Game Programming Using Unity 3D

Lesson 4: Creating The Player Character Class

Ferdinand Joseph Fernandez

Chief Technological Officer, Dreamlords Digital Inc.

Admin and Co-founder, Unity Philippines Users Group

September 2011

Overview

This marks the start of the development of our game, “The Game”, a zombie-shooter in a third-person, chase camera style view.

The object of this course is centered on programming, so we'll be making use of ready-made art assets instead of making our own. For example, the player and zombie characters you see in the screenshot above are freely available from the Unity Asset Store.

In this lesson, we'll concentrate on the first thing: making a player character that walks and responds to user input.

Make a new empty project. Name the project folder, “TheGame”.

The Stage

First, we need a floor for our character to walk on. Make a new cube (GameObject > Create Other > Cube). Position it at (0, 0, 0). Scale it as (50, 1, 50) so it appears flat.

Name our cube “Ground”.

Let's add light. Choose GameObject > Create Other > Directional Light. Rotate it as (30, -30, 0).

Let's use a good texture for the ground. In our Lesson Assets folder, you'll find an image named “Dark concrete diffuse.png”, together with a “Dark concrete normal.png”. Copy them both to your Project.

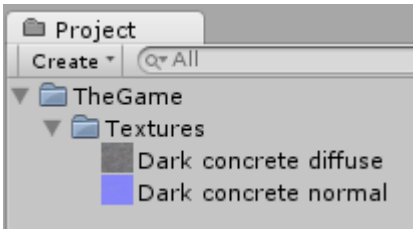
Folder Organization

Now would be a good time to lay down some rules on folder organization. Unity doesn't impose any rules for making your folders, but it has some special properties on some situations. (We don't need to worry about those yet).

We'll lay down our own rules so our files don't get messed up.

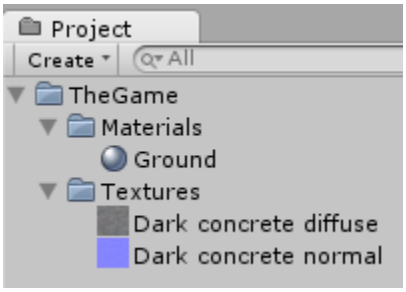
In your Project View, create a folder named “TheGame”. This will be our top-level folder where all assets specific to our game will be stored.

Inside “TheGame”, make a folder named “Textures”. Move the two dark concrete images in there.

