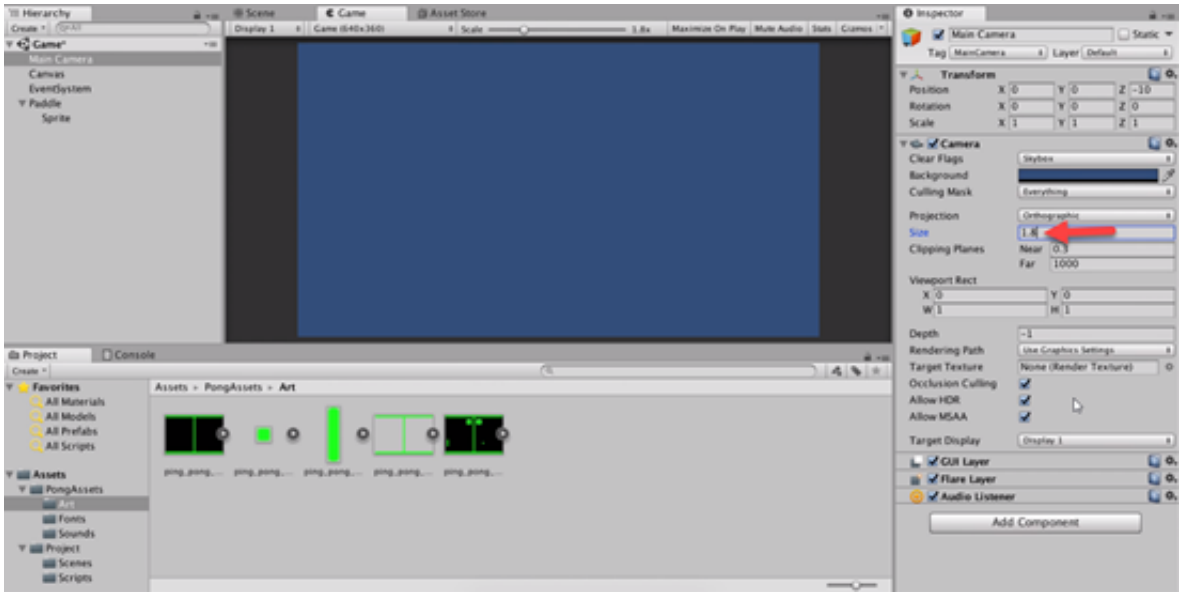
This component is what activates Physics.

We need to make one more adjustment on the **Rigidbody** component, we need to set the **Gravity** to 0 instead of 1 so that are paddle doesn't fall down.

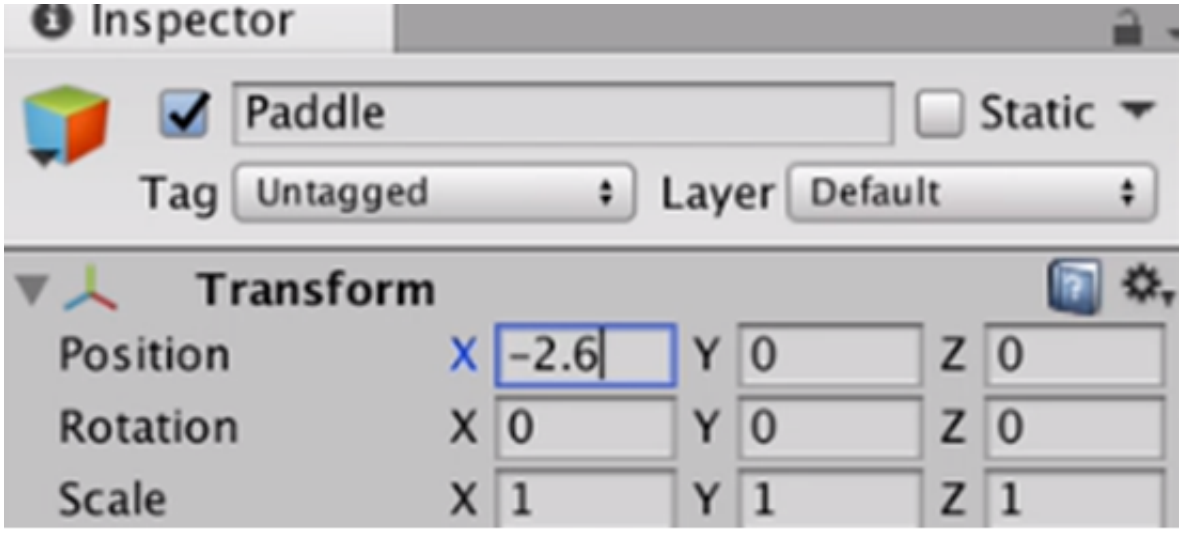


Save the Scene and we can now move onto the next lesson.

There is one more thing we need to change before we move on with this lesson. In the **Main Camera** we are using the size 3.2, but we have to remember that this game we are making will be using the height of the scene as the resolution reference. So, instead of using half of the **horizontal resolution** we should be using half of the **vertical resolution**. 360 pixels translates in Unity to 3.6 units, and half of that is 1.8. So we need to adjust the size to 1.8 instead of the 3.2 we had there before.



Then we will need to select the **Paddle** game object and adjust the **position** on this object to 2.6 on the X value.



Adding Code to the Paddle Script

Open up the **Paddle** script in the code editor. We want to move the paddle up or down depending on what key we are pressing.

We want to create an if statement that checks for input, and we can put this in the Update function.

```
if(Input.GetAxis("Vertical")) != 0){
    Debug.Log(Input.GetAxis("Vertical"));
```



```
}
```

Save the script and open the Unity Editor back up. If you hit the Play button and test out the changes we made to the script you will see in the Console window the number 1 as you press the up arrow on the keyboard.

If you release the up arrow you will see the number start to decrease slowly to zero, and if you press the down arrow you will see the number change to -1.

There is a way to simplify this code we have written with a float variable. Open the script up in the code editor.

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

GetComponent<Rigidbody2D>().velocity = new Vector2(0, verticalMovement);
```

Save the script and go back to the Unity Editor and press Play to test the changes. You will see that when you press the up arrow the paddle moves upwards and when you press the down arrow the paddle moves down.

You can even use the “W” and “S” key to move the paddle up and down.

You might think the paddle is moving to slow, and we can fix this by adding a public variable in the script. Open the Paddle script back up in the code editor.

