Game Programming Using Unity 3D

Lesson 5: Creating The Camera System

Ferdinand Joseph Fernandez

Chief Technological Officer, Dreamlords Digital Inc.

Admin and Co-founder, Unity Philippines Users Group

September 2011

Overview

We'll be continuing where we left off from the previous lesson. If you haven't done Lesson 4, you need to do it before you can continue here.

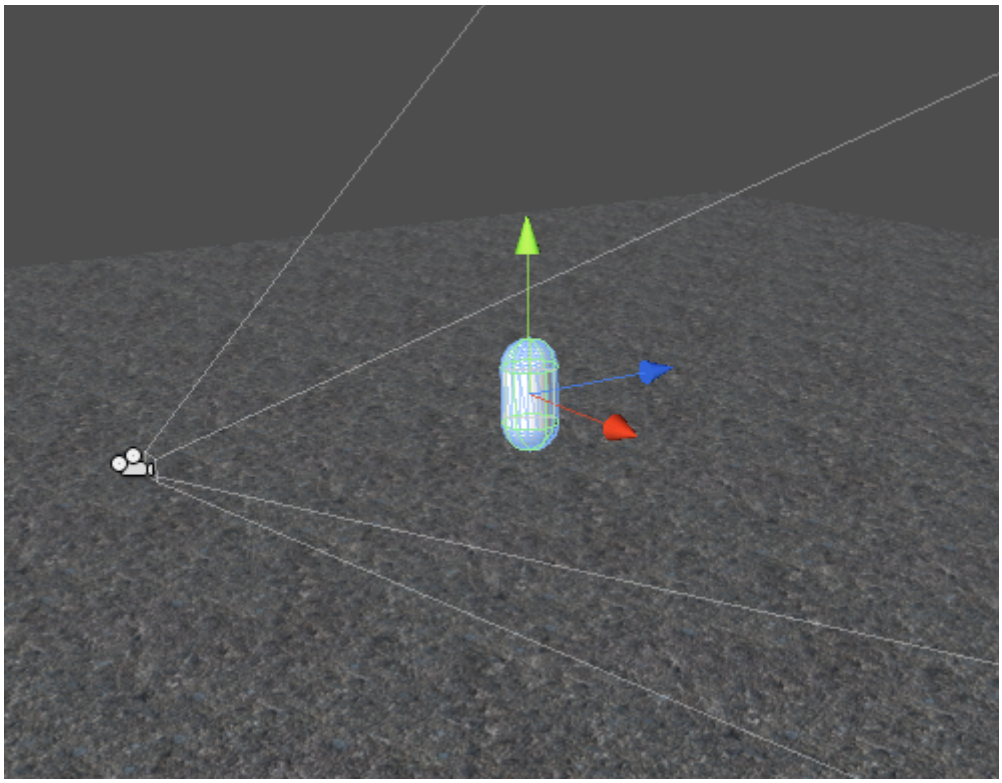
After creating the code to make the player move, we need code to let the player look around. Here we'll make a camera system modeled after so-called third person games, where the camera is fixated behind the character, even while he's moving.

Since we'll be controlling the camera this time, we'll be using mouse input instead of keyboard input.

Positioning The Camera

First off, to make the camera follow the player, parent your Main Camera game object to the Player game object.

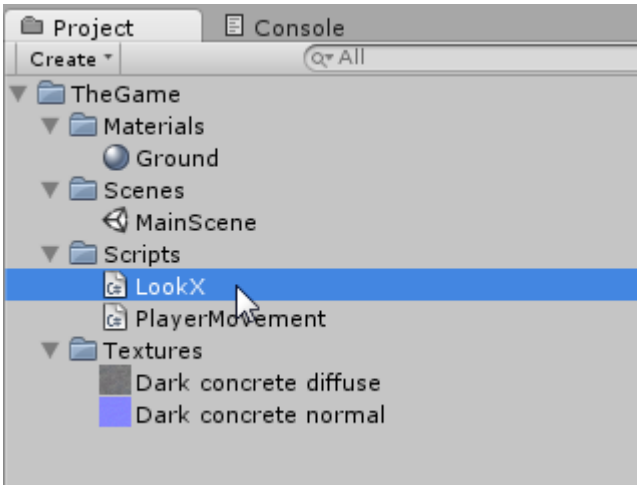
Make sure the camera ends up behind the player: set the camera's position to (0,0,-7). Make sure the camera's rotation is set to (0,0,0).