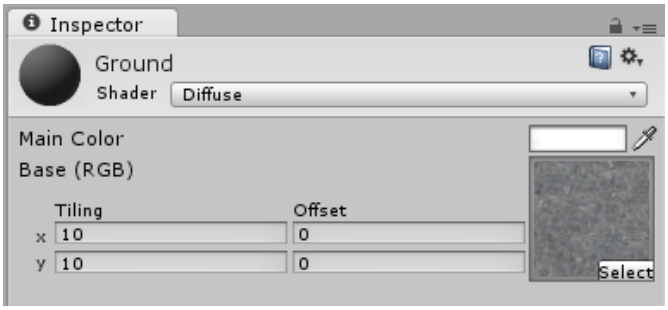


Assigning A Texture For Our Ground

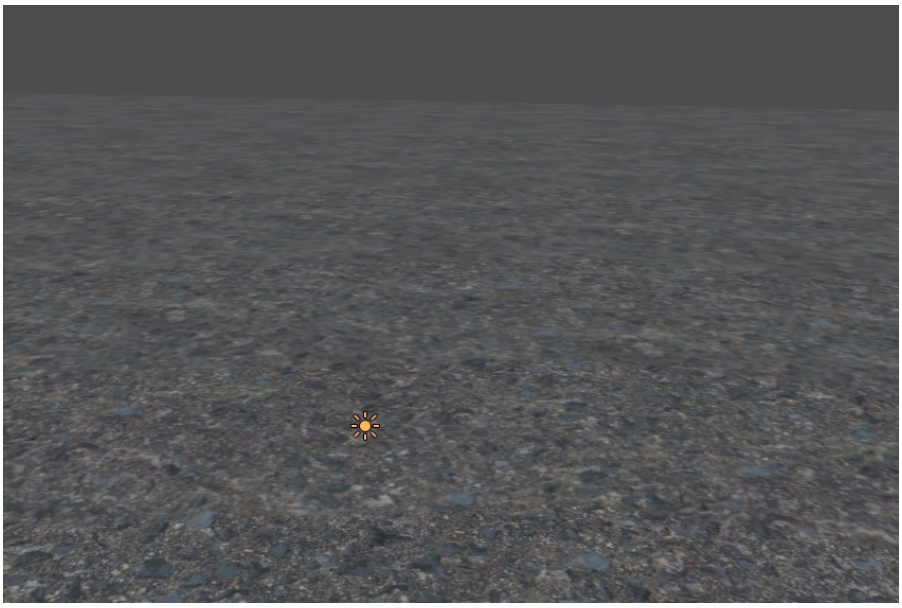
Now inside “TheGame”, make a folder named “Materials”. Select the Materials folder. Now create a new material inside, name it “Ground”.



Select your Ground material. Assign the “Dark concrete diffuse” texture to the Ground material. Set the Tiling property to 10 in both x and y.

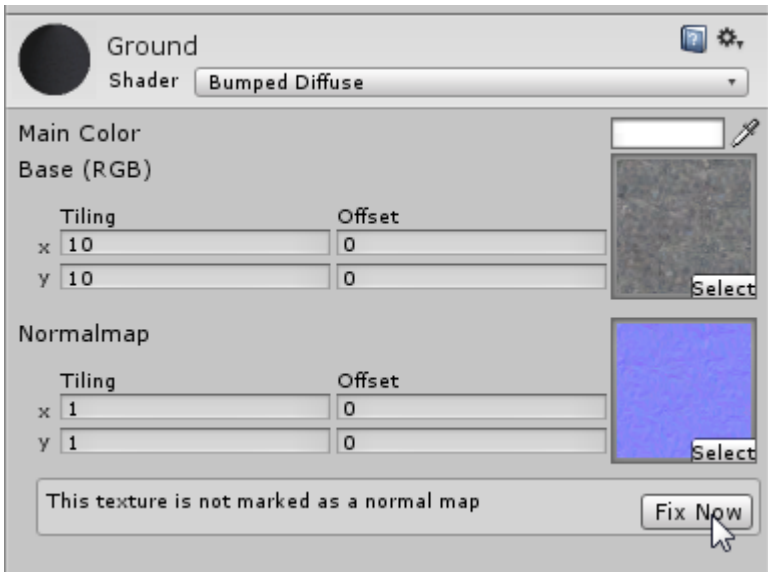


Now let's assign this Ground material to our cube. Select the Cube from the Hierarchy View. You'll see a Mesh Renderer component. Expand its Materials property. Drag the Ground material from the Project view into the Element 0, replacing the Default-Diffuse that's currently assigned there.