```
public float speed = 1f;
GetComponent<Rigidbody2D>().velocity = new Vector2(0, verticalMovement * speed);
```

Save the script and go back to the Unity Editor, and on the paddle game object if you look at the script component in the Inspector you will see the Speed variable now listed. Because, we made the variable public in the script we can make changes to it right here in the editor by just changing the value from 1 to 3.

Hit the Play button and test out the changes, you will see that the paddle is now moving at a faster speed.

Save the scene and project and you can move on to the next lesson.

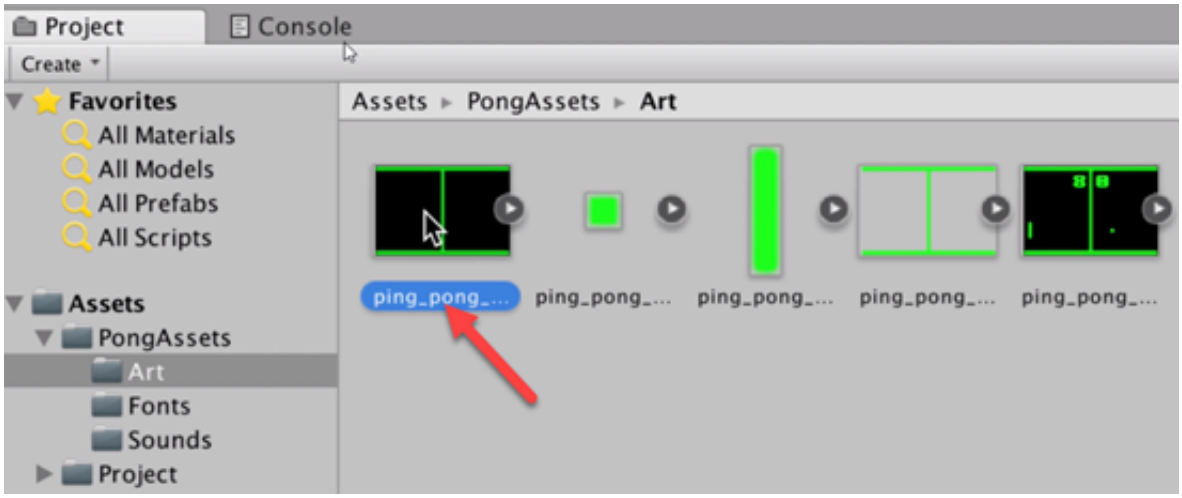
Testing the Limits of Newly Added Features

Every time you add a new feature to your game you should always test the limits.

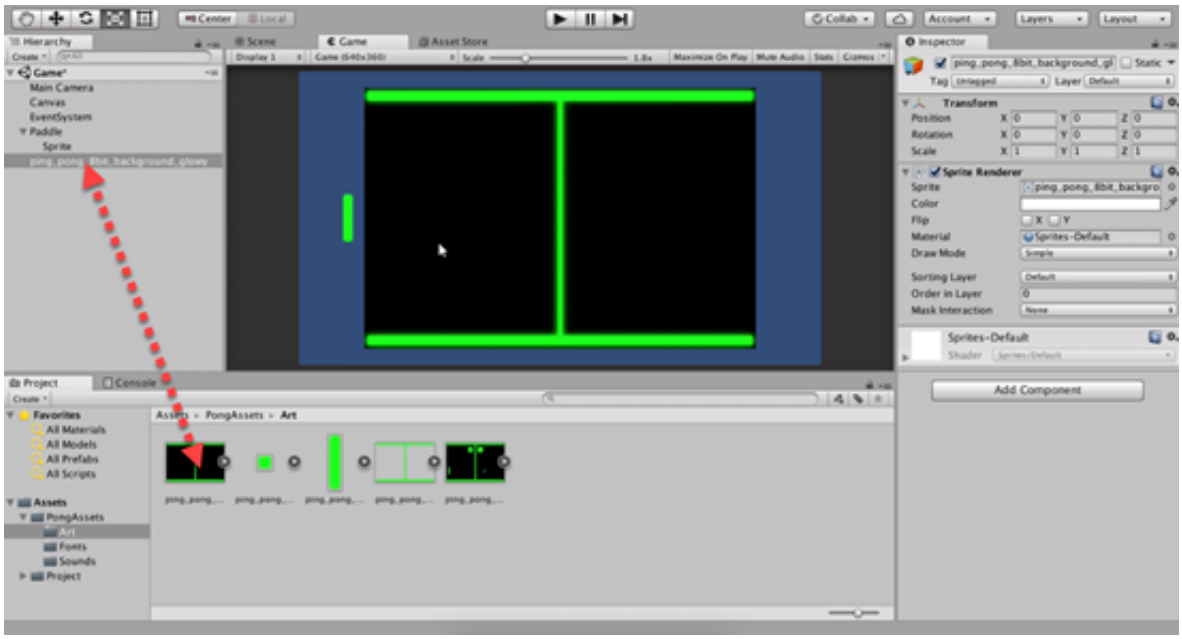
Hit the **Play** button and keep holding up, the paddle keeps moving upwards and is no longer visible because it has moved out of the display area.

So now we need to make the **bounds** so that the paddles will not be able to leave the scene area. One thing that will help us here is the background in the art folder of the project.

Navigate to the **Assets** folder and then the **Art** folder, open the folder and select the **background image**.



We are going to simply just drag this background image right into the **Hierarchy** and drop it. This is different from before where we used the empty game object and attached the Sprite Renderer and added it that way.



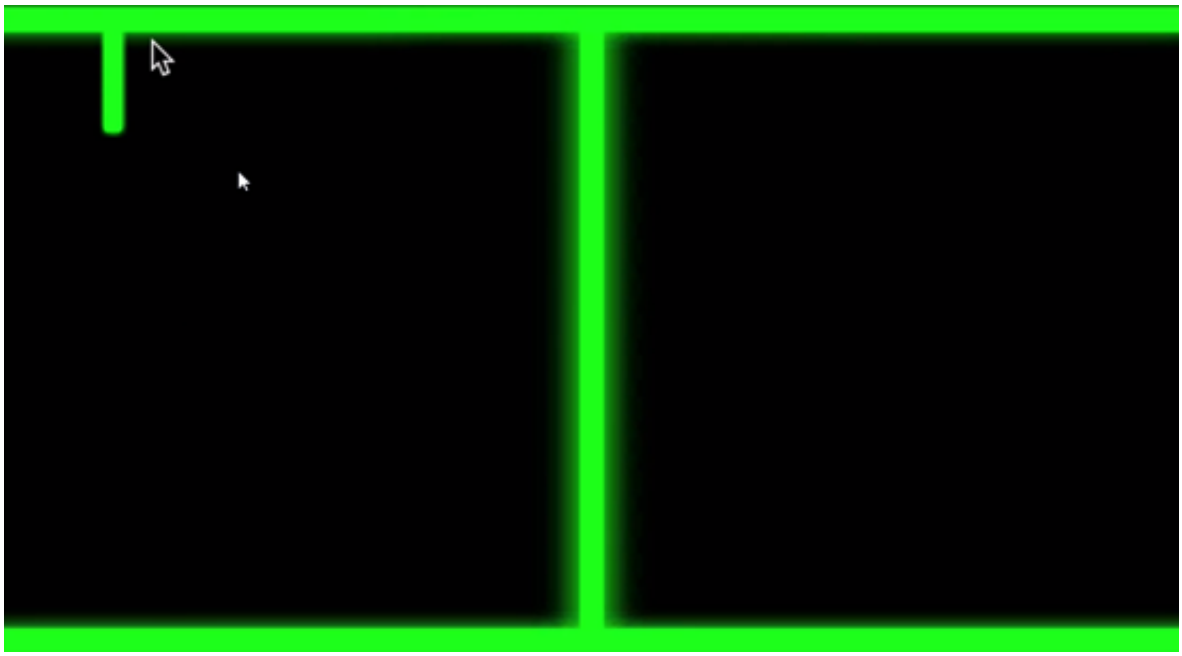
Make sure the background image is above the Paddle in the Hierarchy.

Rename the image to “Background” but you will notice that the background isn’t fitting in the screen entirely.

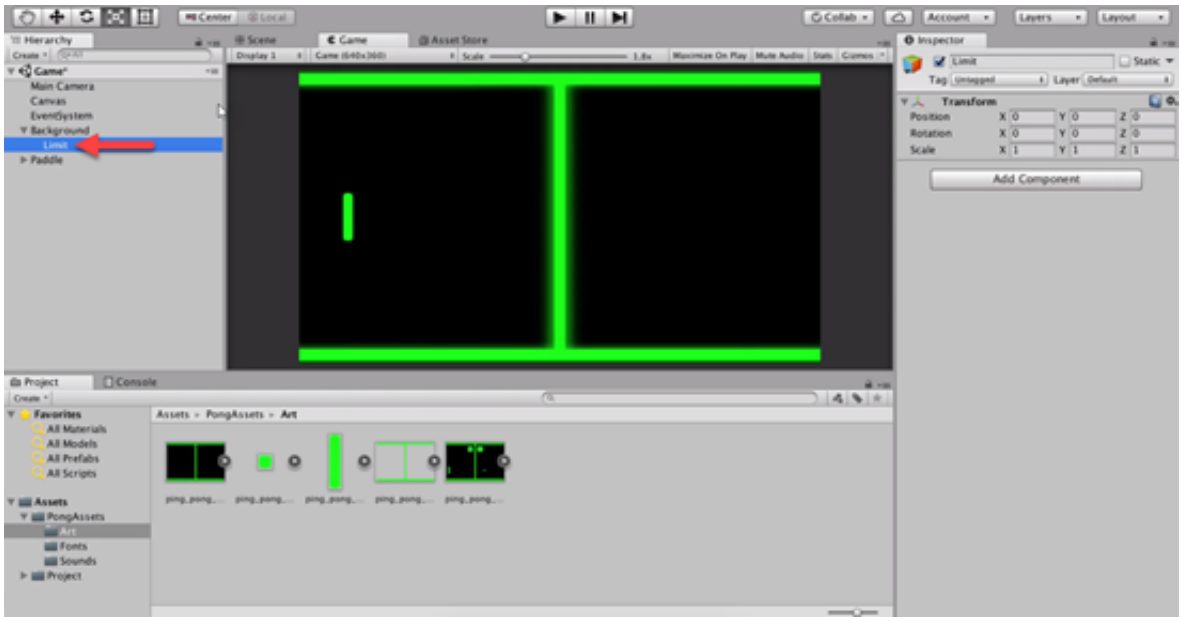
Remember, the vertical size of our screen is 360 but, are background is only 320.

So we want it to fit the screen perfectly. What we need to do is adjust the **scale** on the **Y axis** to divide 360 by 320 and you can do that mathematical operation right there in the value field and the number you will get is 1.125. 1.125 is the size we want, and you will see that the background now fits perfectly vertically. We can increase the **scale** on the X value to 2.25.

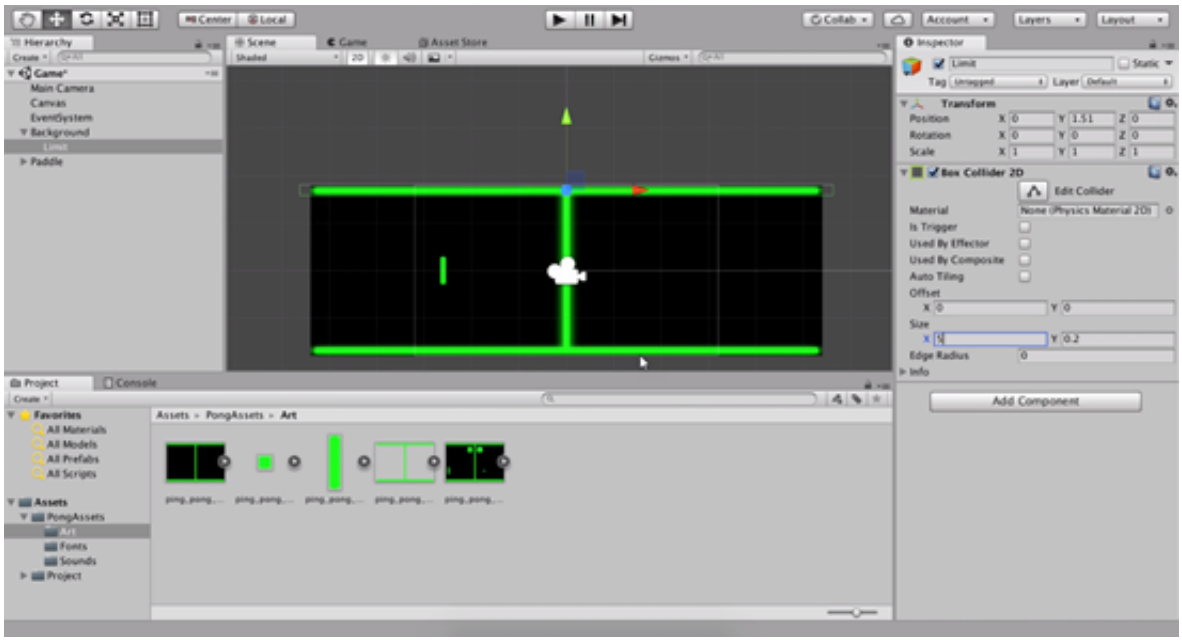
Now that we have the background in place perfectly we want the paddle to not be able to move over the green border like in the screenshot below:



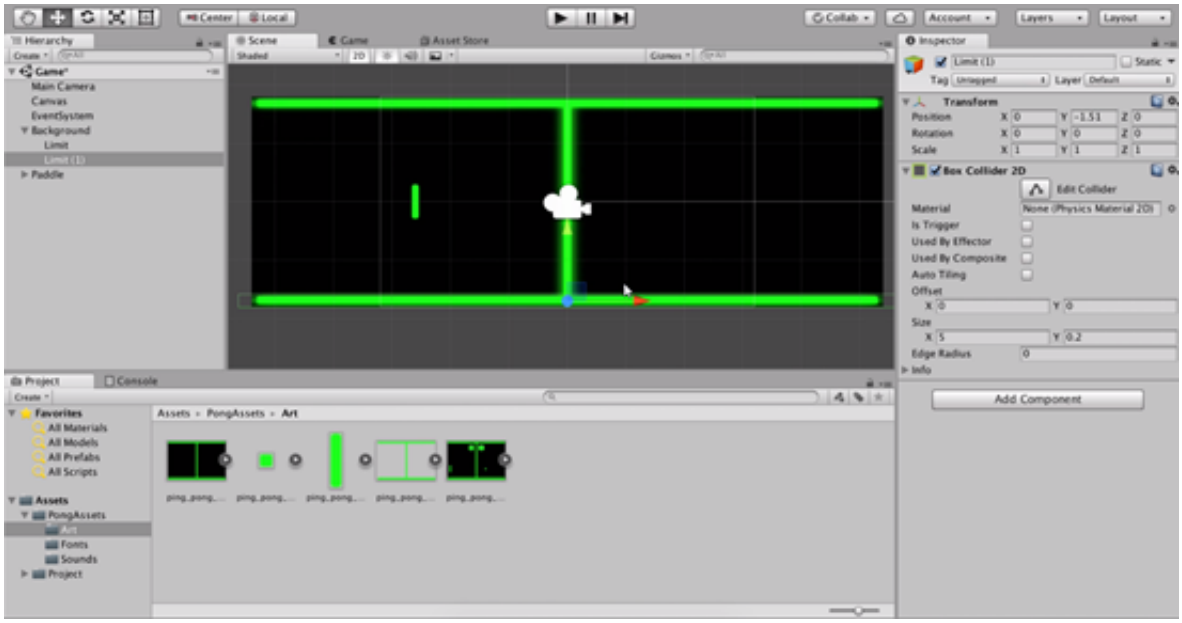
We can add some empty game objects inside the **Background**. **Right Click** and choose **Create Empty**. Rename it to “Limit.”



With the Limit object selected adjust the position on the Y value to 1.51 and add a 2D box collider to it as a component. Set the **Size** on the box collider to 5 on the X value and 0.2 on the Y value. This will make the box collider cover the whole top of the background.



Duplicate the Limit game object by hitting Ctrl/CMD +D. You will now have a Limit(1) object added to the Hierarchy. Put this is the position of -1.51 on the Y value. This will move it down and cover the whole bottom.

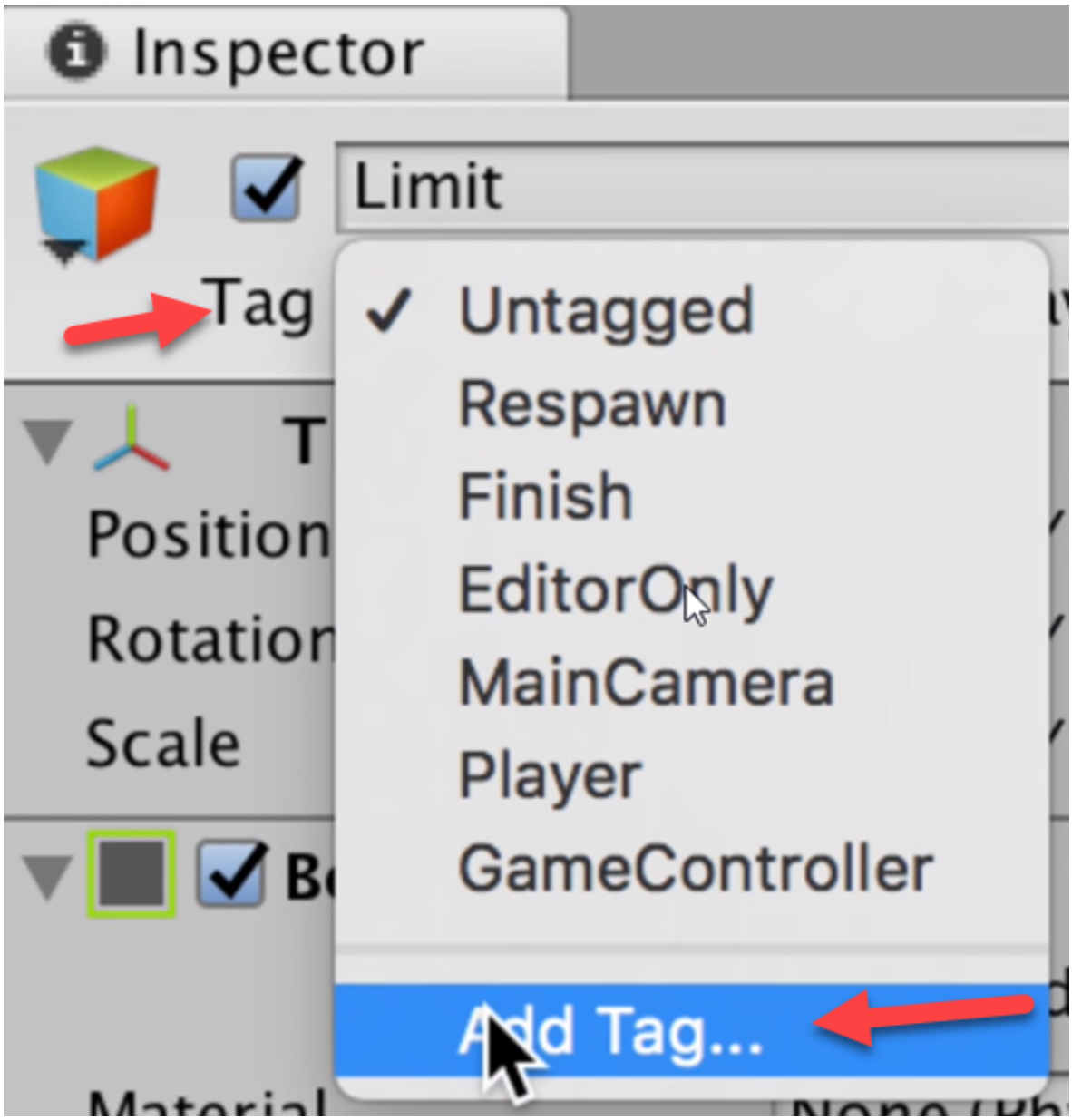


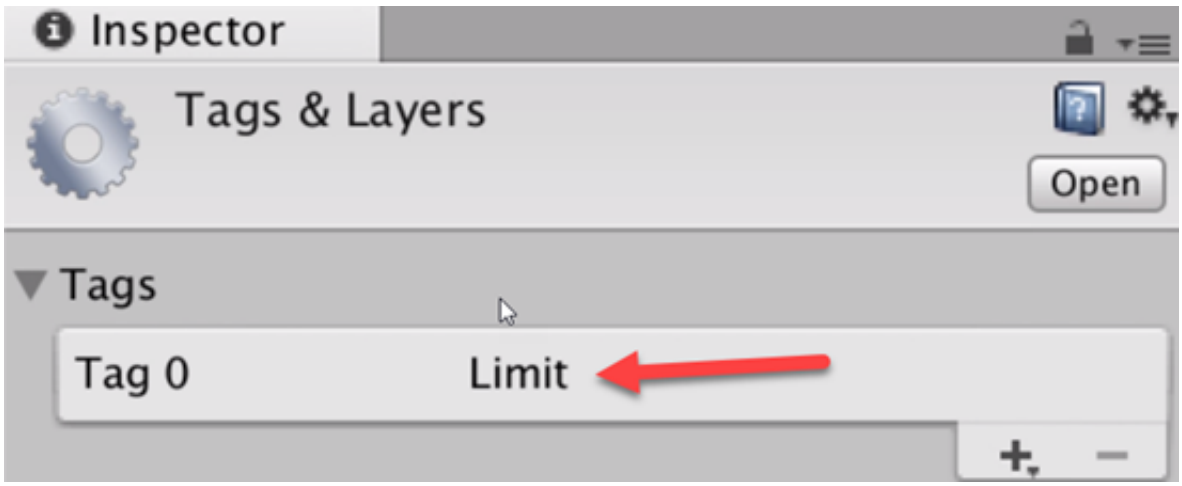
Also, whenever the Paddle **collides** with the Limit game objects we want to tell the paddle we actually **collided** with a Limit.

We can do this by adding a tag to the Limit game objects.

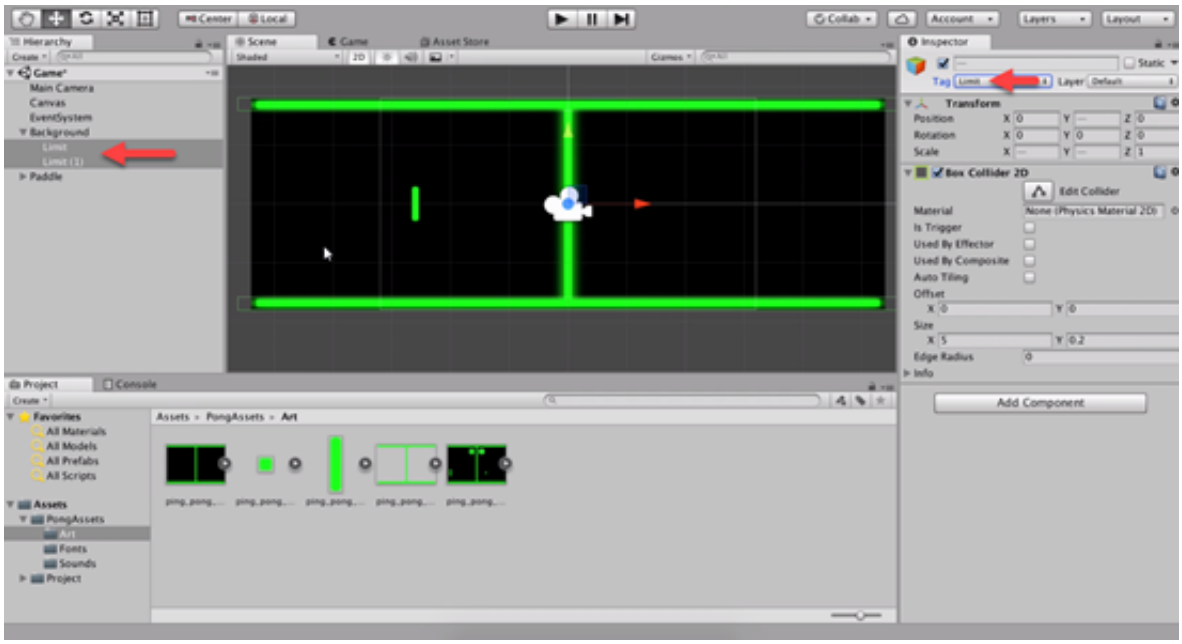
Choose the first Limit and click the **Tag** drop down menu and then choose **Add Tag**.

Left click the **plus sign** to add a new tag to the project and name the tag “Limit” and hit the **Save** button.





We can now use this tag to “tag” both the Limit game objects. Select both the objects and tag them “Limit.”

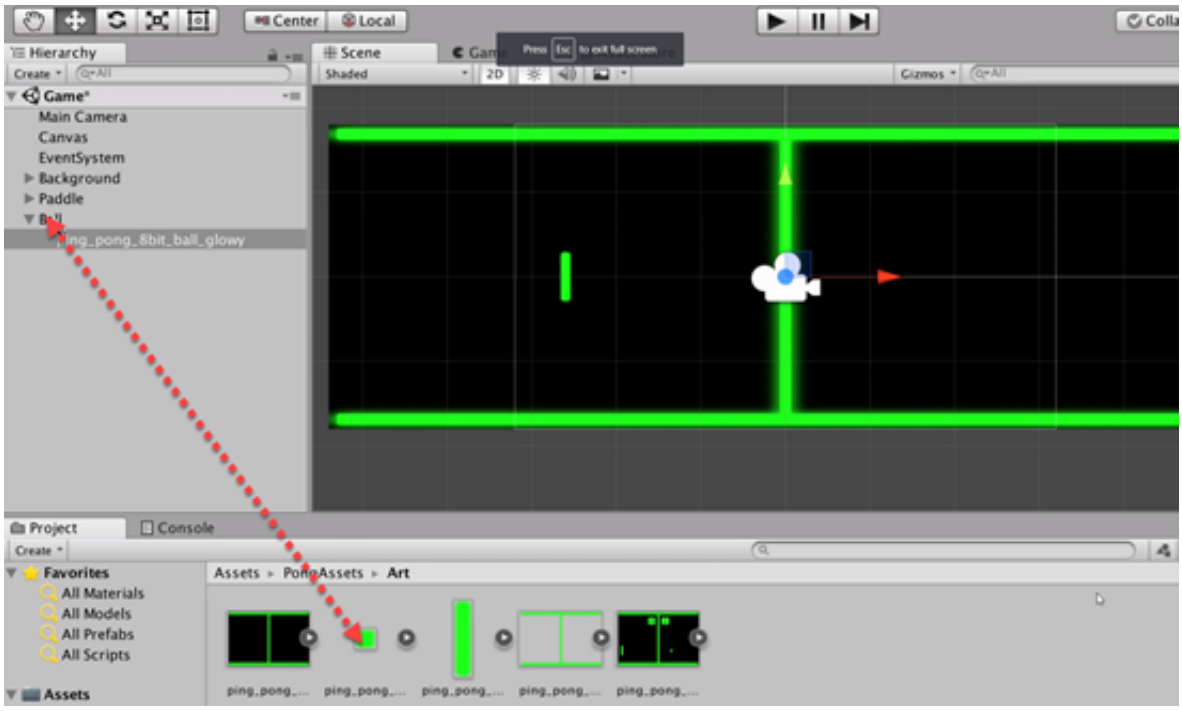


Now that we have these objects tagged we can use this to detect collision between the Paddles and the Limit game objects when the game is being played. Then we can code the logic to make this work plus make the ping pong ball bounce off the Limit game objects. We will writing the code for all this later on, in the next lesson we will be creating the ping pong ball.

