Game Programming Using Unity 3D

Lesson 8: Damaging

Ferdinand Joseph Fernandez

Chief Technological Officer, Dreamlords Digital Inc.

Admin and Co-founder, Unity Philippines Users Group

September 2011

Overview

Again, we'll be continuing where we left off from the previous lesson. If you haven't done Lesson 7, you need to do it before you can continue here.

So we got our enemy zombies chasing us. Now we'll be adding code to let the player shoot those zombies.

Adding Health

Before we can damage enemies, they need to have the concept of health with them. Basically we'll have an integer value that starts at 100 (for example). That 100 indicates full health. It gets subtracted every time the character gets damaged. 50 would mean half-health. Upon reaching zero, the character is considered dead.

We'll make the Health be its own class. Create a new C# script and name it “Health”. The contents will be simple for now:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Health : MonoBehaviour
05 {
06     [SerializeField]
07     int _maximumHealth = 100;
08
09     int _currentHealth = 0;
10
11     void Start()
12     {
13         _currentHealth = _maximumHealth;
14     }
15 }
```

We have a “_maximumHealth”. This is used to set the initial health value of the character. We can set this to any value we want. For example, we can set it to a high value for enemies that are hard to kill and low values for enemies that we want to be easy to kill.

Then “_currentHealth” keeps track of the current health of that character.

Now we'll add a new function:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Health : MonoBehaviour
05 {
06     [SerializeField]
07     int _maximumHealth = 100;
08
09     int _currentHealth = 0;
10
11     void Start()
```

```
12      {
13          _currentHealth = _maximumHealth;
14      }
15
16      public void Damage(int damageValue)
17      {
18          _currentHealth -= damageValue;
19      }
20  }
```

Our Damage function takes care of subtracting the current health with whatever the damage value is. We'll be calling this function later when our player deals damage to the enemies.

Now let's add this Health script to all of our enemies.

Select the Enemy prefab from the Project View. Now in the top menu bar of Unity, go to Component > Scripts > Health. This will add the Health script component to our Enemy prefab. Now all of the enemies in the scene should have the Health component.

Shooting Using Raycasts

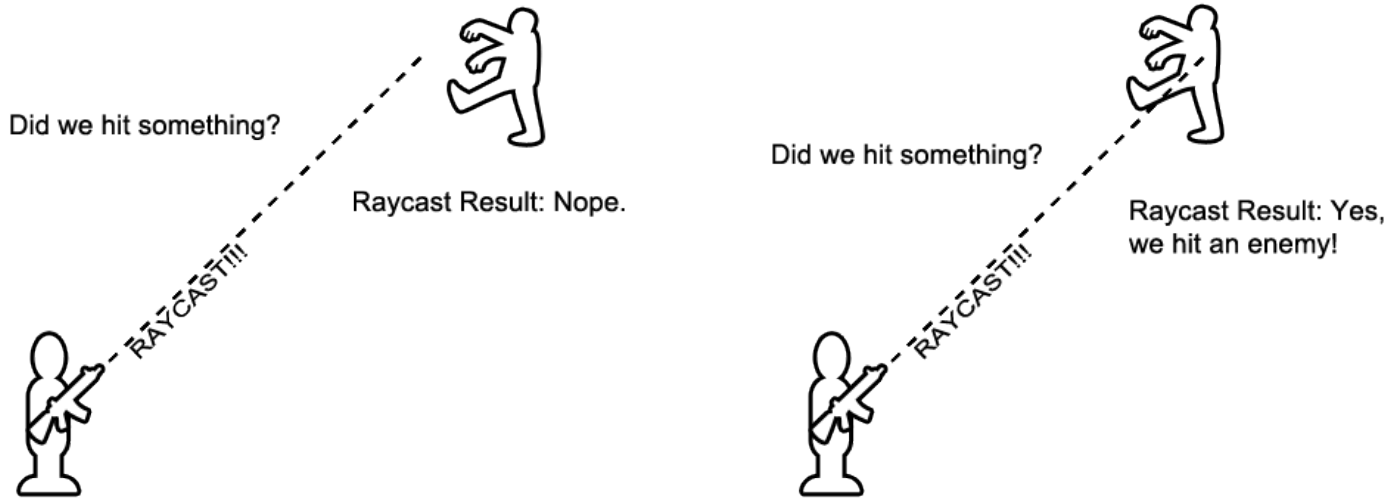
Now we'll make a script that allows the player to damage the enemies. We'll be making a simple gun that fires bullets. Make a new C# script. Name it “RifleWeapon”.