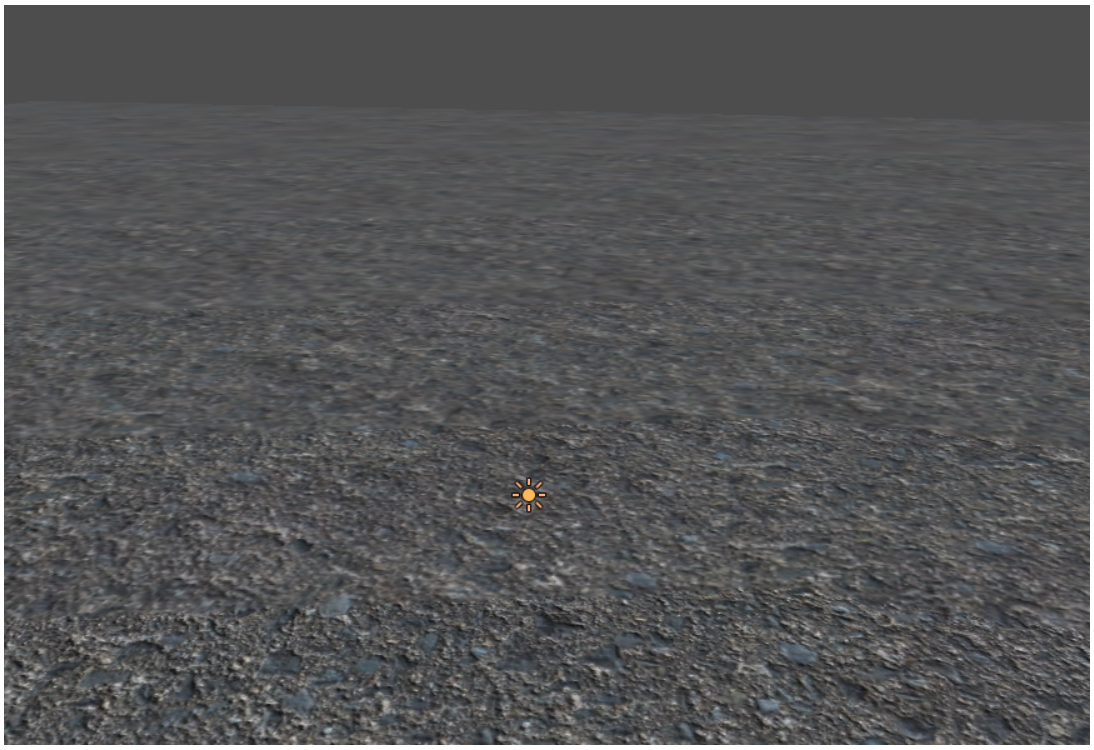
We'll change the shader to take advantage of the so-called “normal” image (the other image we imported). Change the shader to use “Bumped Diffuse”. Assign the “Dark concrete normal” image to the second texture slot that's currently empty.

You'll see it says the texture is not marked as a normal map. Just click on the button that says “Fix Now”.



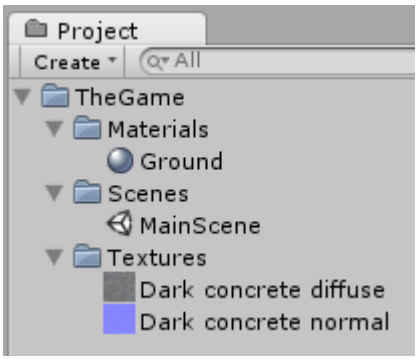
You'll also have to change the Tiling property for the normal map to 10 in both x and y.

As a result, you'll see the texture “pop out” more because of the normal map.



Save Your Work

Now would be a good time to save your work if you haven't yet. Press Ctrl + S to save. It'll prompt you where you want to save your scene. Create a new folder named “Scenes” from inside the “TheGame” folder. Save your scene there. Name it “MainScene”.



The Character Controller

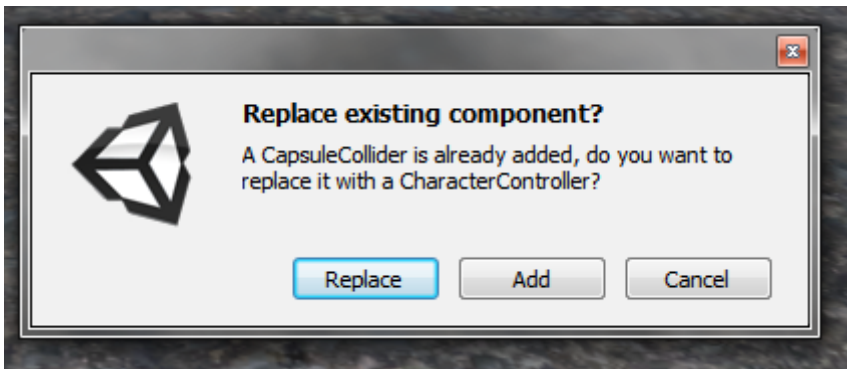
In Unity, the basic method to have a moving character is to use the Character Controller component. This Character Controller component isn't of much use by itself; we need to control it from a script.

Let's make use of a placeholder for our player for now. Create a capsule (GameObject > Create Other > Capsule). Position it to (0, 1.75, 0). If you play the game now, you should see the capsule centered nicely on our screen.