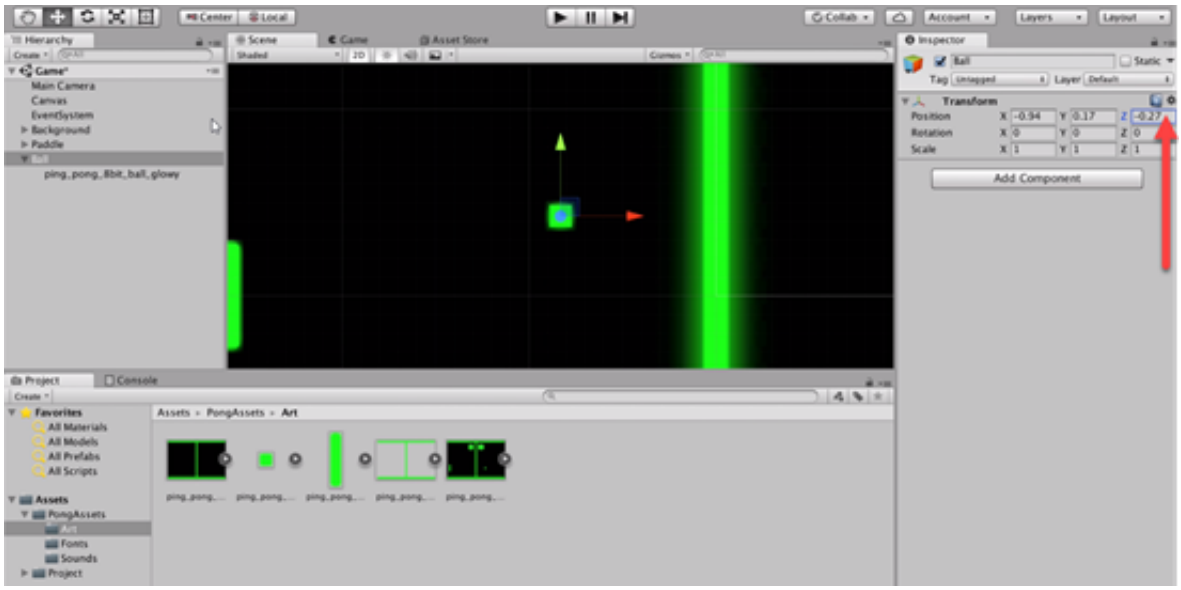
Save the scene and project, and proceed to the next lesson.

We will begin this lesson by creating the ball. Right click in the **Hierarchy** and choose create empty and rename this game object to “Ball.”

Now we need to drag and drop the ball art asset onto the **Ball** object, this will make the sprite a child to the Ball object.



You may notice that you cannot see the ball sprite yet, and you can adjust the Z value on the position to -0.27 on the transform component for the **Ball game object**, not the sprite itself, and once you have done this you should now see that ball sprite in the scene.

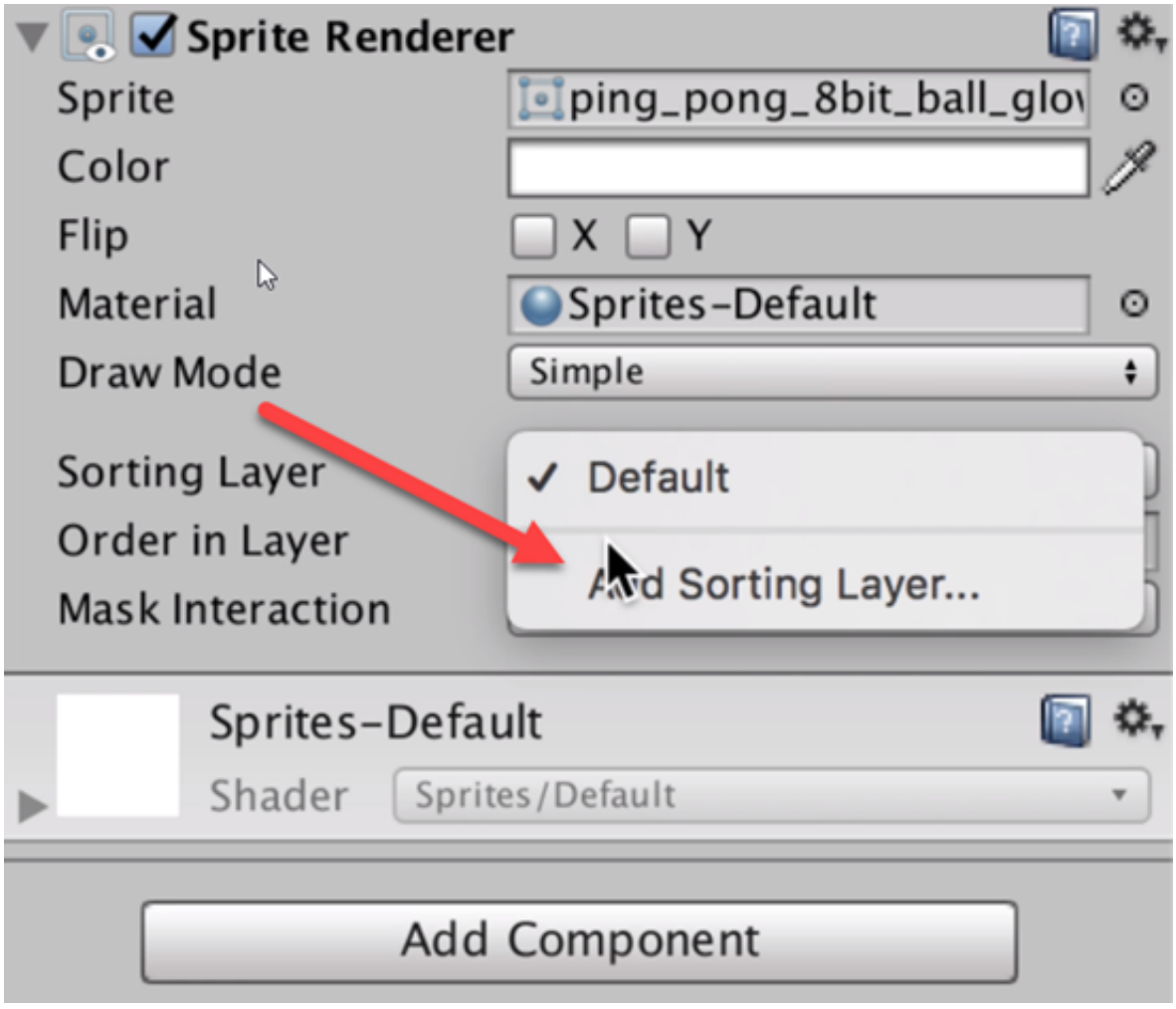


Now, you may be wondering why we couldn’t see the ball sprite initially and this is because the ball was actually behind the background. This is a good time to talk about **Sorting Layers**, and tell you what they are and how they function within Unity.

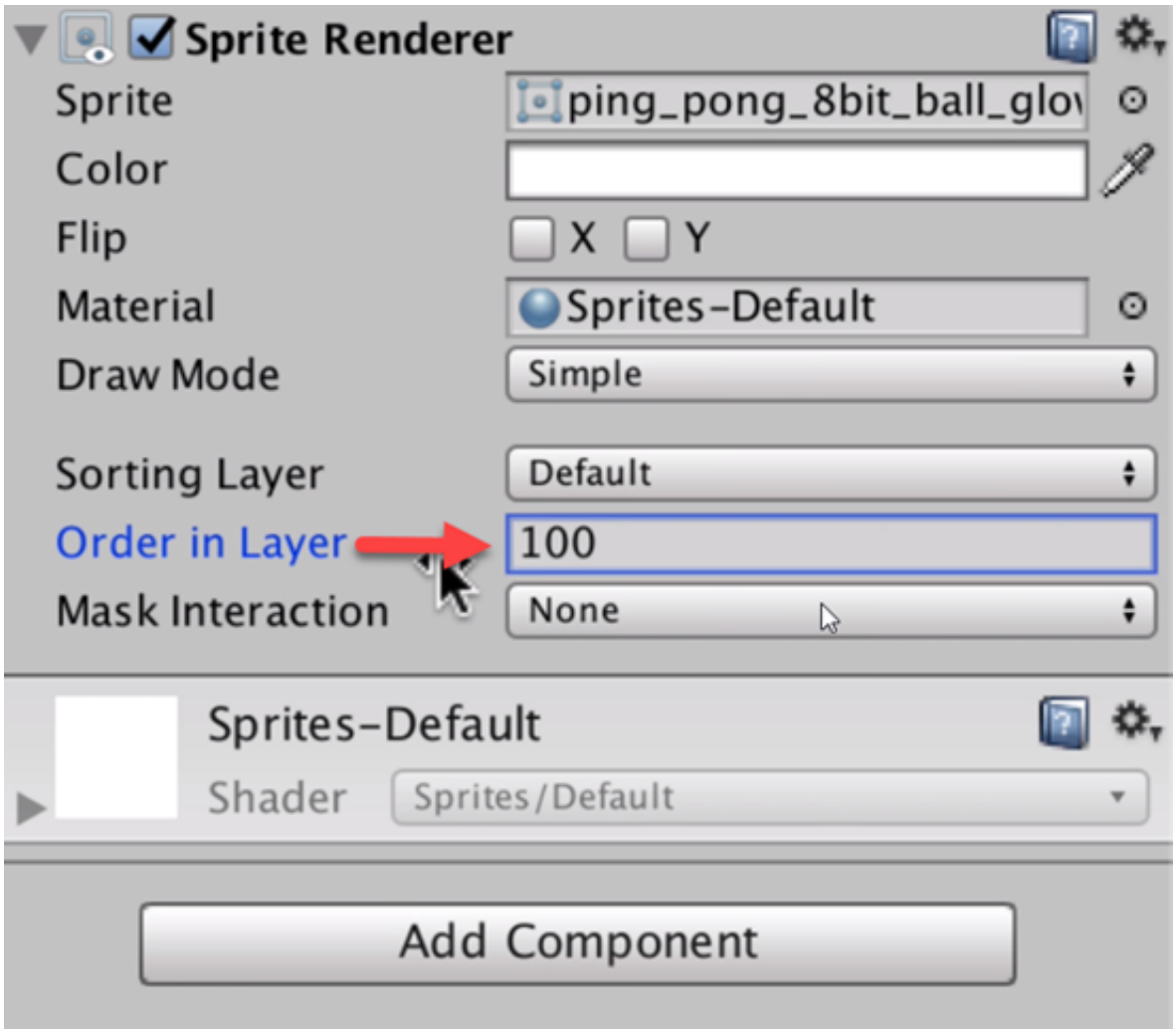
Sorting Layer's

It's important to understand what a Sorting Layer is and how they can be used in Unity. Sometimes, with two dimensional games you may have the rendering problem we had where the ball was actually behind the background, and the camera wasn't able to render the sprite. The Z value for the position on the transform component will effect the way the camera can render game objects in Unity. We moved the ball object in our scene to -0.27 on the Z axis. This made it so we were able to see it, but the best thing to do here is to make sure every value on the Z position is set to 0. Then for rendering purposes we will worry about the **Sorting Layer**. The Sorting Layer is used for sprites in two dimensional games.

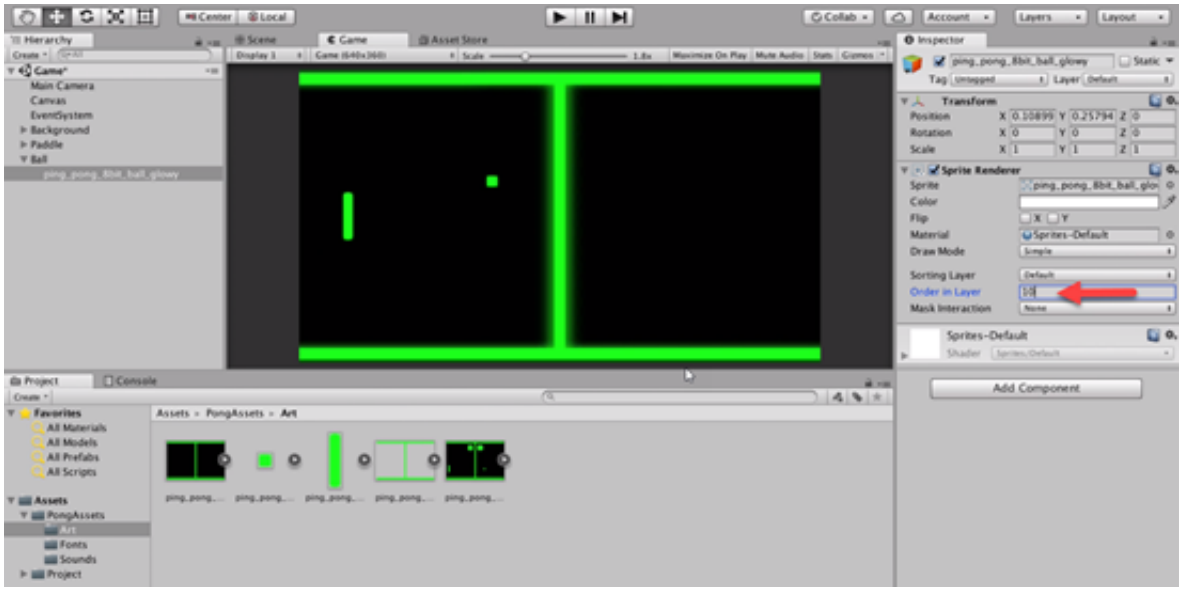
Whenever you have two layers, the one that is closest to the bottom of the list on the drop down menu is going to be rendered first by the camera.



Then inside the different **Layers,** the ones with the higher order are going to be rendered first.



So, what we are going to do here is select the ping pong ball sprite that is a child of the ball object in the scene and put it on the order 10 in the **Order in Layer** value on the **Sprite Renderer** component.

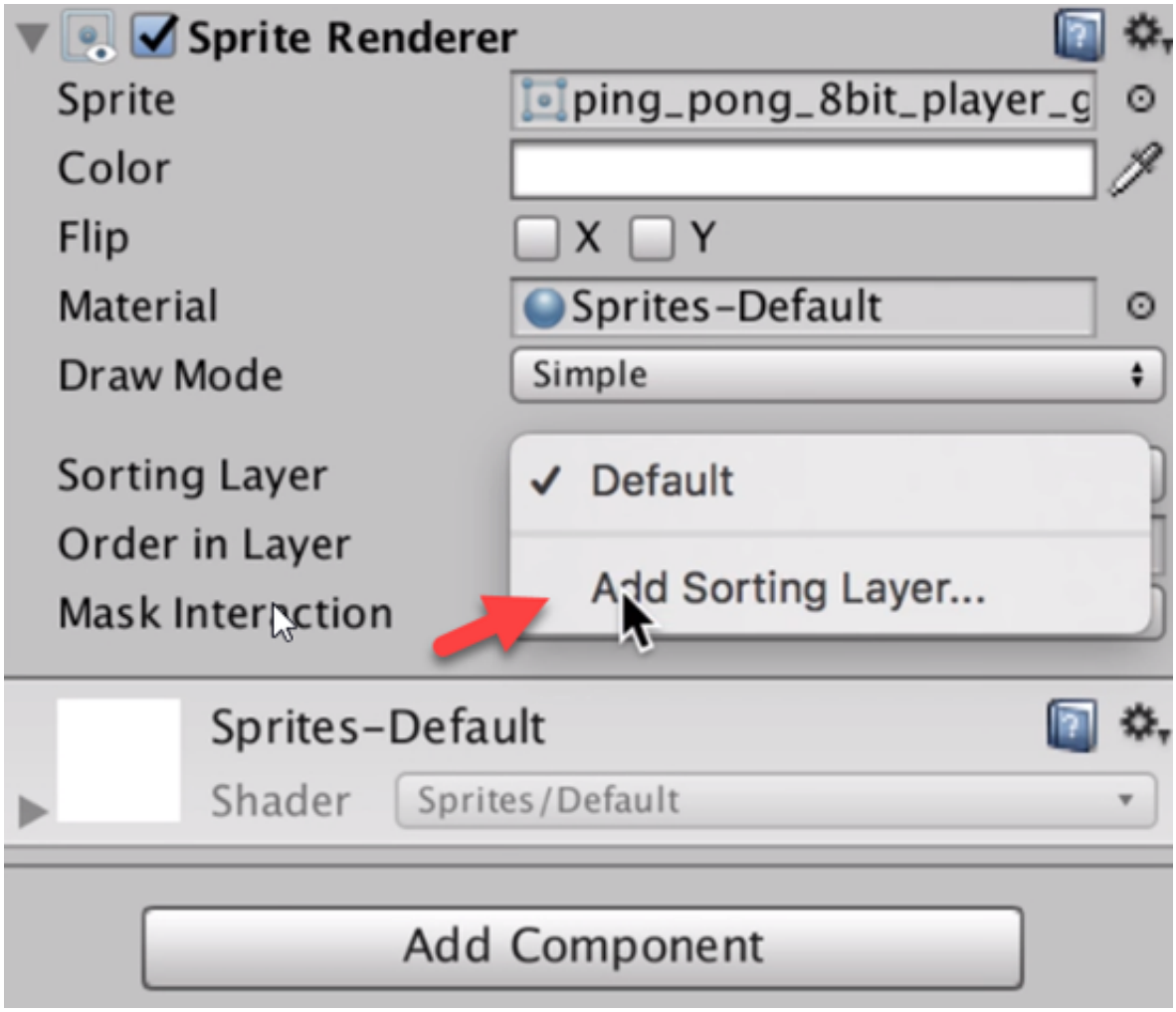


Now, we are able to see the ball by doing this.

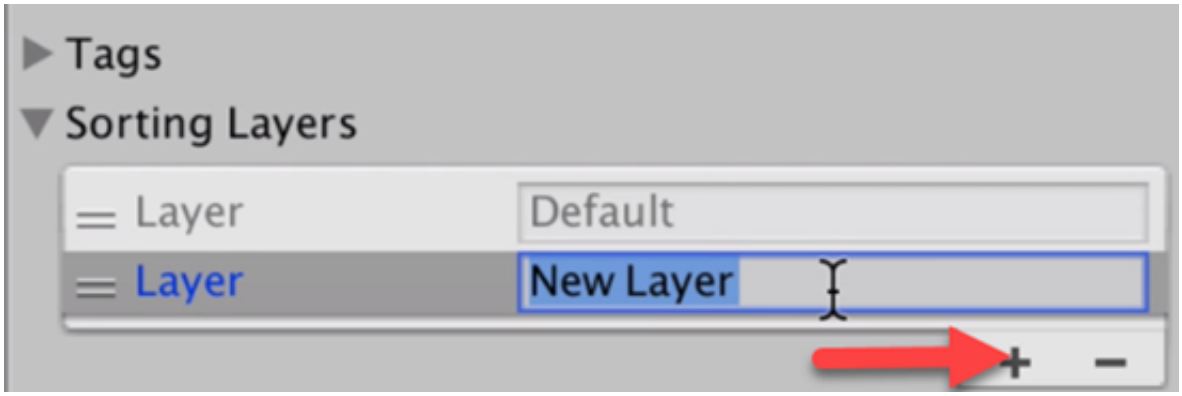
Then, select the paddle sprite and change its Order in Layer value to 5. This will make sure there is no conflict with the Background.

Make sure the Background sprite's Order in Layer value is set to 0.

The next thing to do is to add a sorting layer, you can do this by selecting the drop down menu and selecting the **Add Sorting Layer** option.



Then click the plus sign on the next menu to add a new Sorting Layer.



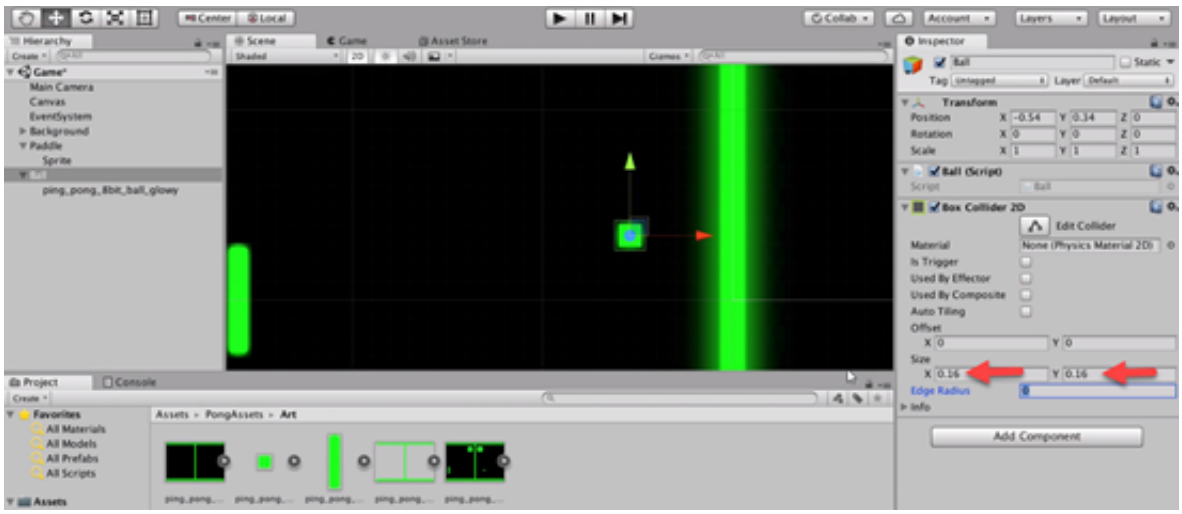
Then name the new layer that was created to "Game."

Then select the paddle sprite and change the sorting layer on the Sprite Renderer component to the new layer we just created **Game**. Set the Order in Layer to 0.

For the ping pong ball sprite change the layer to **Game**, and the order in layer to 0 as well.

Make sure that the ping pong ball sprite is in the position of 0,0,0 on the transform component. This is so that the sprite is exactly where the ball game object is.

The size of the ping pong ball sprite was 16X16, so we need to make sure the box collider is the same size. Set the X value to 0.16 and the Y value to 0.16.



We also need to add another component which is the Rigibody 2D, you can add this component now.

We should no have everything we need to work on the ball bouncing logic between the paddles.

Save the scene and project and we will proceed to the next lesson where we will begin coding the ball logic.

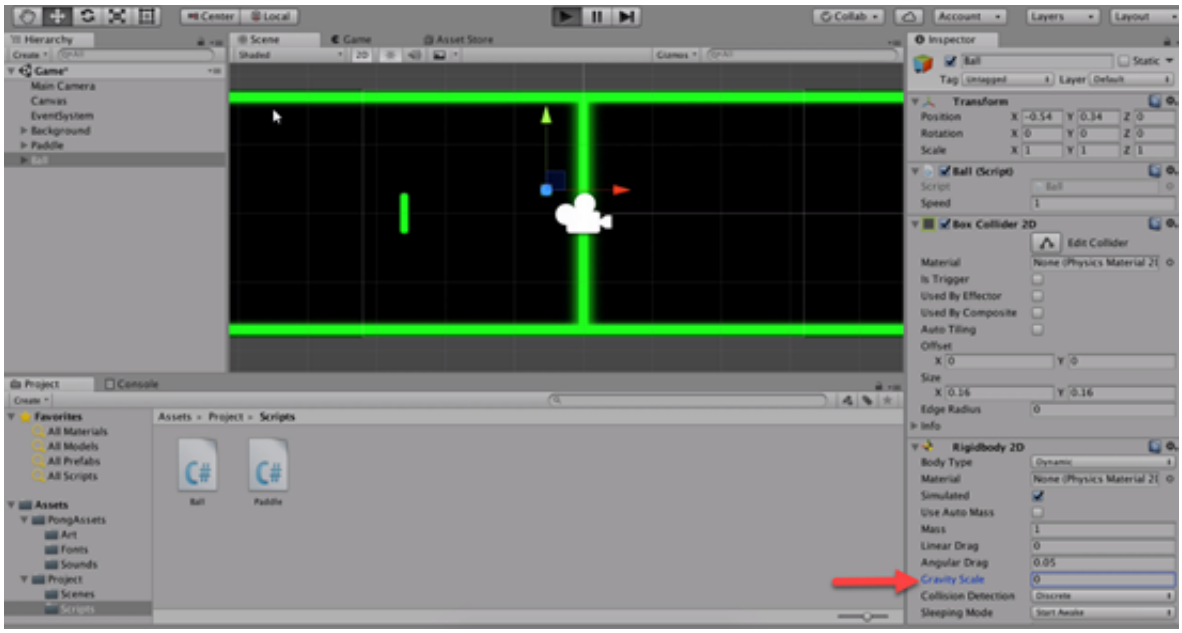
The Ball Logic

Open the Ball script up in the code editor.

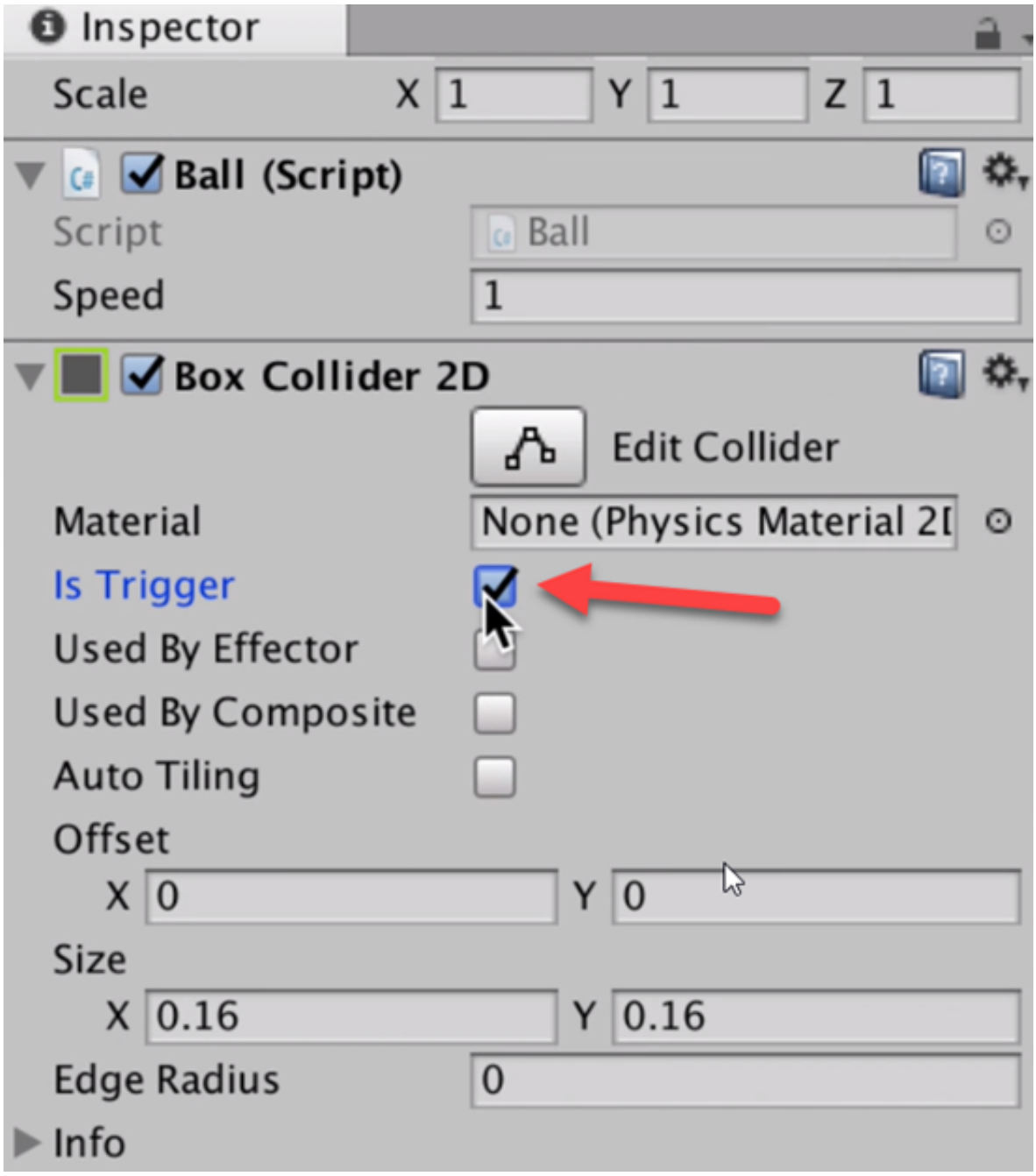
Just like we did for the paddle, we need to add a speed.