Remember to save your work often.

Looking Left And Right

In our “Scripts” folder, make a new script. Name it “LookX”.



Put this code in LookX:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class LookX : MonoBehaviour
05 {
06     [SerializeField]
07     float _mouseX = 0.0f;
08
09     void Update()
10     {
11         _mouseX = Input.GetAxis("Mouse X");
12     }
13 }
```

Now, attach the LookX script to your Player game object.

Run the game. When you move your mouse, you should see the Mouse X variable change. In this case, its not from -1.0 to 1.0 anymore. Mouse X holds by how much your mouse cursor moved in that instant (in that frame).

When you move the mouse to the left, the value is negative. When you move it to the right, the value is positive. This is the value that we'll use to know by how much we should rotate our camera.

Now change the code to this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class LookX : MonoBehaviour
05 {
06
07     float _mouseX = 0.0f;
08
09     void Update()
10     {
```

```
11            _mouseX = Input.GetAxis("Mouse X");
12
13            Vector3 rot = transform.localEulerAngles;
14            rot.y += _mouseX;
15            transform.localEulerAngles = rot;
16        }
17 }
```

We don't really need the Mouse X variable to be visible in the Inspector, so we remove the [SerializeField] from there.

Moving on, we added code to change the local euler angles of our transform. Remember, transform (with a small T) is a quick way to get our Transform component (so we don't need to use GetComponent here).

Basically, we're changing one of our rotation axes based on how much the mouse moved from left to right or right to left.

Play the game. When you move the mouse, you'll be able to change where the camera is looking.

Setting Mouse Sensitivity

Right now, looking around may be too slow. So like our movement code, we need to modulate this by a number that we can easily set to.

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class LookX : MonoBehaviour
05 {
06     [SerializeField]
07     float _sensitivity = 5.0f;
08
09     float _mouseX = 0.0f;
10
11     void Update()
12     {
13         _mouseX = Input.GetAxis("Mouse X");
14
15         Vector3 rot = transform.localEulerAngles;
16         rot.y += _mouseX * _sensitivity;
17         transform.localEulerAngles = rot;
18     }
19 }
```

We simply multiply the value of Mouse X with a number that we set from the Inspector. This is so we can make the looking faster or slower if we wanted.

Take note we don't need the mouse movement to be multiplied by Time.deltaTime. Mouse movement doesn't really slow down if you have a slow computer. Though the display of the mouse cursor can slow down, we're not dealing with the mouse cursor's display here.

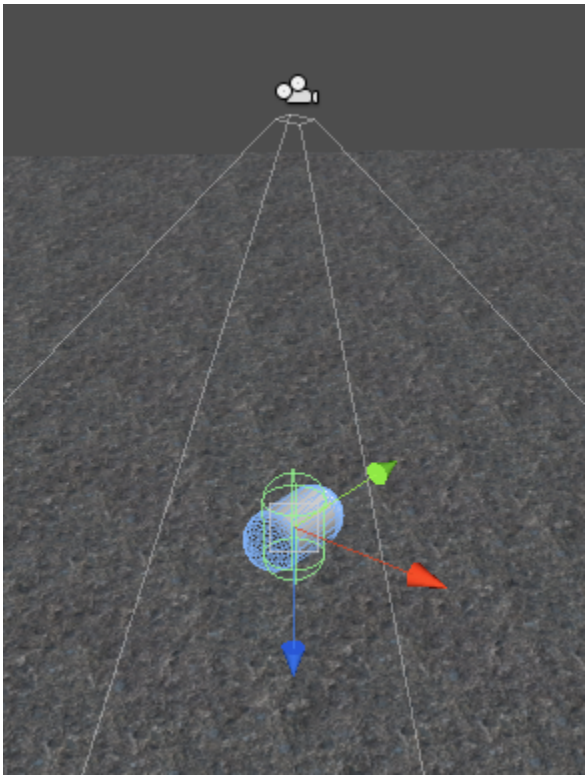
Looking Up And Down

Make a new class in the Scripts folder. Name it “LookY”. The code for this is similar, with some small changes:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class LookY : MonoBehaviour
05 {
06     [SerializeField]
07     float _sensitivity = 5.0f;
08
09     float _mouseY = 0.0f;
10
11     void Update()
12     {
13         _mouseY = -Input.GetAxis("Mouse Y");
14
15         Vector3 rot = transform.localEulerAngles;
16         rot.x += _mouseY * _sensitivity;
17         transform.localEulerAngles = rot;
18     }
19 }
```