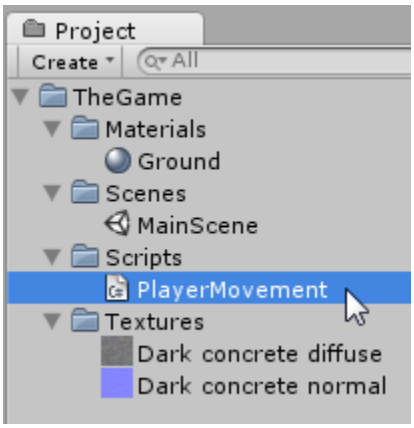
Name the capsule game object, “Player”.

Now let's add the Character Controller component. Select the Capsule. Go to Component > Physics >

Character Controller. You'll be warned with a message saying the capsule collider has to be replaced. Go ahead and click “Replace”.



Now we'll create a script for the player. In the “TheGame” folder, make a new folder named “Scripts”. This will be where all our script files reside. Inside the “Scripts” folder, make a new C# file. Name it “PlayerMovement”.



Select the Player game object. Drag the PlayerMovement script from the Project View into the empty area in the Inspector. The Player should now have the PlayerMovement script component.

Now let's add some code.

Getting Keyboard Input

First, a little explanation about retrieving user input from the keyboard. Add the following code to our PlayerMovement.cs file:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class PlayerMovement : MonoBehaviour
05 {
06     public float h = 0;
07
08     // Use this for initialization
09     void Start()
10     {
11
12     }
```

```
13
14      // Update is called once per frame
15      void Update()
16      {
17          h = Input.GetAxis("Horizontal");
18      }
19 }
```

Don't worry about the variable being public. We'll be removing it later. The reason we have that variable public right now is so you could see its value in the Inspector.

Select your player game object. Run the game. Hold either the left or right arrow keys. Take a look at the Inspector. You should see the value of H change from 0.0 to 1.0 or -1.0 depending on which key you pressed. You can also use the A and D keys.

This is how we retrieve player input. Input.GetAxis gives us a value from -1.0 to 1.0 depending on the kind of user input that we want. The “Horizontal” string that we specify is, by default, for the left and right movement.

Getting The Character Controller

We'll remove that Input code for now. Meanwhile, change your code to exactly this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class PlayerMovement : MonoBehaviour
05 {
06      CharacterController _controller;
07
08      // Use this for initialization
09      void Start()
10      {
11          _controller = GetComponent<CharacterController>();
12      }
13
14      // Update is called once per frame
15      void Update()
16      {
17      }
18 }
```

To move our character, we need to call on the help of our Character Controller component. And we need to get a handle, or a reference, to our Character Controller component. Line 6 creates a variable where we will store the handle to our existing Character Controller component.

We initialize it with a value in our Start function. This is important. Don't use the constructor for classes that inherit from MonoBehaviour, or it will mess up with Unity's way of doing things.

GetComponent is a function provided for us by MonoBehaviour. It allows us to get a handle on an existing component. Remember we added a Character Controller to our Player game object in the Unity Editor. We're just accessing that here.

Moving The Player

Now finally let's add again the Input code. That will make our player move:

```
01 using  UnityEngine;
02 using  System.Collections;
03
04 public class  PlayerMovement : MonoBehaviour
05 {
06       CharacterController _controller;
07
08       // Use this for initialization
09       void  Start()
10       {
11            _controller = GetComponent<CharacterController>();
12       }
13
14       // Update is called once per frame
15       void  Update()
16       {
17            Vector3 direction = new  Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
18            _controller.Move(direction);
19       }
20 }
```

We're making use of Input.GetAxis to detect user input. We use “Horizontal” for left and right movement, and “Vertical” for forward and backward movement. Then we're feeding that value into a temporary Vector3 variable.

Vector3 is a struct that holds x, y, z values. Vector3 is what we use to hold position values, rotation values, scale values, and so on. We use it here so its easier to compute our movement code.