```
public float speed = 1f;


void Start(){
    GetComponent<Rigidbody2D>().velocity = new Vector2(0,speed,);
```

We now need to adjust the Gravity Scale on the Ball game object to 0 on the Rigidbody 2D component.



We want to set the Is Trigger option on the Box Collider 2D to true.



This is so we can check for overlapping collisions when the ball collides with something in the scene.

Open up the ball script again in the code editor.

```
public float speed = 1f;

private Rigidbody2D ballRigidbody;

void Start(){
    ballRigidbody = GetComponent<Rigidbody2D>();
    ballRigidbody.velocity = new Vector2(0.2f,speed);
}

void OnTriggerEnter2D (Collider2D otherCollider) {
```



```
  if (otherCollider.tag == "Limit") {

   // Collided with the top limit.
   if (otherCollider.transform.position.y > transform.position.y && ballRigidbody.vel
ocity.y > 0) {
    ballRigidbody.velocity = new Vector2 (
     ballRigidbody.velocity.x,
     -ballRigidbody.velocity.y
    );
   }

   // Collided with the bottom limit.
   if (otherCollider.transform.position.y < transform.position.y && ballRigidbody.vel
ocity.y < 0) {
    ballRigidbody.velocity = new Vector2 (
     ballRigidbody.velocity.x,
     -ballRigidbody.velocity.y
    );
   }
```

Save the script and go back into the Unity Editor to test the changes.

Hit the Play button and you will see that when the ball hits the top limit it bounces down and when it hits the bottom limit it bounces back upwards.

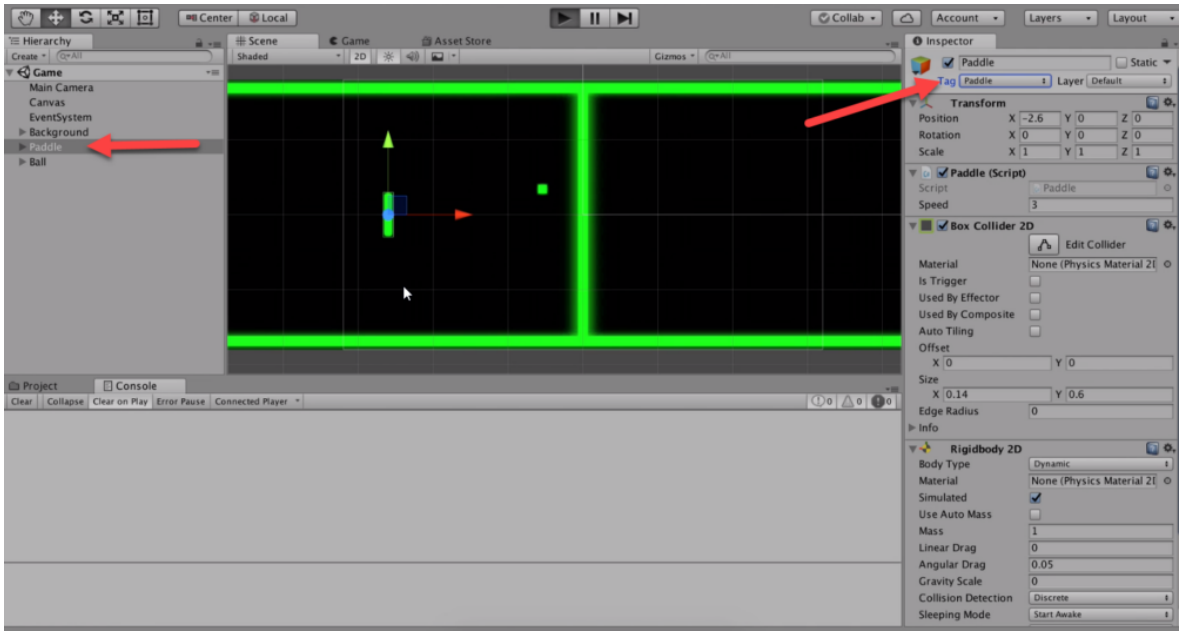
We now need to set up the collision with the paddle and the ball through code.

```
else if (otherCollider.tag == "Paddle") {

   // Collided with the left paddle.
   if (otherCollider.transform.position.x < transform.position.x && ballRigidbody.vel
ocity.x < 0) {
    ballRigidbody.velocity = new Vector2 (
     -ballRigidbody.velocity.x,
     ballRigidbody.velocity.y
    );
   }

   // Collided with the right paddle.
   if (otherCollider.transform.position.x > transform.position.x && ballRigidbody.vel
ocity.x > 0) {
    ballRigidbody.velocity = new Vector2 (
     -ballRigidbody.velocity.x,
     ballRigidbody.velocity.y
    );
   }
  }
 }
}
```

Save the script and go back into the Unity Editor and we need to make a tag for the Paddle.

Select the Paddle game object and click on the drop down menu for tag and select Add Tag.

Select the plus sign and name the tag “Paddle.” Once this is created make sure the Paddle game object is tagged with this now.



Save the scene and project.

Hit the Play button and test out the changes. You will see that if the ball collides with the paddle it bounces back to the right side.

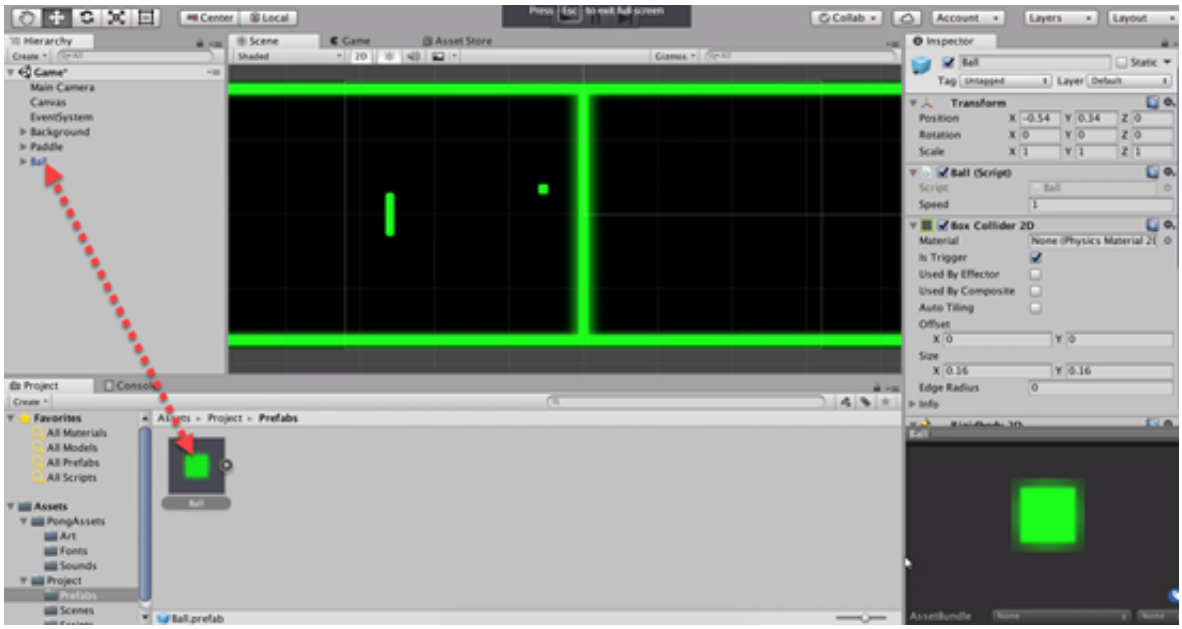
In the next lesson we will be adding a random velocity to the ball.

To dynamically add the ball in this game we need to transform it into something that can be copied and added to the Hierarchy. We need to create a prefab, and make the ball a prefab to do this.

Creating the Ball Prefab

We need to create a new folder and add it to the Project folder. Name this new folder “Prefabs.”

Enter the folder by double clicking on the Prefabs folder, and drag the Ball game object and drop it inside the Prefabs folder.



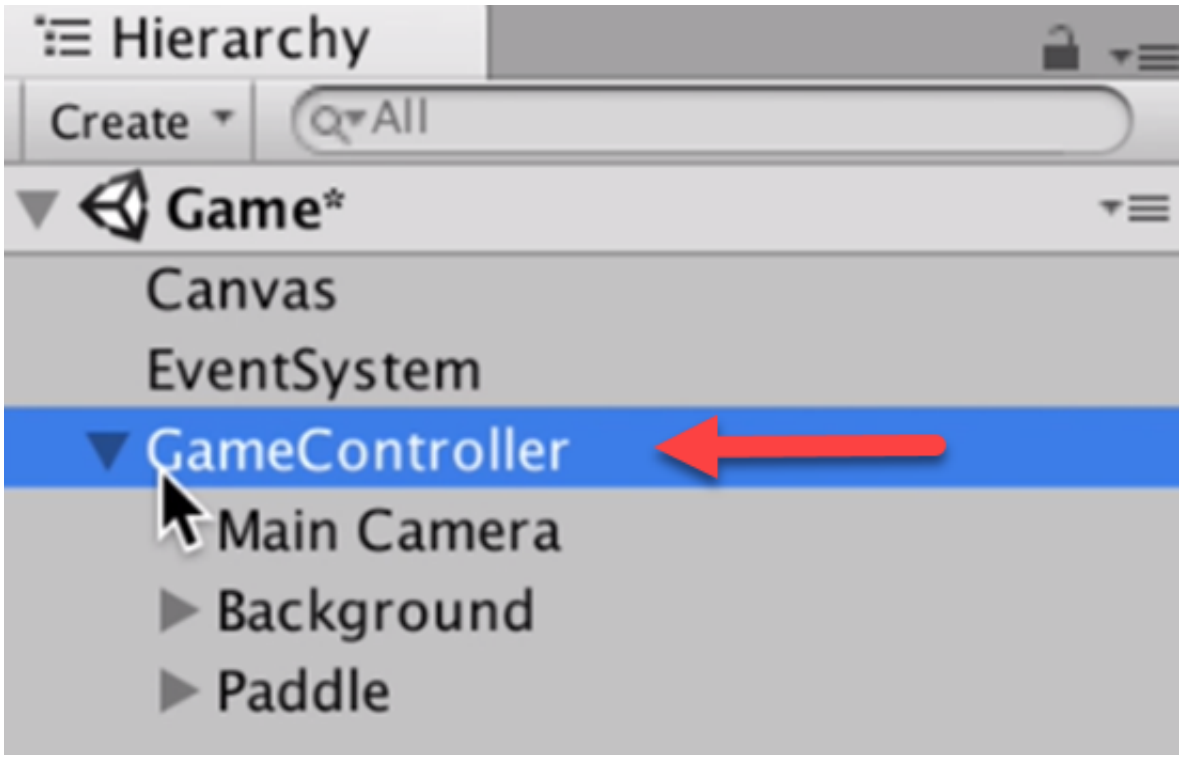
Now that the ball is a prefab we can select the ball in the scene and delete it.

We will be adding the ball back in shortly, but first we must create a new game object that will add the ball into the game and control the rules of the game.

Game Controller Setup and Creation

Create a new empty game object and rename it “GameController.” Reset the transform component so that the position of the object is at 0,0,0.

Now we want to make the camera, background, and paddle children to the GameController.



And move the GameController to the top of the Hierarchy.

We need to make a script for this game object, so create a new C# script and name it "GameController." Create the script in Scripts folder and then attach it to the GameController object by dragging and dropping it on the object.

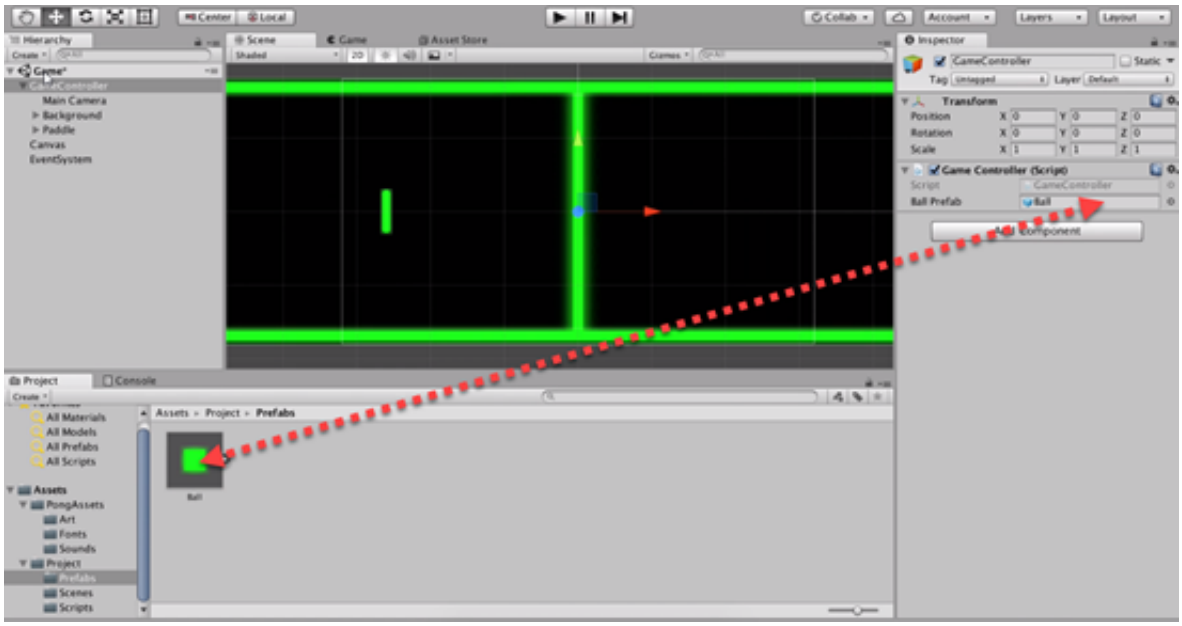
Coding the Game Controller

Open the GameController script up in the code editor and we need to make a simple instantiation of the ball prefab.

```
public GameObject ballPrefab;
```

Save the script and go back into the Unity Editor. If you select the GameController object in the scene and look at the Inspector window you will see a spot for the ballPrefab to be added.

You can drag and drop the ball prefab onto this spot in the Inspector.



Save the scene and project, and go back to the script. Notice that, despite our game being 2D, positions in Unity always have three dimensions, so we need to set it to Vector3.zero. However, this should have no impact on your game.

```
private Ball currentBall;


// Use this for initialization
 void Start () {
  GameObject ballInstance = Instantiate(ballPrefab, transform);

                currentBall = ballInstance.GetComponent<Ball>();
                currentBall.transform.position = Vector3.zero;

 }
```

Save the script and go back into the Unity Editor, and hit the Play button, you will see that a ball appears in the middle and starts moving itself.

Even though we now have a dynamic ball being added here in the game, the ball is always moving at the same angle. What we want to do here is make the ball move to a random place.

We also need to decide what the minimum and maximum speeds that are going to be applied here, this is going to make the game a little bit more dynamic.

We will be working on this type of logic in the next lesson, go ahead and save the scene and project and proceed to the next lesson.

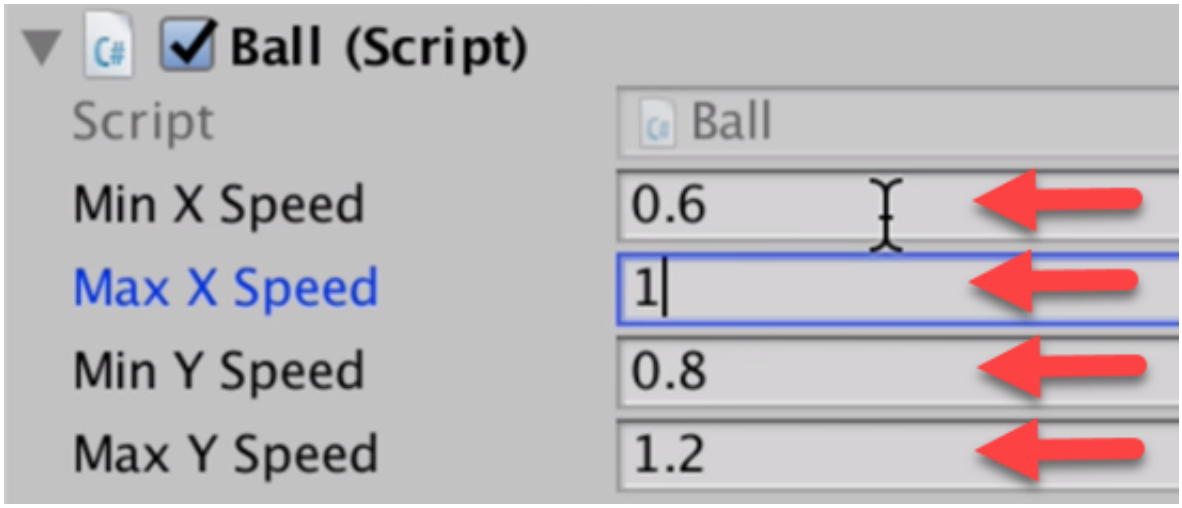
Improving the Speed for the Ball

Open the Ball script up in the code editor, and lets improve the speed for the ball.