This script will be the one that calls the Damage function of the Health script.

For this script to know if it hit an enemy, it will use the method known as “raycasting”.

Raycasting simply means drawing a line somewhere in the 3d world and checking if that line is intersecting any object.

So what we'll do is, we draw a line starting from the player, extending forwards. We then check if that line intersected with an enemy. If it did, we'll call the Damage function of that Enemy's Health.



In effect, we don't really have bullets that travel in the 3d world, we just draw a line that will serve as the bullet's trajectory. Raycasts are instantaneous. The line doesn't get drawn slowly; it gets computed in an instant. So, we're just faking the bullet's movement as an instantaneous line. In

videogames, this is known as “hitscan”.

For now, add this code first:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : MonoBehaviour
05 {
06     void Update()
07     {
08         Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
09         RaycastHit hitInfo;
10
11         if (Physics.Raycast(mouseRay, out hitInfo))
12         {
13             Debug.Log(hitInfo.transform.name);
14         }
15     }
16 }
```

This piece of code makes a raycast, with the line starting from the center of the screen, outwards. Then it simply prints the name of the object that it intersected with.

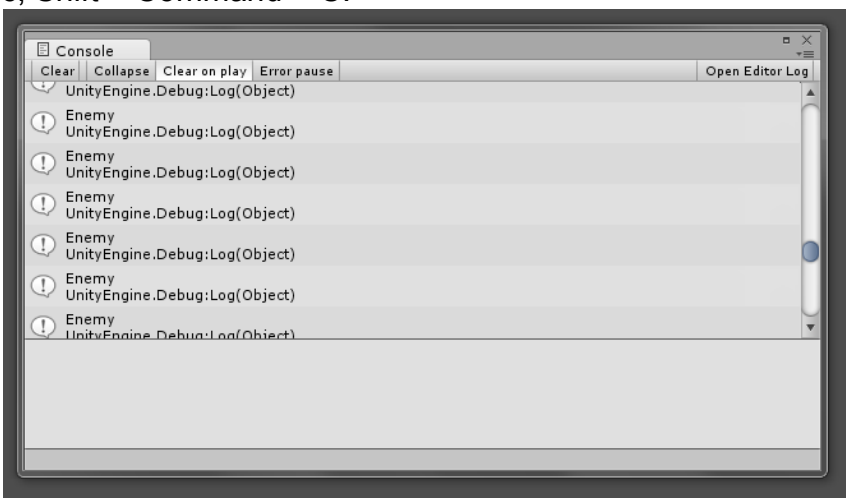
Line 8 creates a Ray whose starting point is the center of the screen (that is, the current camera being used). The two 0.5 values indicate that we want it to be the center of the screen. The first 0.5 is the X axis and the second is for the Y axis. The Z axis indicates the depth. Zero for depth means it should start from where the camera is. A negative value would be behind the camera, and a positive value would be ahead of the camera.

Line 9 creates a local variable that will hold the results of the raycast (which object it hit, how far it was, etc.). RaycastHit is a struct.

Line 11 does the actual raycast. The raycast function returns true if it hit something so we put it in an if statement for convenience. The “out” keyword is used by C# to indicate that the variable passed (hitInfo) should have its contents changed.

Line 13 gets called if the raycast intersected with something. The name of that something, is then printed out into Unity's console. Debug.Log is a handy function for printing out messages for debugging your game.