The “Horizontal” user input goes into the X coordinate, and the “Vertical” user input goes into the Z coordinate. The Y coordinate is left at zero, since we don't want our player to move up and down; he doesn't fly.

Look at line 18. The Move function of our Character Controller is what causes our game object to move. And we're passing the Vector3 to that Move function. So, in effect, we are now moving our game object from user input.

If you play the game, you can now move your player with the arrow keys on your keyboard, or the WASD keys.

Modulate By Framerate

Now recall its not a good idea to just use constant values. We need to modulate the speed by the computer's frame rate. Change line 18 to this:

```
18            _controller.Move(direction * Time.deltaTime);
```

Recall in your early Science lectures that speed = distance over time. Well in this case we are getting distance, which is speed multiplied by time. And that is exactly what we're doing here. The direction variable

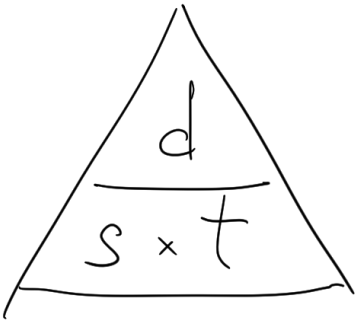


Illustration 1: You may remember this from your early Science lectures.

connotes our speed, and Time.deltaTime is our time.

Modulate By Custom Speed

Now the problem is the movement is really slow. Remember Input.GetAxis returns a value from -1.0 to 1.0. So now we are actually moving our player game object by 1 (or -1) meter per second in both X and Z axes. We need to multiply these values to something that we can change easily. Change the code to this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class PlayerMovement : MonoBehaviour
05 {
06     CharacterController _controller;
07
08     [SerializeField]
09     float _moveSpeed = 5.0f;
10
11     // Use this for initialization
12     void Start()
13     {
14         _controller = GetComponent<CharacterController>();
15     }
16
17     // Update is called once per frame
18     void Update()
19     {
20         Vector3 direction = new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
21         Vector3 velocity = direction * _moveSpeed;
22         _controller.Move(velocity * Time.deltaTime);
23     }
24 }
```

Now we're multiplying each of our direction x, y, z values by a single number. So, in this code, our -1.0 to 1.0 range becomes -5.0 to 5.0. We're now moving our player by 5 meters per second. Recall that in Physics, velocity is speed with direction. If you look at line 21, that's exactly what we're doing there.

Making The Character Jump

Now we'll make the character jump when pressing spacebar. Change the code to this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class PlayerMovement : MonoBehaviour
05 {
06     CharacterController _controller;
07
08     [SerializeField]
09     float _moveSpeed = 5.0f;
10
```

```
11      [SerializeField]
12      float _jumpSpeed = 20.0f;
13
14      [SerializeField]
15      float _gravity = 1.0f;
16
17      float _yVelocity = 0.0f;
18
19      // Use this for initialization
20      void Start()
21      {
22          _controller = GetComponent<CharacterController>();
23      }
24
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          _controller.Move(velocity * Time.deltaTime);
46      }
47 }
```

We got a few new variables that we can tweak: “_jumpSpeed” controls how high we can jump, and “_gravity” controls how fast we fall. The “_yVelocity” variable records our current Y velocity. Its the variable that we modify when jumping or falling.

In line 31, the isGrounded detects whether our Character Controller is on the ground. Its only then that we allow the player to jump.

In line 33, Input.GetButtonDown detects whether we pressed a button. The string “Jump”, by default, is assigned to the spacebar. So when you press the spacebar, our Y velocity is suddenly set to the value of our jump speed (line 35). This causes the player to jump.

Line 40 gets called if he's not on the ground (if he's afloat). In that case, we reduce his Y velocity to simulate gravity. Eventually it'll be a negative value, which causes our player to fall down when he's in the air.

The value of “_yVelocity” is eventually fed into our Move function (line 43 and line 45). Remember its the Move function that actually does the moving.

In Conclusion...

You've gotten through the basics of converting user input into something your game can use: you've made a game object respond to keyboard controls.

You also learned a little about file organization in your project.

In the next lesson, we'll be making code to control the camera with the mouse, so the player can look around better.