```
public float minXSpeed = 0.8f;
public float maxXSpeed = 1.2f;

public float minYSpeed = 0.8f;
public float maxYSpeed = 1.2f;

private Rigidbody2D ballRigidbody;

// Use this for initialization
 void Start () {
  ballRigidbody = GetComponent<Rigidbody2D> ();
  ballRigidbody.velocity = new Vector2 (
   Random.Range(minXSpeed, maxXSpeed) * (Random.value > 0.5f ? -1 : 1),
   Random.Range(minYSpeed, maxYSpeed) * (Random.value > 0.5f ? -1 : 1)
  );
 }
```

Save the script and go back into the Unity Editor and hit the Play button and look where the ball goes then stop Play mode and hit Play again and you will notice the ball is randomly changing each time the Play button is hit.

Since we made the speed variables public you can always adjust them directly in the Inspector window on the Ball script component.



Next, we need to know if we scored a point or not, and we will work on coding that logic in the next lesson.

Save the scene and project and proceed to the next lesson.

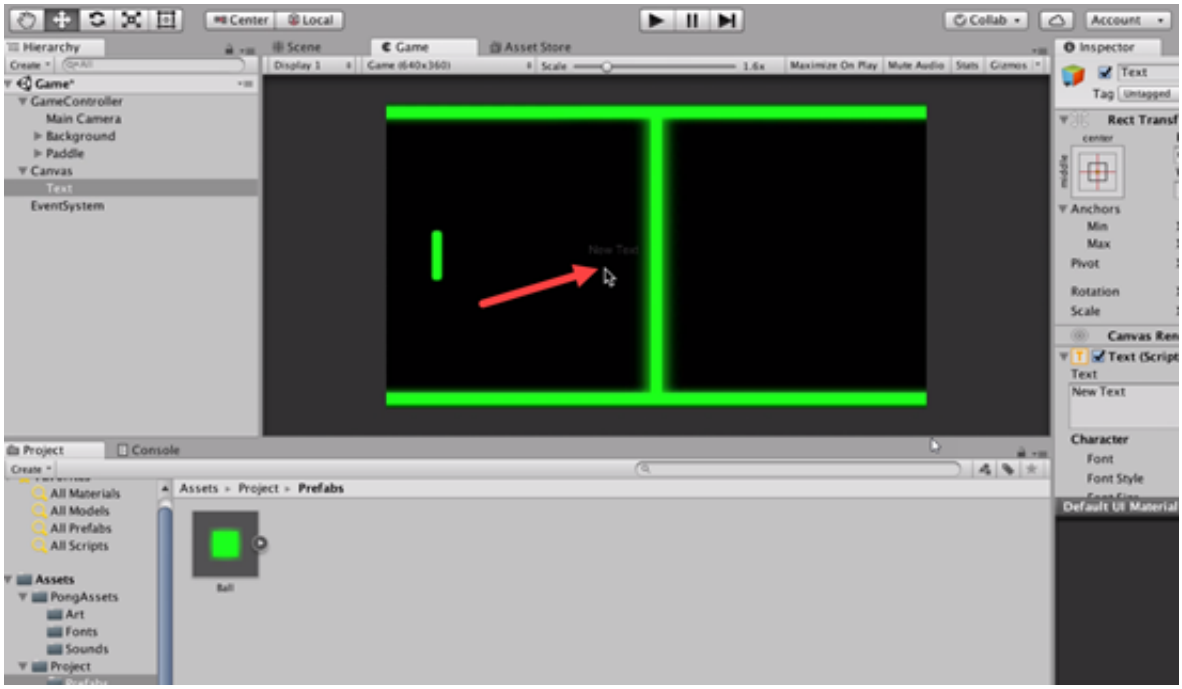
The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected instruction.

For the scoring logic we are going to need a simple user interface.

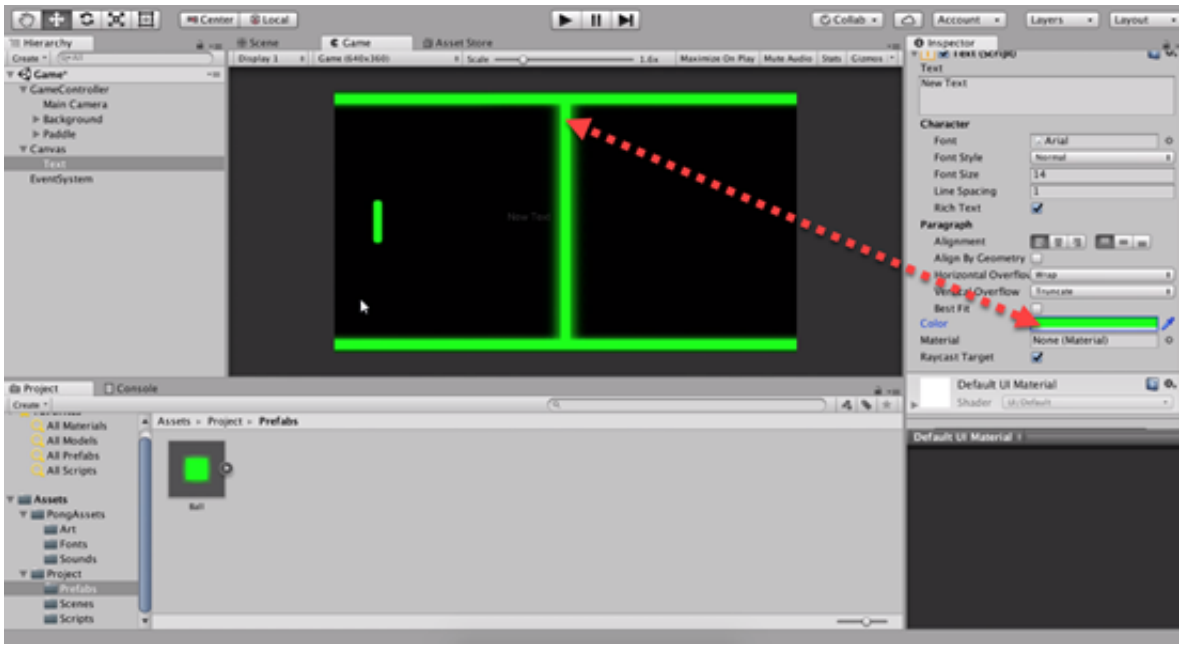
Scoring U.I. Creation

The following instructions have been updated, and differ from the video: Please note that this course uses the legacy Text UI component. Newer versions of Unity have prioritized the use of the **Text – TextMeshPro** component. You can find the legacy Text UI component at **Hierarchy > Right-click > UI > Legacy > Text**.

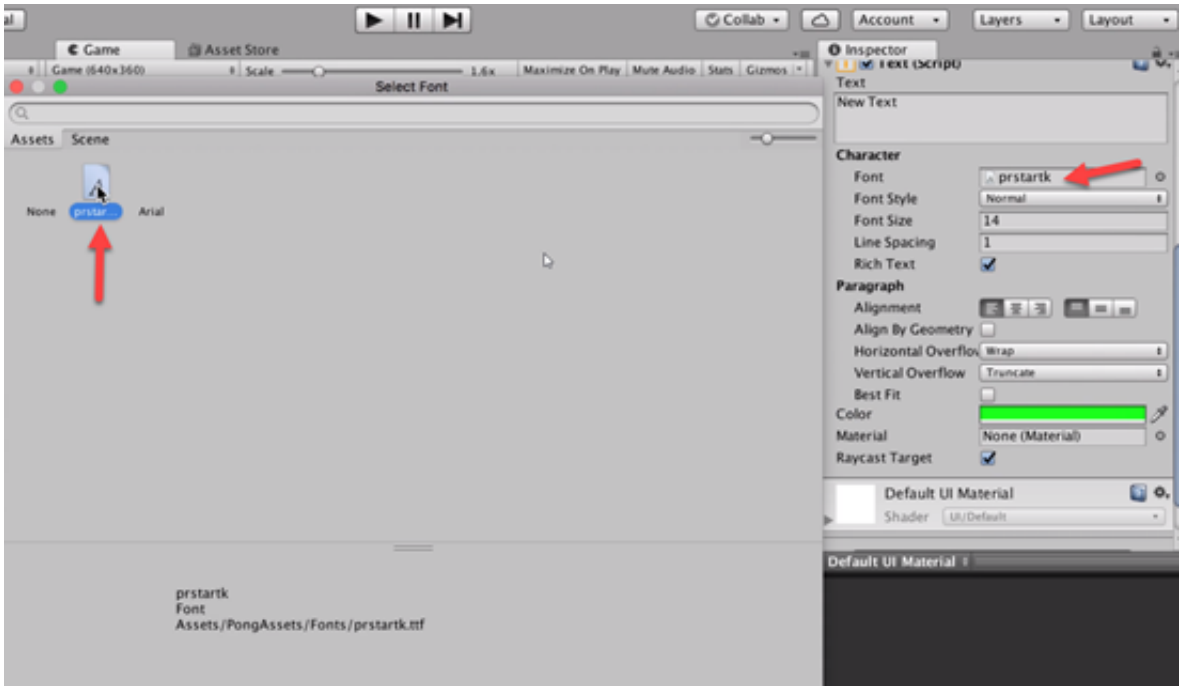
Select the Canvas game object in the Hierarchy>Right Click>UI>Text. This will add a text object to the scene and you will see it in the scene.



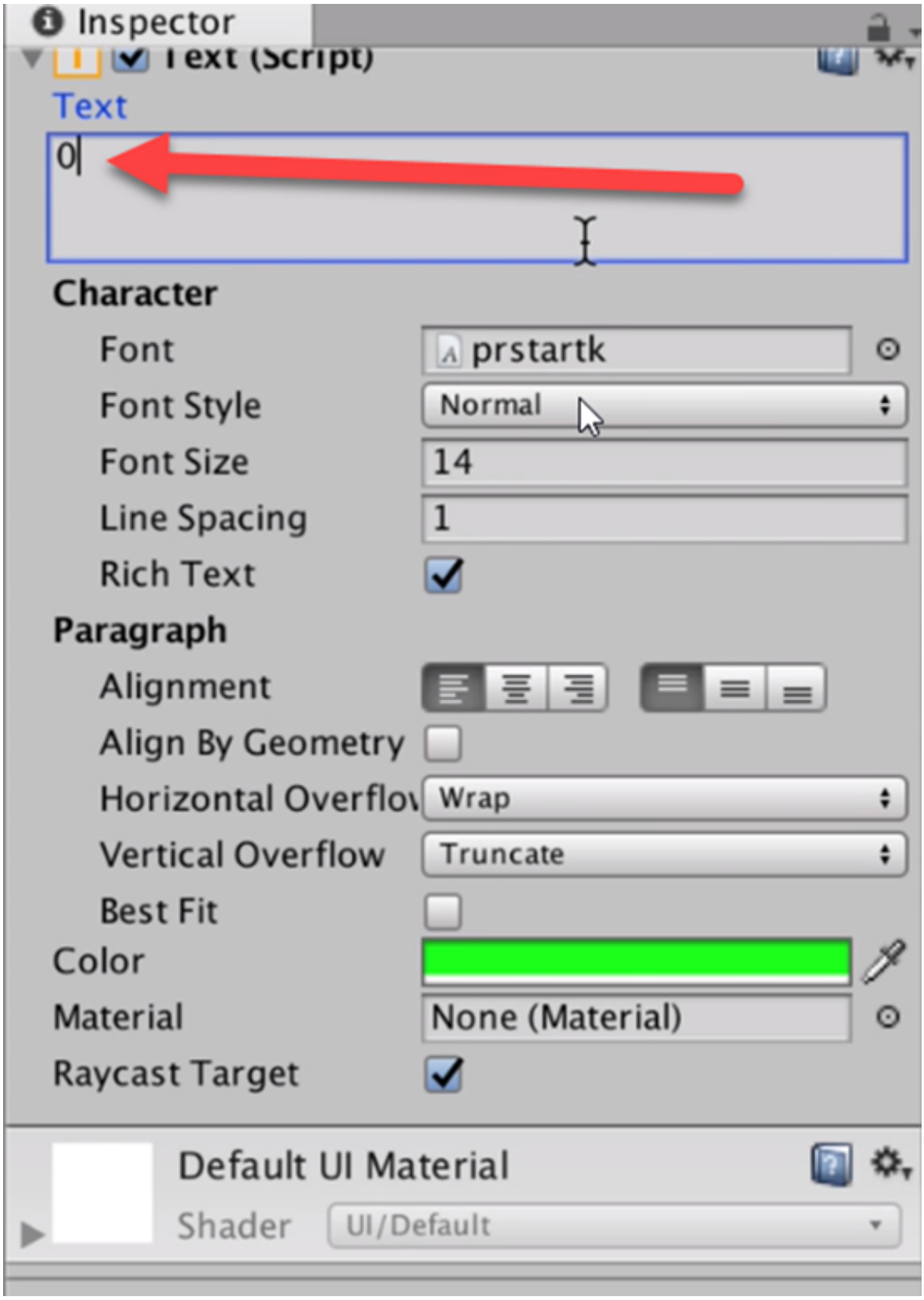
Use the color picker in the Inspector to choose a tone of green.



change the font to the **prstartk** font that came with the course assets.

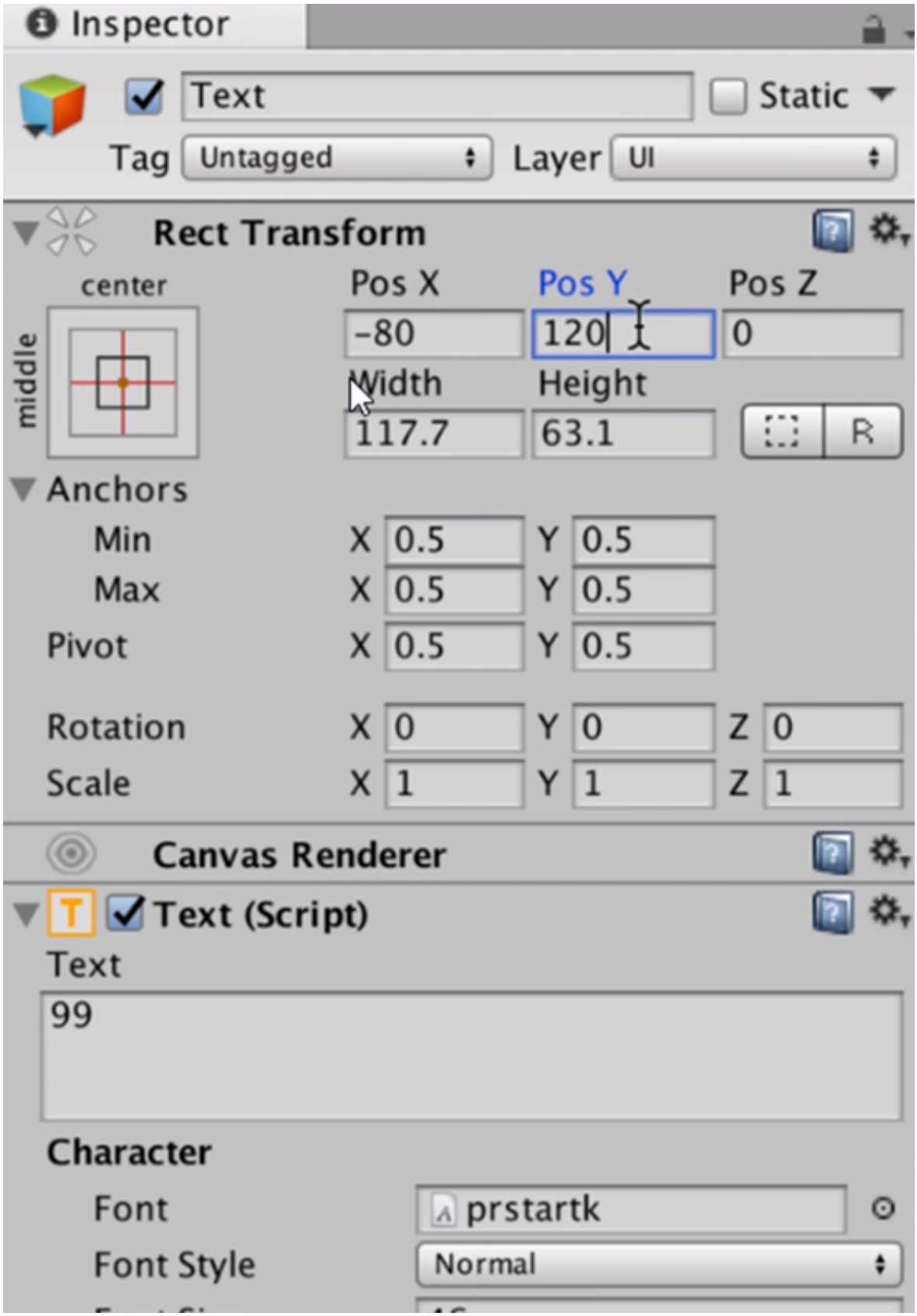


Change the text to 0 in the text field.



Go to the Scene window and focus on the Text object by selecting it and hitting “F.”

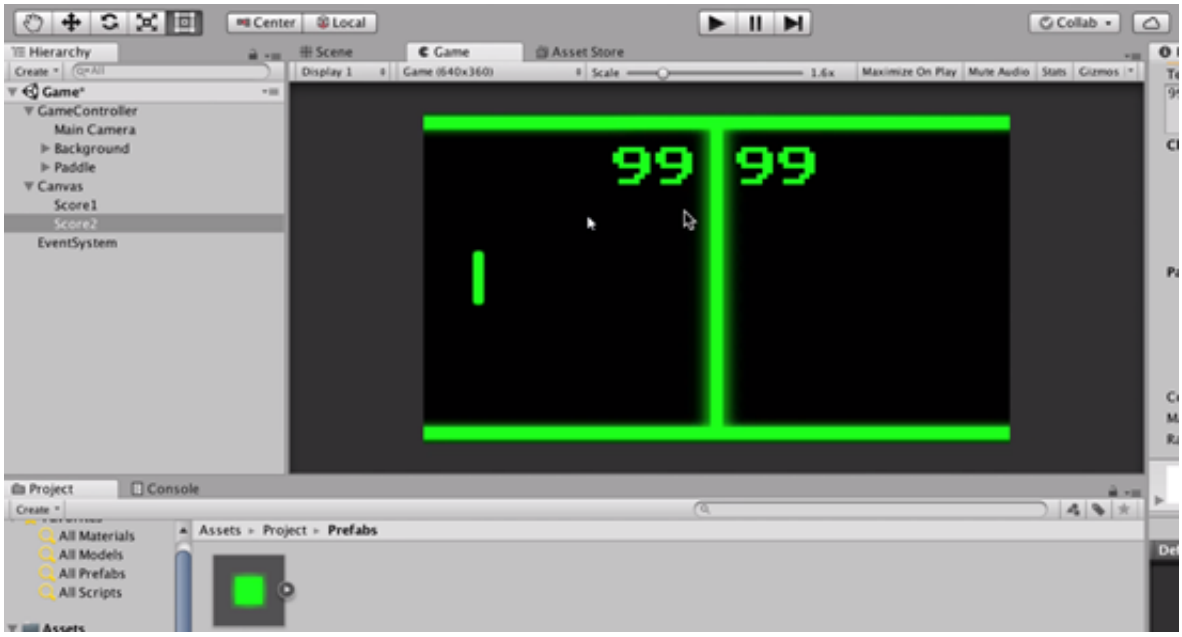
We want to increase its size to Pos X = -80, Pos Y = 110, and Pos Z = 0, and the font size to 46. The width can be set to 117.7 and the height to 63.1.



Rename the Text game object to “Score1.”

Duplicate the Score1 game object, then rename the duplicated object to “Score2.”

Then move the Score2 game object to Pos X = 80, Pos Y = 110, Pos Z = 0. Then align the text to the left.



We now have these two text fields to work with.

Open up the GameController script in the code editor.

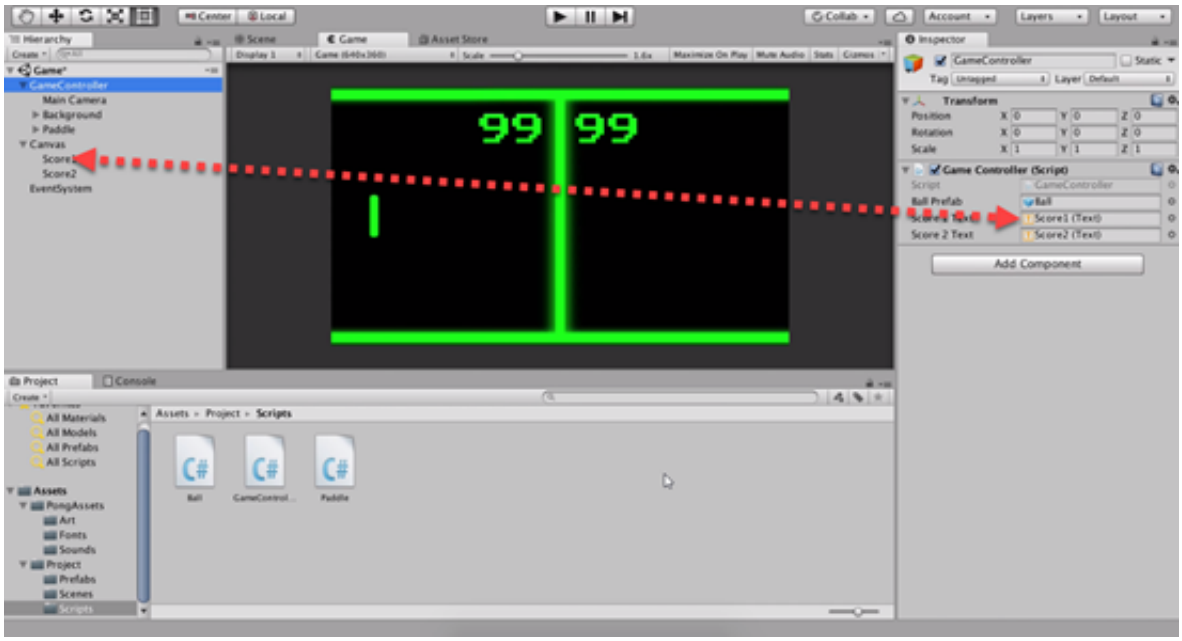
We need to get references to the text objects we just created in this script.

We need to first import a namespace in order to access the user interface elements with Unity within scripts properly.