Other than the appropriate name changes, another change is we are negating the return value of Input.GetAxis. This is because mouse Y input is inverted from what we want it to be. We also rotate by the X axis instead of the Y axis.

Now, you shouldn't attach this script yet or when looking down, you'll get what's happening in the picture to the right.



Remember we're changing the X axis rotation. So if you attached this to the player, the player object will rotate along with it. We obviously don't want it this way.

We want only the camera to rotate. But we can't put this in the camera game object either. If we do that, the camera's vision won't center on the player anymore.

Here's what we do. Stop the game if its running.

Remove the LookY script from the Player game object if its there.

Make a new game object. Name it “LookY”. Parent it to the Player game object. Make sure its position is (0,0,0). Make sure the rotation is (0,0,0), and the scale is (1,1,1).

Move the Main Camera game object inside LookY.

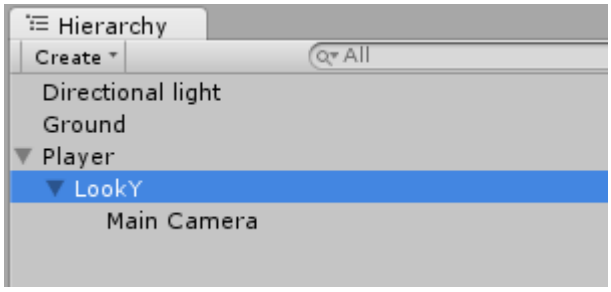


Illustration 1: Your Player game object's hierarchy should look like this.

Now assign the LookY script to the LookY game object.

Run the game. The camera system should now work as expected.

What's happening is the LookY game object acts as a pivot point for our camera.

Fixing Movement Direction

In case you haven't tried moving yet, you should do so now. You'll notice the movement direction doesn't respect where the camera is facing. This is a problem with world space and local space.

Recall in Lesson 1 about world space and local space. A world space Z axis points the same way no matter what. But the local space Z axis of an object points to where that object is facing.

When you think in world space, you are thinking in absolute values, like specifying longitude and latitude coordinates.

When you think in local space, you need to mention a reference point, like saying “my house is three blocks from the supermarket”. In that statement, “the supermarket” is your reference point, and “three blocks” is an offset from the reference point. That is how local space works.

Back to our player object, the problem is CharacterController's Move function expects a value in world space, but our velocity is in local space. So we need to convert this local space value into world space.

Taking our house and supermarket example, the idea is instead of saying our house is three blocks away from the supermarket, we need to specify the location of our house in exact longitude and latitude coordinates instead.

You could say that the position of objects in Unity can be expressed in either local space, or its world space equivalent. Both expressions point to the same thing. Its just a different way of pointing to it.

Fortunately, the code to do the conversion has been automated for us by Unity. We just need to make use of it. Go to your PlayerMovement script. See the highlighted code in yellow:

```
25      // Update is called once per frame
26      void  Update()
27      {
28          Vector3 direction = new  Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
29          Vector3 velocity = direction * _moveSpeed;
30
31          if  (_controller.isGrounded)
32          {
33              if  (Input.GetButtonDown("Jump"))
```

```
34                {
35                    _yVelocity = _jumpSpeed;
36                }
37            }
38            else
39            {
40                _yVelocity -= _gravity;
41            }
42
43            velocity.y = _yVelocity;
44
45            velocity = transform.TransformDirection(velocity);
46
47            _controller.Move(velocity * Time.deltaTime);
48        }
49 }
```

Line 45 is the only code you need to add to fix the direction problem. TransformDirection expects a Vector3 that has local space values and converts it to world space.