```
using UnityEngine.UI;



public Text score1Text;
public Text score2Text;

private int score1 = 0;
private int score2 = 0;

// Use this for initialization
 void Start () {
  score1Text.text = score1.ToString ();
                score2Text.text = score2.ToString ();
 }
```

Save the script.

We now need to hook up these text game object to the script component on the Game Controller object.



Scoring Logic

Open the Game Controller script in the code editor and we need to add some coordinates to the script in order to update the score if the ball passes these coordinates during play.

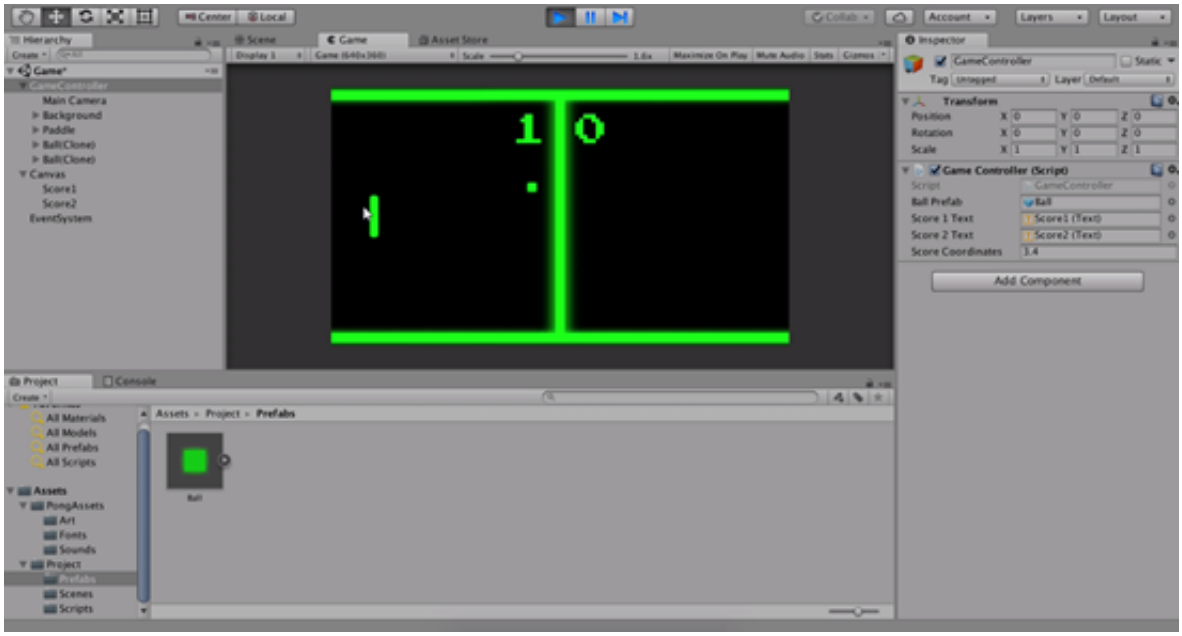
```
public float scoreCoordinates = 3.4f;


// Use this for initialization
 void Start () {
  SpawnBall ();
 }

 void SpawnBall () {
  GameObject ballInstance = Instantiate (ballPrefab, transform);

  currentBall = ballInstance.GetComponent<Ball> ();
  currentBall.transform.position = Vector3.zero;

  score1Text.text = score1.ToString ();
  score2Text.text = score2.ToString ();
 }

// Update is called once per frame
 void Update () {
  if (currentBall != null) {
   if (currentBall.transform.position.x > scoreCoordinates) {
    score1++;
    SpawnBall ();
   }

   if (currentBall.transform.position.x < -scoreCoordinates) {
    score2++;
    SpawnBall ();
   }
```



```
  }
 }
```

Save the script and lets give these changes a test in the Unity Editor.

You will notice that when the ball leaves to the right side the Score1 text object was updated to “1.”



The score logic is now working properly for us.

We just need to remove the balls that keep accumulating in the scene every time the ball is spawned in. We don’t want these balls adding up too many times and this will slow don’t the games performance. We can add a destroy command to the GameController script to take care of this. Open the script up in the code editor.

```
// Update is called once per frame
 void Update () {
  if (currentBall != null) {
   if (currentBall.transform.position.x > scoreCoordinates) {
    score1++;
    Destroy (currentBall.gameObject);
    SpawnBall ();
   }

   if (currentBall.transform.position.x < -scoreCoordinates) {
    score2++;
    Destroy (currentBall.gameObject);
    SpawnBall ();
   }
  }
 }
```


Save the script and test the changes in the Unity Editor.

You will see that now the ball is being destroyed properly now.

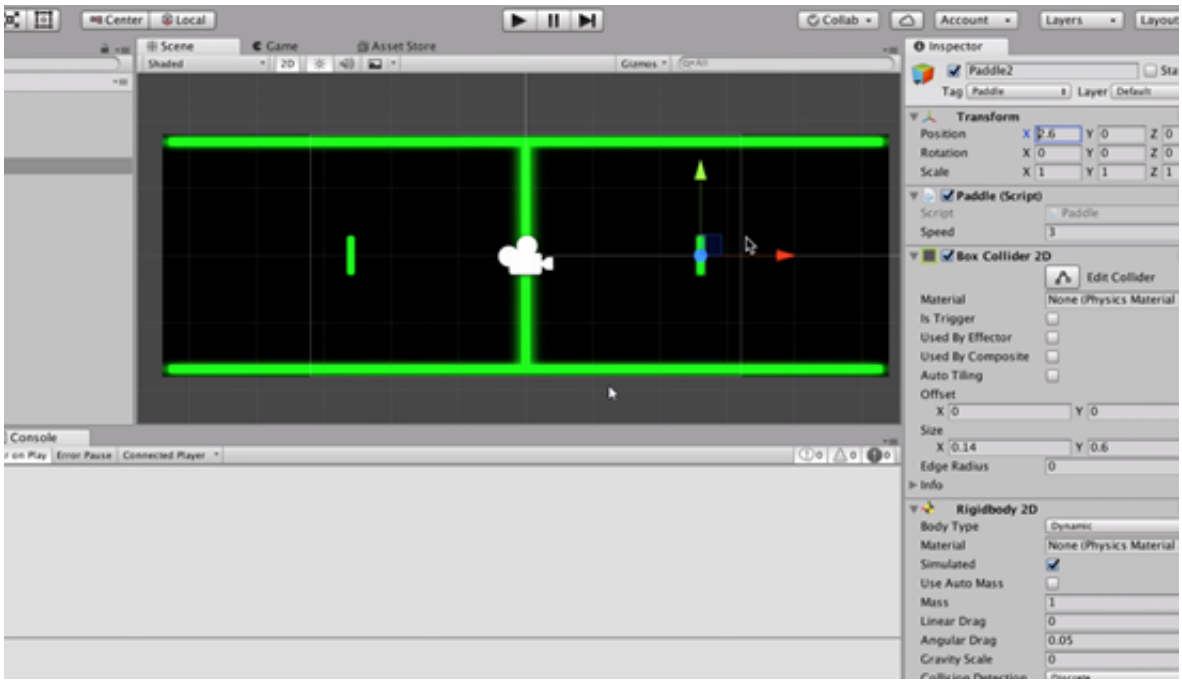
Save the scene and project and proceed to the next lesson.

To control two paddles we will need to make a few adjustments.

Second Paddle Creation

Duplicate the first paddle. Rename the first Paddle to “Paddle1” and rename the second Paddle to “Paddle2.”

Paddle2’s position on the X axis needs to be changed to 2.6.



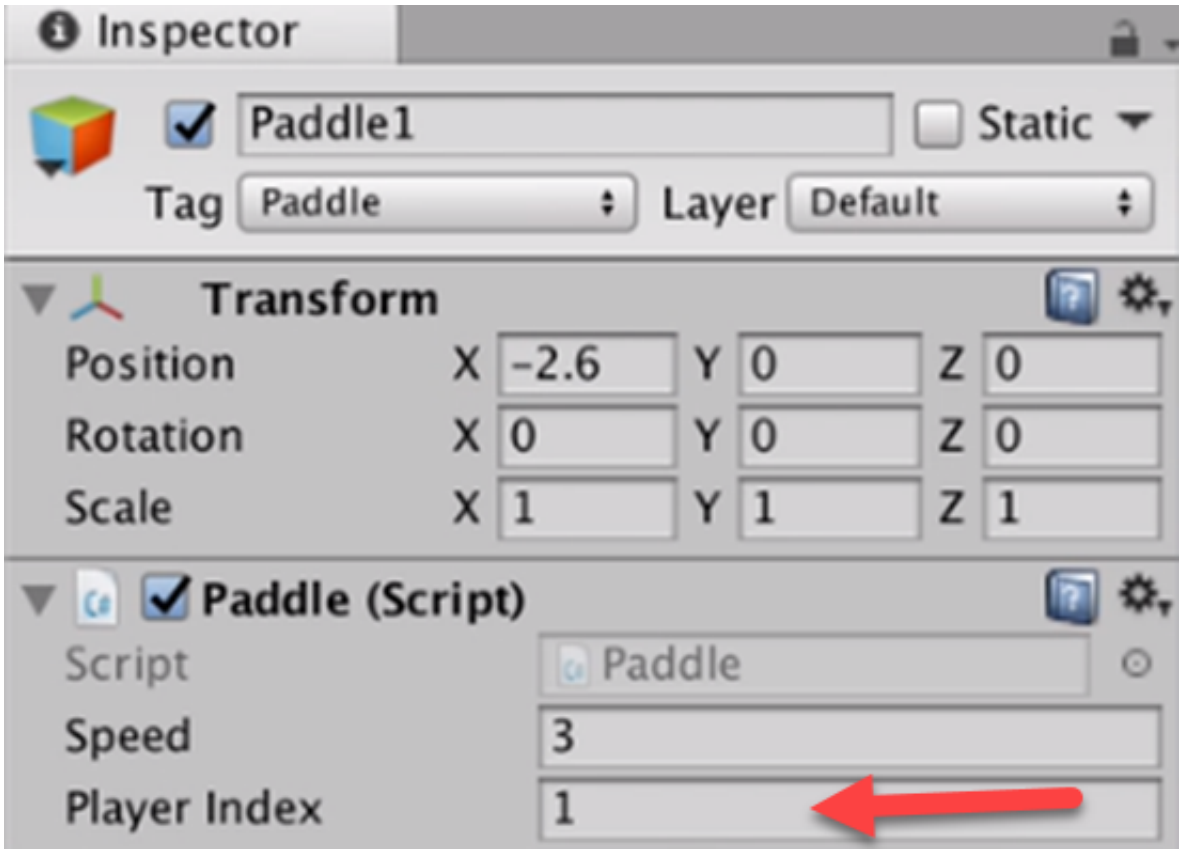
We also need to change the Paddle script to define whether or not the paddle is 1 or 2.

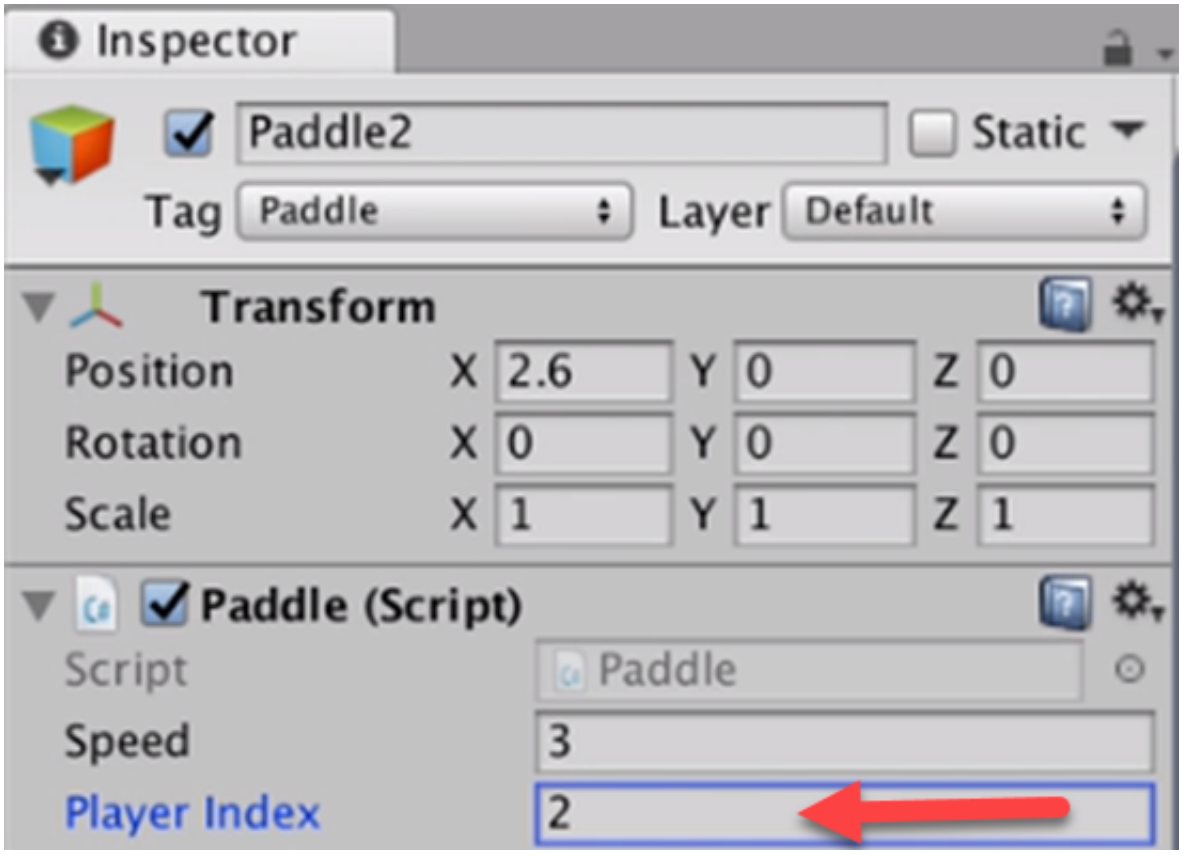
Open the Paddle script up in the code editor.

```
public float speed = 1f;
public int playerIndex = 1;

 // Use this for initialization
 void Start () {

 }

 // Update is called once per frame
 void Update () {
  float verticalMovement = Input.GetAxis ("Vertical" + playerIndex);

  GetComponent<Rigidbody2D> ().velocity = new Vector2 (
   0,
   verticalMovement * speed
  );
 }
```

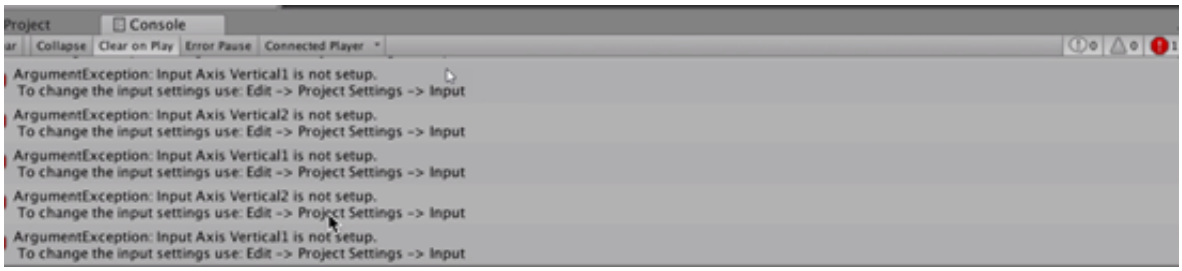
Save the script and change the Player Index value on the Paddle script component in the Inspector

for Paddle1 to Player Index =1 and Paddle2 Player Index = 2.

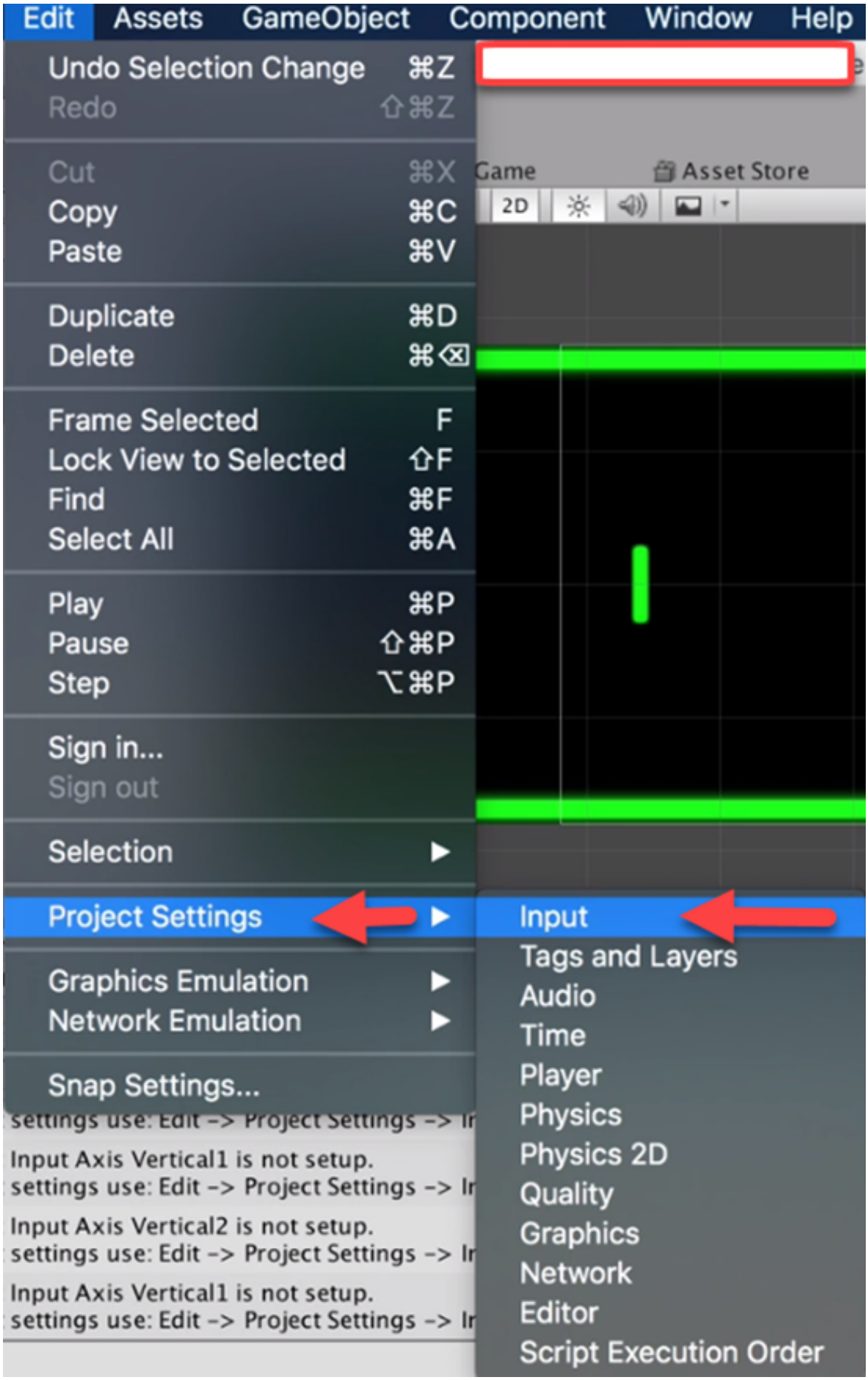




If you hit the Play button you will see we now get a ton of errors in the console:

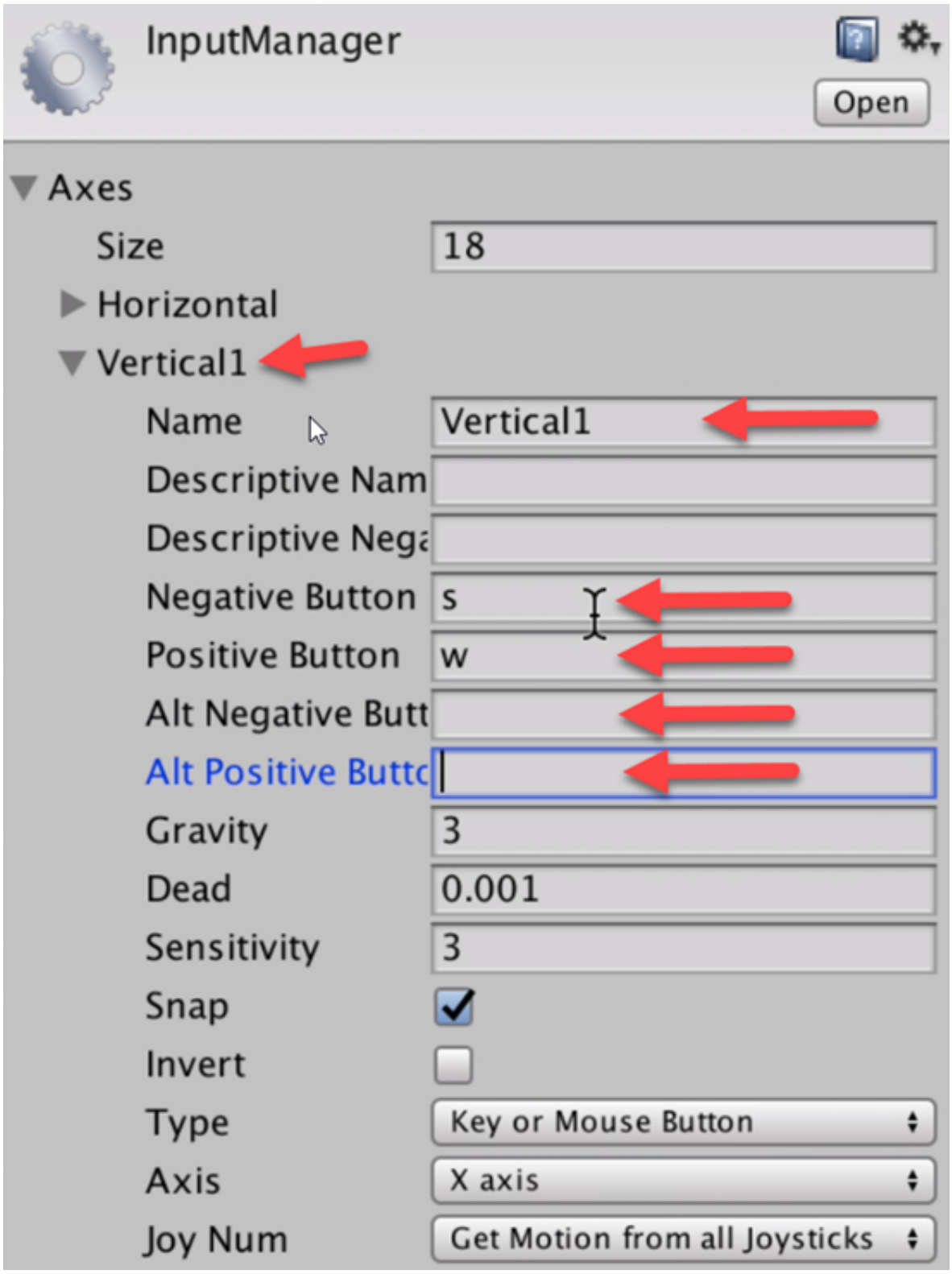


This is because we need to make some changes to the Input settings inside the Project Settings.

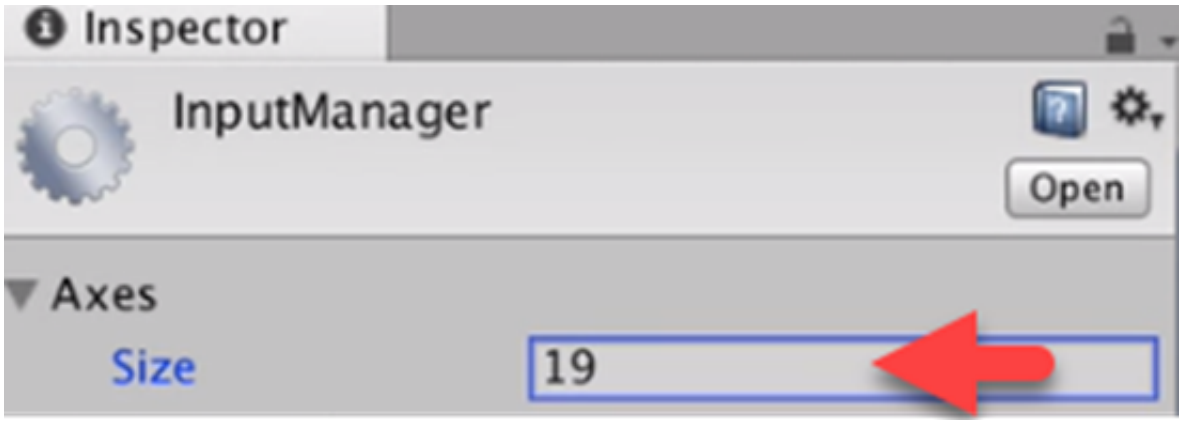


Hit Edit>ProjectSettings>Input.

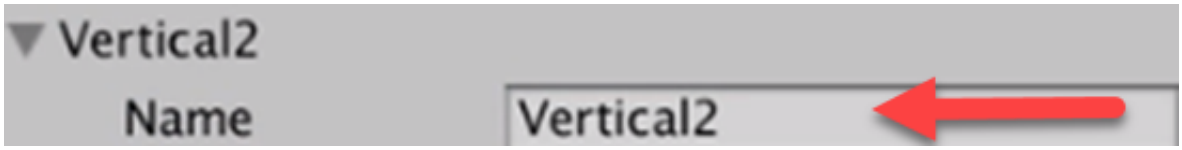
We need to change the name **Vertical** to “Vertical1.” Negative Button will be “S.” Positive Button will be “W.” Remove the Alt Negative Button and the Alt Positive Button. This will make it so the paddle on the left hand side can only be controlled witht he “S” and “W” keys on the keyboard.



Now change the Size of the Axes amount to 19 from 18.

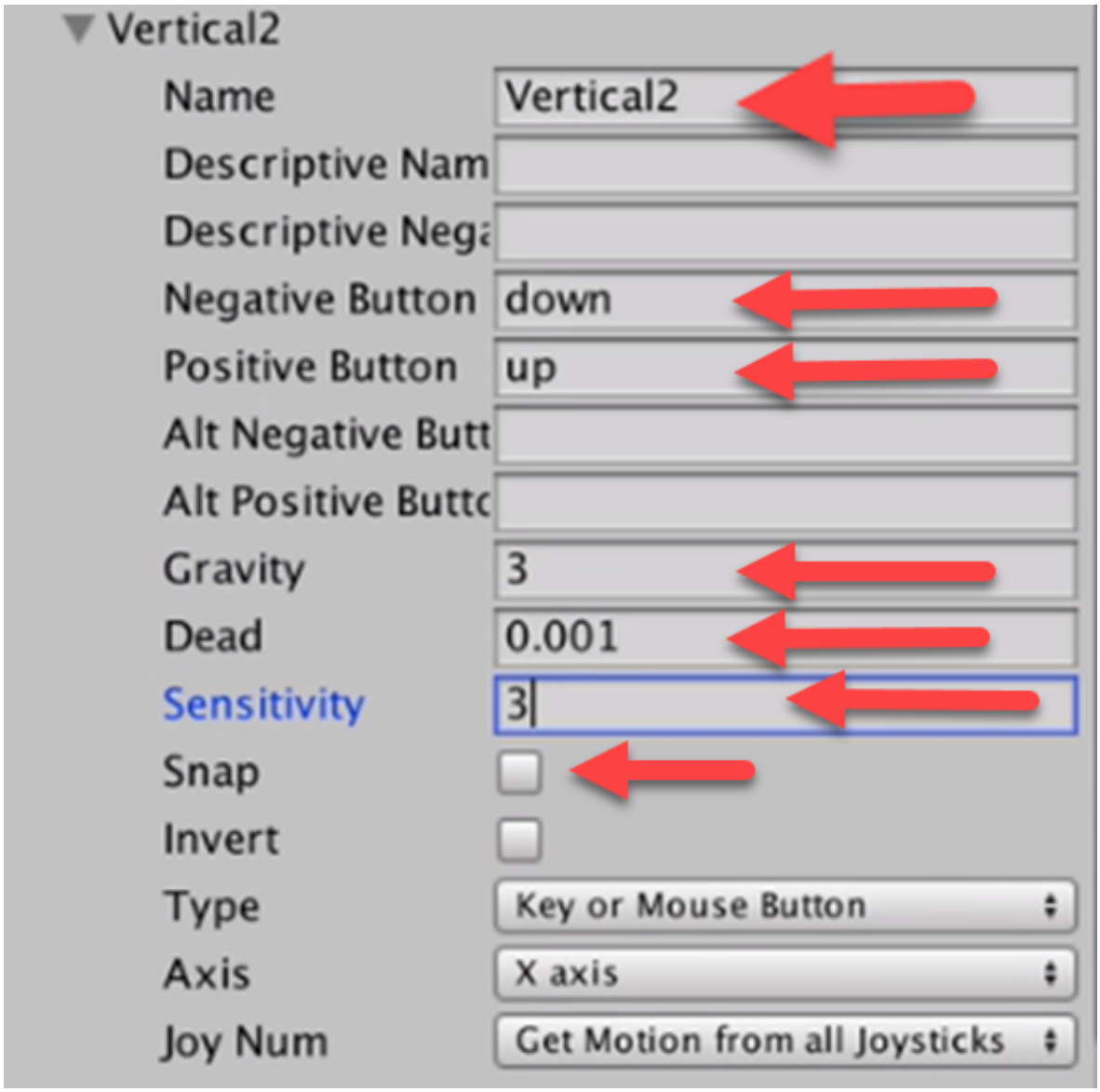


This will add a new Input option at the end of the list, just rename this from cancel to Vertical2.



The Negative Button will be changed to “down” Positive Button will be changed to “up” and remove the Alt Negative and Alt Positive buttons.

Change Gravity to 3, Dead to 0.001, and Sensitivity to 3.



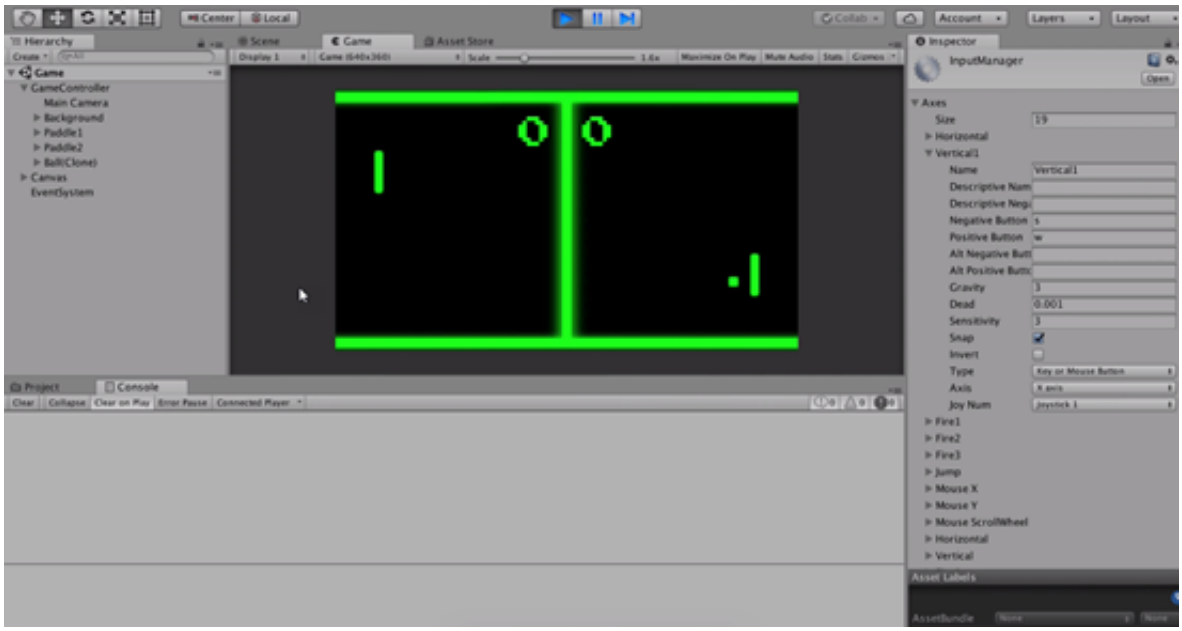
Make sure the Snap option is toggled for both Vertical 1 and Vertical 2.

For Vertical1, the Joy Num is going to be Joystick 1.

For Vertical2, the Joy Num is going to be Joystick 2.

Save the scene and project.

Hit the Play button in the Unity Editor and the errors we were getting before are now gone.



You are now able to control both paddles independently.

You can now proceed to the next lesson.

Open the Ball script up in the code editor.

Adding the Difficulty Modifier

We will be adding a public float variable to the Ball script that will allow us to change the game difficulty.

```
public float difficultyMultiplier = 1.3f;

 public float minXSpeed = 0.8f;
 public float maxXSpeed = 1.2f;

 public float minYSpeed = 0.8f;
 public float maxYSpeed = 1.2f;

 private Rigidbody2D ballRigidbody;

 // Use this for initialization
 void Start () {
  ballRigidbody = GetComponent<Rigidbody2D> ();
  ballRigidbody.velocity = new Vector2 (
   Random.Range(minXSpeed, maxXSpeed) * (Random.value > 0.5f ? -1 : 1),
   Random.Range(minYSpeed, maxYSpeed) * (Random.value > 0.5f ? -1 : 1)
  );
 }

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

 }

 void OnTriggerEnter2D (Collider2D otherCollider) {
  if (otherCollider.tag == "Limit") {
   GetComponent<AudioSource> ().Play ();

   // Collided with the top limit.
   if (otherCollider.transform.position.y > transform.position.y && ballRigidbody.vel
ocity.y > 0) {
    ballRigidbody.velocity = new Vector2 (
     ballRigidbody.velocity.x,
     -ballRigidbody.velocity.y
    );
   }

   // Collided with the bottom limit.
   if (otherCollider.transform.position.y < transform.position.y && ballRigidbody.vel
ocity.y < 0) {
    ballRigidbody.velocity = new Vector2 (
     ballRigidbody.velocity.x,
     -ballRigidbody.velocity.y
    );
   }
  } else if (otherCollider.tag == "Paddle") {
   GetComponent<AudioSource> ().Play ();
```



```
   // Collided with the left paddle.
   if (otherCollider.transform.position.x < transform.position.x && ballRigidbody.vel
ocity.x < 0) {
    ballRigidbody.velocity = new Vector2 (
     -ballRigidbody.velocity.x * difficultyMultiplier,
     ballRigidbody.velocity.y * difficultyMultiplier
    );
   }