Remember, the transform variable (with a small T) is the transform of the game object that this C# script is attached to. In our case, its the Player's transform, which is what we want.

We assign the converted value back to the velocity variable, so that now holds world space values. Then we feed it into the Move function, just like before.

Information

Why are we bothering with local space and world space in the first place?

How come we have to do all these conversions? Well, as it turns out, its more convenient to think in terms of local space in some situations, and to think in world space in other situations. It all depends on the circumstance.

For example, saying “I want to move 5 meters per second ahead of where I'm currently facing” is easier than saying “I want to move 3.535 meters per second in the X axis and -3.535 meters per second in the Z axis”.

Conversely, saying “the entrance is at (3.4, 5.9, 12.0)” is more direct than saying “the entrance is (-1.3, 0, 0,) from the signpost, which is (45.23, 0, 3.2) from the exit”.

Sometimes you will want to specify in exact values, but sometimes you will want to specify values relative to the object's rotation.

Creating The Crosshair

Next we'll add a target crosshair for our player. This will be our first foray into a graphical user interface (GUI). For now, we'll just draw a picture at the center of the screen.

Create a new C# file. Name it “PlayerGui”.

Type this into the file:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class PlayerGui : MonoBehaviour
05 {
06     [SerializeField]
```

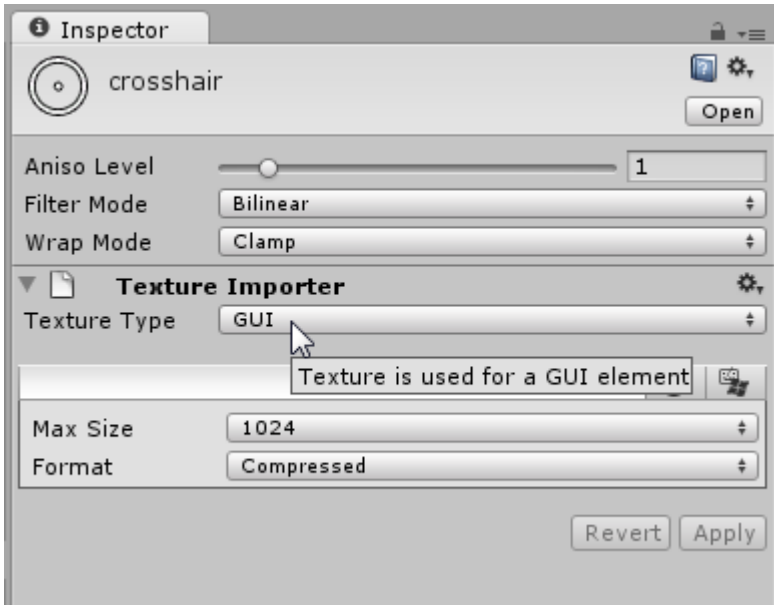
```
07      Texture2D _crosshair;
08
09      void  OnGUI()
10      {
11          GUI.DrawTexture(new  Rect(0, 0, 30, 30), _crosshair);
12      }
13 }
```

Texture2D is a class that holds data on an image being used. We make use of it in the OnGUI function, where GUI is drawn.

GUI.DrawTexture draws our image, in the given X and Y coordinates (0 and 0, the first two numbers) and in the width and height specified (30 and 30, the last two numbers).

Go to your Lesson Assets folder and find the crosshair image. Copy it to your project's “Textures” folder.

Next, this is important, in your Project View, click on the crosshair image. Your Inspector will show the Import Settings. You'll see a property called Texture Type. The value is currently set to “Texture”. You need change this to “GUI”. Then click the “Apply” button.



Now, drag the PlayerGui script to the Player game object. In the PlayerGui, assign the crosshair image (from the Project View) to the crosshair property (in the Inspector).

If you run the game, you should see the crosshair being drawn at the upper left corner of the screen.

First off, let's get rid of using constant values for width and height. We can make use of the actual image's width and height using this:

```
11          GUI.DrawTexture(new  Rect(0, 0, _crosshair.width, _crosshair.height), _crosshair);
```

Now we need to center this image on the screen. We do this by computing it using a formula.

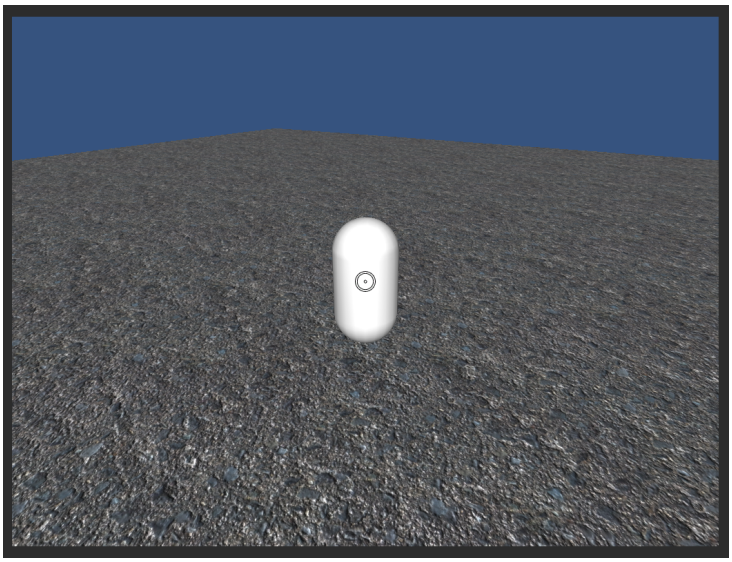
```
11          float  x = (Screen.width - _crosshair.width) / 2;
12          GUI.DrawTexture(new  Rect(x, 0, _crosshair.width, _crosshair.height), _crosshair);
```

Screen.width is our total screen width in pixels. Now doing the same for the Y coordinate:

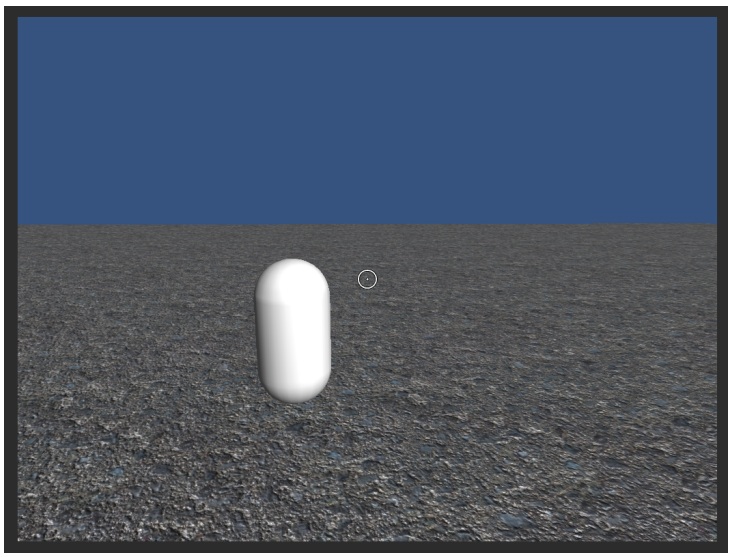
```
01 using  UnityEngine;
```

```
02 using System.Collections;
03
04 public class PlayerGui : MonoBehaviour
05 {
06     [SerializeField]
07     Texture2D _crosshair;
08
09     void OnGUI()
10     {
11         float x = (Screen.width - _crosshair.width) / 2;
12         float y = (Screen.height - _crosshair.height) / 2;
13         GUI.DrawTexture(new Rect(x, y, _crosshair.width, _crosshair.height), _crosshair);
14     }
15 }
```

Now you'll see the crosshair image centered nicely on the screen. However, our player is now in the way of the crosshair, so we'll move things a bit.



In your Player game object, find the “LookY” child game object. Set its position to (1, 0.5, 1). This will cause our camera to move a bit to the player's side. Gamers call this an “over-the-shoulder” camera view.



In Conclusion...

You've gotten through the second part of creating the player's controls. You can now control the camera using the mouse, and you've learned a little bit about Unity's GUI system.

In the next lesson, we'll start the foundation for the code of our enemy AI (artificial intelligence).