   // Collided with the left paddle.
   if (otherCollider.transform.position.x > transform.position.x && ballRigidbody.vel
ocity.x > 0) {
    ballRigidbody.velocity = new Vector2 (
     -ballRigidbody.velocity.x * difficultyMultiplier,
     ballRigidbody.velocity.y * difficultyMultiplier
    );
   }
  }
 }
```

Save the script and go back to the Unity Editor and hit play to test out the changes.

You will notice that every time the ball hits a paddle the ball moves faster.

With the way we are adjusting the difficulty here, the player will really need to anticipate the movement of the paddle they are controlling.

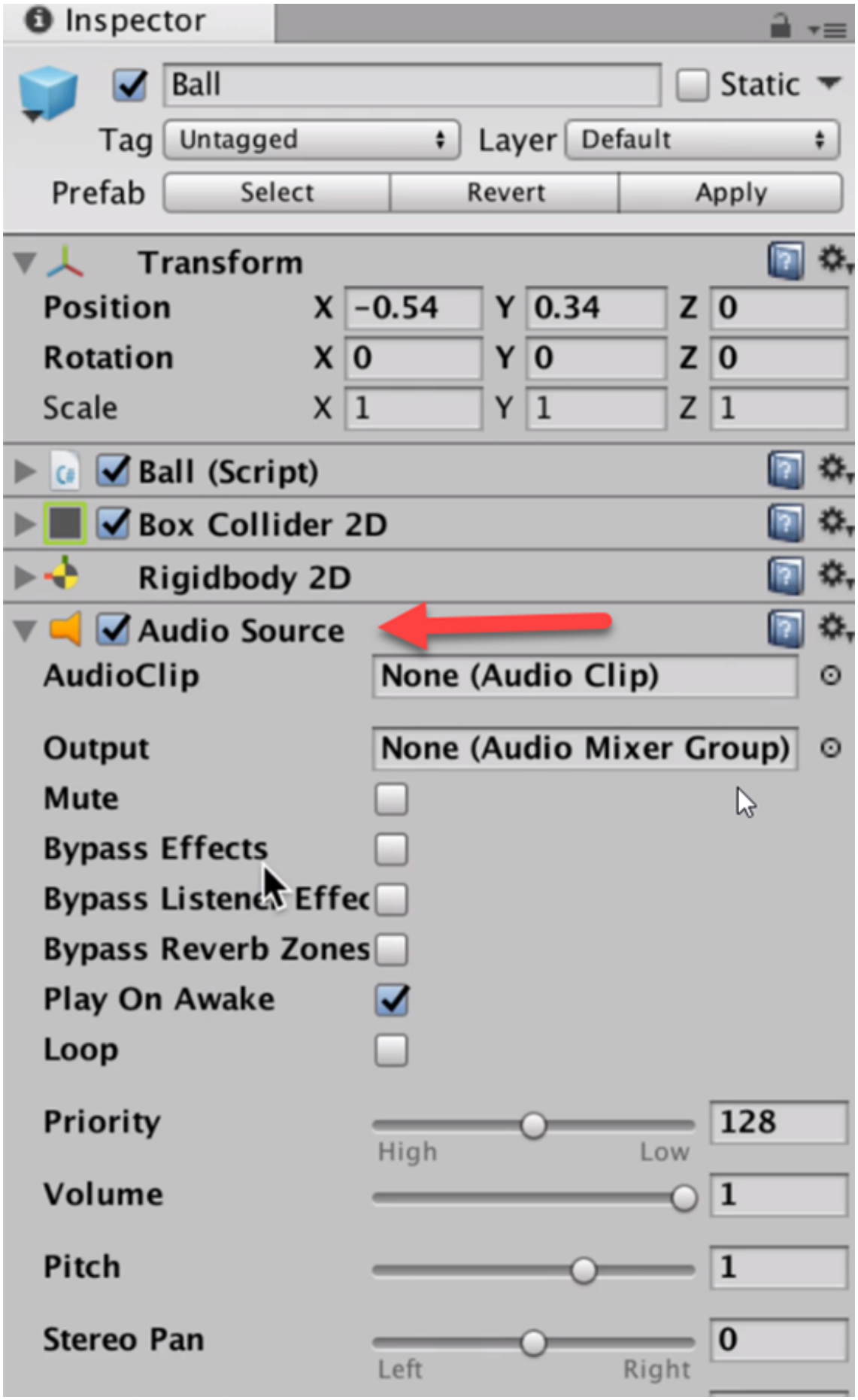
Save the scene and project and proceed to the next lesson where we will setup some audio sources.

There is just one more small thing remaining which is adding a **sound** to the game.

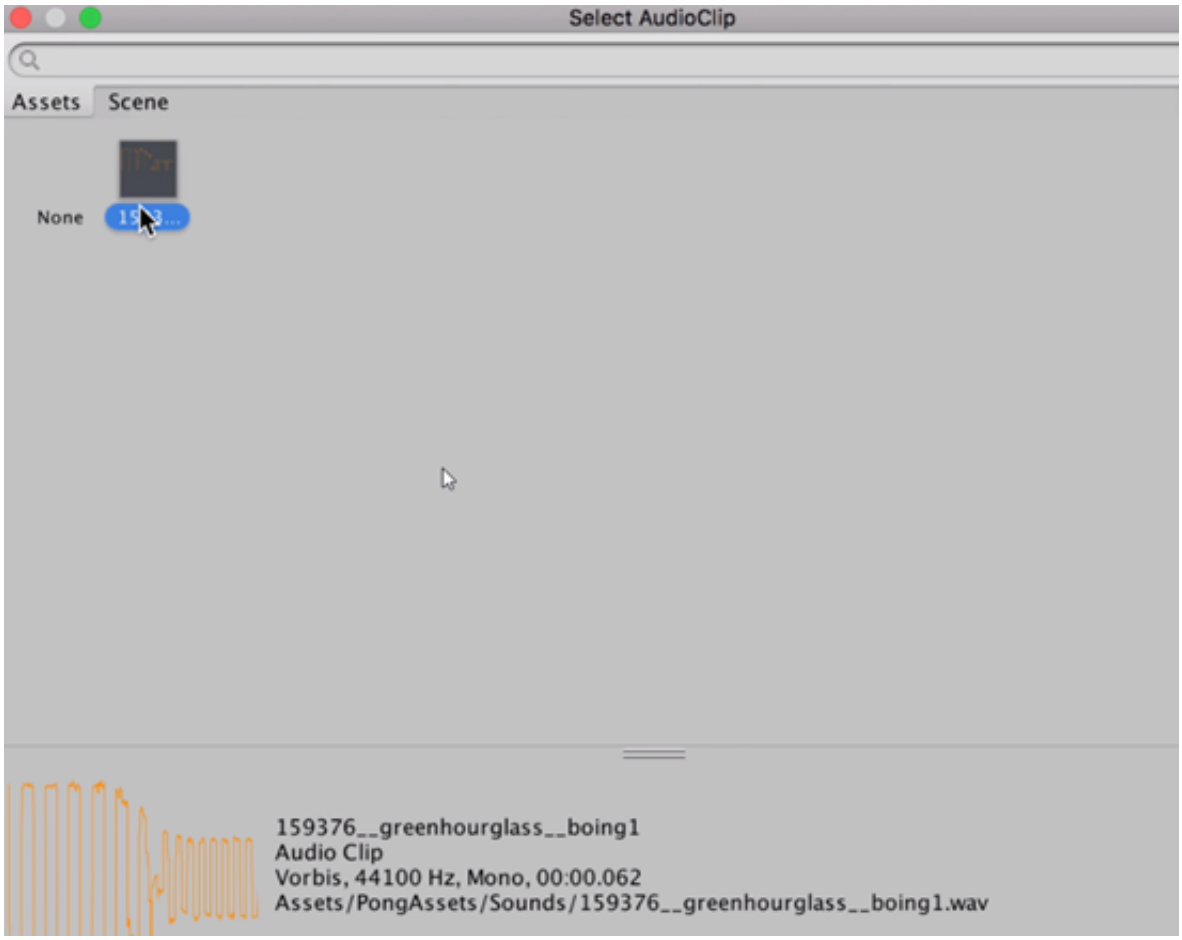
Adding Sound

Navigate to the **Prefabs** folder and drag the **Ball prefab** into the **Hierarchy**, with he ball selected, select the **Add Component button** from the Inspector window.

Add an **Audio Source component** to the Ball.

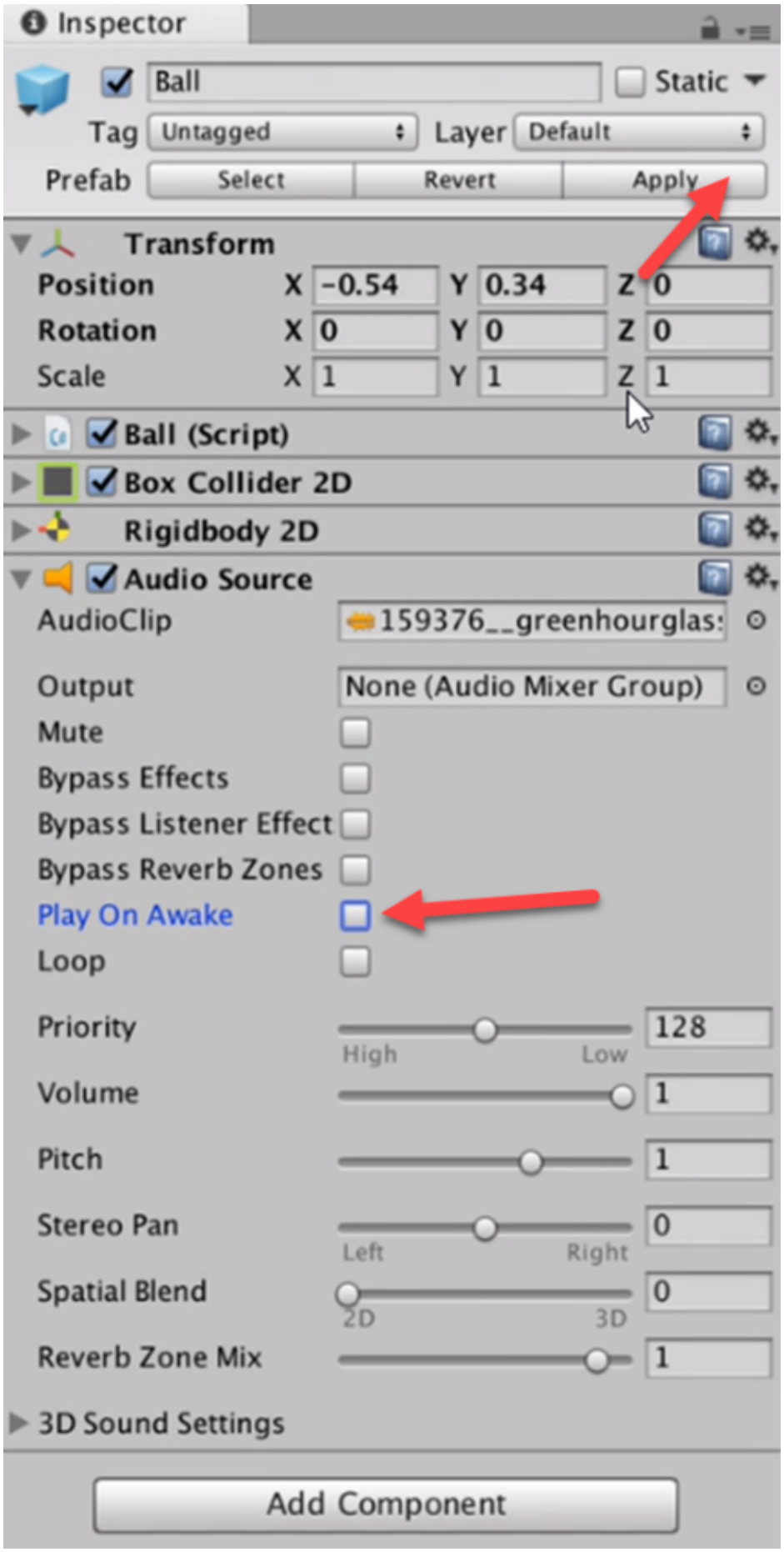


On the Audio clip field, if you click on the circle next to it, this will bring up a menu where you can select the sound effect from the course assets.



Make sure to uncheck the **Play on Awake** option, we don't want this sound to play when the Play button is hit every time so this is why we are disabling the Play on Awake option.

Hit the **Apply** button on the Prefab of the ball to apply the changes we just made to the Prefab.



Delete the Ball Prefab from the scene now.

Open the Ball script up in the code editor, we need to setup the sound to play so that every time the ball collides with a limit or a paddle the sound will play.

```
public float difficultyMultiplier = 1.3f;

 public float minXSpeed = 0.8f;
 public float maxXSpeed = 1.2f;

 public float minYSpeed = 0.8f;
 public float maxYSpeed = 1.2f;

 private Rigidbody2D ballRigidbody;

 // Use this for initialization
 void Start () {
  ballRigidbody = GetComponent<Rigidbody2D> ();
  ballRigidbody.velocity = new Vector2 (
   Random.Range(minXSpeed, maxXSpeed) * (Random.value > 0.5f ? -1 : 1),
   Random.Range(minYSpeed, maxYSpeed) * (Random.value > 0.5f ? -1 : 1)
  );
 }

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

 }

 void OnTriggerEnter2D (Collider2D otherCollider) {
  if (otherCollider.tag == "Limit") {
   GetComponent<AudioSource> ().Play ();

   // Collided with the top limit.
   if (otherCollider.transform.position.y > transform.position.y && ballRigidbody.vel
ocity.y > 0) {
    ballRigidbody.velocity = new Vector2 (
     ballRigidbody.velocity.x,
     -ballRigidbody.velocity.y
    );
   }

   // Collided with the bottom limit.
   if (otherCollider.transform.position.y < transform.position.y && ballRigidbody.vel
ocity.y < 0) {
    ballRigidbody.velocity = new Vector2 (
     ballRigidbody.velocity.x,
     -ballRigidbody.velocity.y
    );
   }
  } else if (otherCollider.tag == "Paddle") {
   GetComponent<AudioSource> ().Play ();

   // Collided with the left paddle.
   if (otherCollider.transform.position.x < transform.position.x && ballRigidbody.vel
```

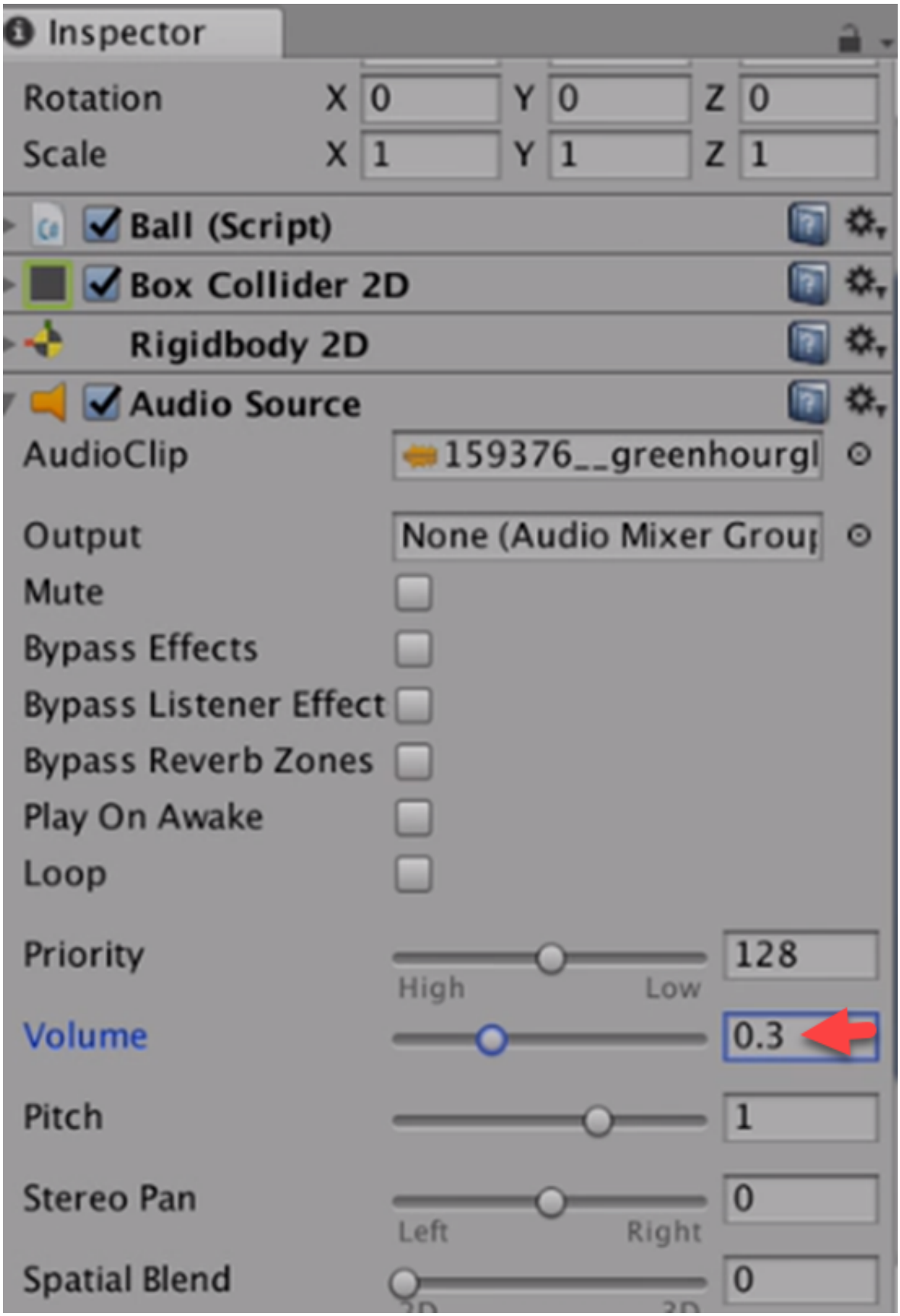


```
ocity.x < 0) {
    ballRigidbody.velocity = new Vector2 (
     -ballRigidbody.velocity.x * difficultyMultiplier,
     ballRigidbody.velocity.y * difficultyMultiplier
    );
   }

   // Collided with the left paddle.
   if (otherCollider.transform.position.x > transform.position.x && ballRigidbody.vel
ocity.x > 0) {
    ballRigidbody.velocity = new Vector2 (
     -ballRigidbody.velocity.x * difficultyMultiplier,
     ballRigidbody.velocity.y * difficultyMultiplier
    );
   }
  }
 }
```

Save the script and hit the Play button in the Unity Editor to test out the changes.

You will hear the sound effect play now once the ball collides with a limit or paddle.

If, you think the sound is too high, you can select the Ball from the Prefabs folder and change the **Volume value** on the Audio Source component.



Save the scene and project, and congratulations are now in order! You have successfully made a Pong game.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

GameController.cs

Found in Project/Pong Project/Assets/Project/Scripts

The GameController script manages a Pong-like game where it spawns balls in the center of the play area and keeps track of the players' scores. The players' scores are updated depending on whether the ball crosses the left or right scoring coordinates, and a new ball is spawned each time a point is scored.

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

public class GameController : MonoBehaviour {

  public GameObject ballPrefab;
  public Text score1Text;
  public Text score2Text;
  public float scoreCoordinates = 3.4f;

  private Ball currentBall;
  private int score1 = 0;
  private int score2 = 0;

  // Use this for initialization
  void Start () {
    SpawnBall ();
  }

  void SpawnBall () {
    GameObject ballInstance = Instantiate (ballPrefab, transform);

    currentBall = ballInstance.GetComponent<Ball> ();
    currentBall.transform.position = Vector3.zero;

    score1Text.text = score1.ToString ();
    score2Text.text = score2.ToString ();
  }

  // Update is called once per frame
  void Update () {
    if (currentBall != null) {
      if (currentBall.transform.position.x > scoreCoordinates) {
        score1++;
        Destroy (currentBall.gameObject);
        SpawnBall ();
      }
```



```
      if (currentBall.transform.position.x < -scoreCoordinates) {
        score2++;
        Destroy (currentBall.gameObject);
        SpawnBall ();
      }
    }
  }
}
```

Ball.cs

Found in Project/Pong Project/Assets/Project/Scripts

The Ball script controls the behavior of a ball in a Pong-like game. It assigns an initial random velocity to the ball and alters this velocity when collisions occur. If the ball hits a paddle, its velocity is reversed and increased as per the difficulty multiplier. If the ball hits the top or bottom limits of the scene, its y-velocity is reversed, creating a bouncing effect.

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

public class Ball : MonoBehaviour {

  public float difficultyMultiplier = 1.3f;

  public float minXSpeed = 0.8f;
  public float maxXSpeed = 1.2f;

  public float minYSpeed = 0.8f;
  public float maxYSpeed = 1.2f;

  private Rigidbody2D ballRigidbody;

  // Use this for initialization
  void Start () {
    ballRigidbody = GetComponent<Rigidbody2D> ();
    ballRigidbody.velocity = new Vector2 (
      Random.Range(minXSpeed, maxXSpeed) * (Random.value > 0.5f ? -1 : 1),
      Random.Range(minYSpeed, maxYSpeed) * (Random.value > 0.5f ? -1 : 1)
    );
  }

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

  }

  void OnTriggerEnter2D (Collider2D otherCollider) {
    if (otherCollider.tag == "Limit") {
      GetComponent<AudioSource> ().Play ();
```



```
      // Collided with the top limit.
      if (otherCollider.transform.position.y > transform.position.y && ballRigidbody.
velocity.y > 0) {
        ballRigidbody.velocity = new Vector2 (
          ballRigidbody.velocity.x,
          -ballRigidbody.velocity.y
        );
      }

      // Collided with the bottom limit.
      if (otherCollider.transform.position.y < transform.position.y && ballRigidbody.
velocity.y < 0) {
        ballRigidbody.velocity = new Vector2 (
          ballRigidbody.velocity.x,
          -ballRigidbody.velocity.y
        );
      }
    } else if (otherCollider.tag == "Paddle") {
      GetComponent<AudioSource> ().Play ();

      // Collided with the left paddle.
      if (otherCollider.transform.position.x < transform.position.x && ballRigidbody.
velocity.x < 0) {
        ballRigidbody.velocity = new Vector2 (
          -ballRigidbody.velocity.x * difficultyMultiplier,
          ballRigidbody.velocity.y * difficultyMultiplier
        );
      }

      // Collided with the left paddle.
      if (otherCollider.transform.position.x > transform.position.x && ballRigidbody.
velocity.x > 0) {
        ballRigidbody.velocity = new Vector2 (
          -ballRigidbody.velocity.x * difficultyMultiplier,
          ballRigidbody.velocity.y * difficultyMultiplier
        );
      }
    }
  }
}
```

Paddle.cs

Found in Project/Pong Project/Assets/Project/Scripts

The Paddle script is designed for a Pong-like game where it controls the vertical movement of a paddle. Using player input, the paddle is able to move at a specified speed on the screen. Different paddles can be assigned different player indexes to allow for multiple separate inputs.

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



```
public class Paddle : MonoBehaviour {

  public float speed = 1f;
  public int playerIndex = 1;

  // Use this for initialization
  void Start () {

  }

  // Update is called once per frame
  void Update () {
    float verticalMovement = Input.GetAxis ("Vertical" + playerIndex);

    GetComponent<Rigidbody2D> ().velocity = new Vector2 (
      0,
      verticalMovement * speed
    );
  }
}
```