To see Unity's console window, go to Window > Console. Alternatively in Windows, press Shift + Control + C, or in Mac, Shift + Command + C.



In the Debug.Log, “hitInfo.transform” returns the Transform component of the object that we hit. Then we simply access that transform's name.

Try using the script. Attach the script to your Player game object and run the game.

When you move your mouse around, the console should be printing out the name of whatever is at your crosshair.

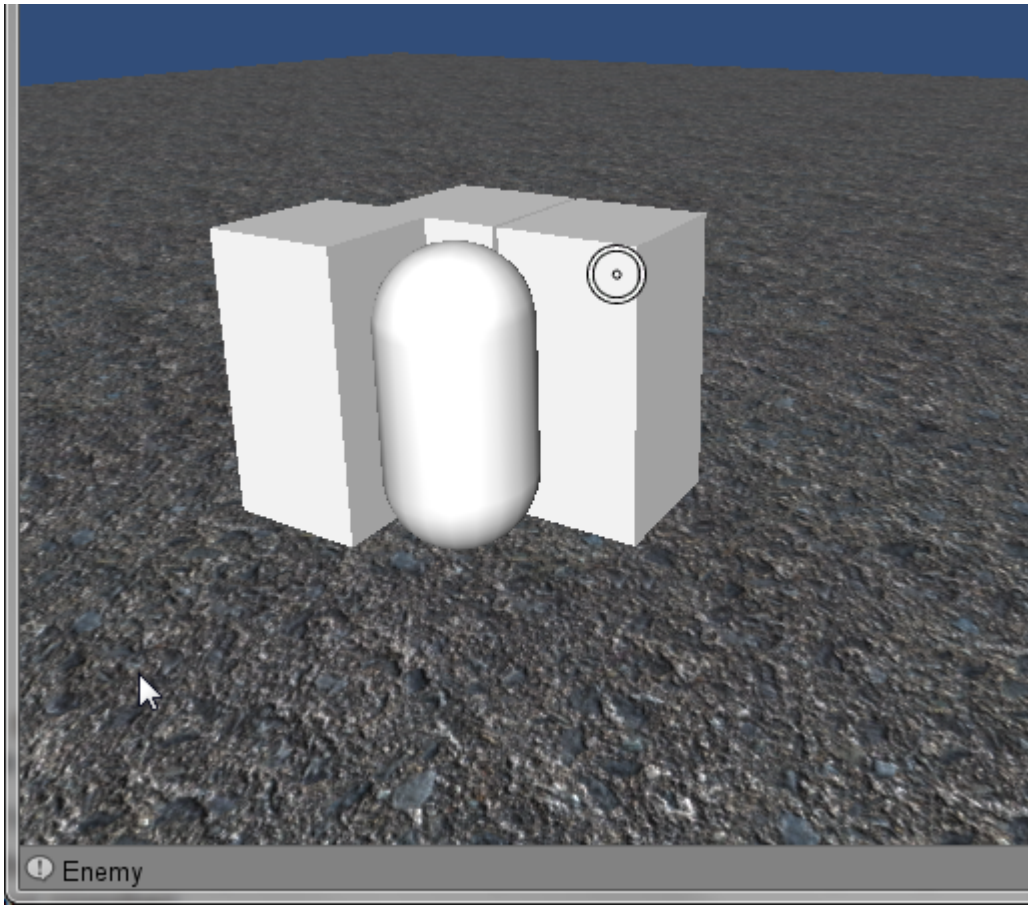


Illustration 1: You'll also see the latest console message at the lower left corner of your Unity window.

Now let's change the code. We shouldn't do raycasts always. We need to check first if the player wants to shoot.

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : MonoBehaviour
05 {
06     void Update()
07     {
08         if (Input.GetButtonDown("Fire1"))
09         {
10             Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
11             RaycastHit hitInfo;
12
13             if (Physics.Raycast(mouseRay, out hitInfo))
14             {
15                 Debug.Log(hitInfo.transform.name);
16             }
```

```
17          }
18      }
19 }
```

We now add an if statement around the raycasting process. GetButtonDown checks if a button has been pressed. “Fire1” by default is assigned to the left mouse button. So now, **the raycasting will only work if the player pressed the left mouse button**.

Dealing Damage

Now that the raycasting process is explained, let's finally add the part that will do the damaging.

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : MonoBehaviour
05 {
06      void Update()
07      {
08          if (Input.GetButtonDown("Fire1"))
09          {
10              Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
11              RaycastHit hitInfo;
12
13              if (Physics.Raycast(mouseRay, out hitInfo))
14              {
15                  Health enemyHealth = hitInfo.transform.GetComponent<Health>();
16                  if (enemyHealth != null)
17                  {
18                      enemyHealth.Damage(50);
19                  }
20              }
21          }
22      }
23 }
```

Instead of printing out the name of the object we hit, we now check if the object we hit has Health. If it has Health, we damage it by 50.

In line 15 we use GetComponent to get the Health. This can return null, for example, if the raycast hit the ground (which has no health). So we check first if its null and only then will we do the damaging.

Now let's change that constant value of 50 to something we can change easily:

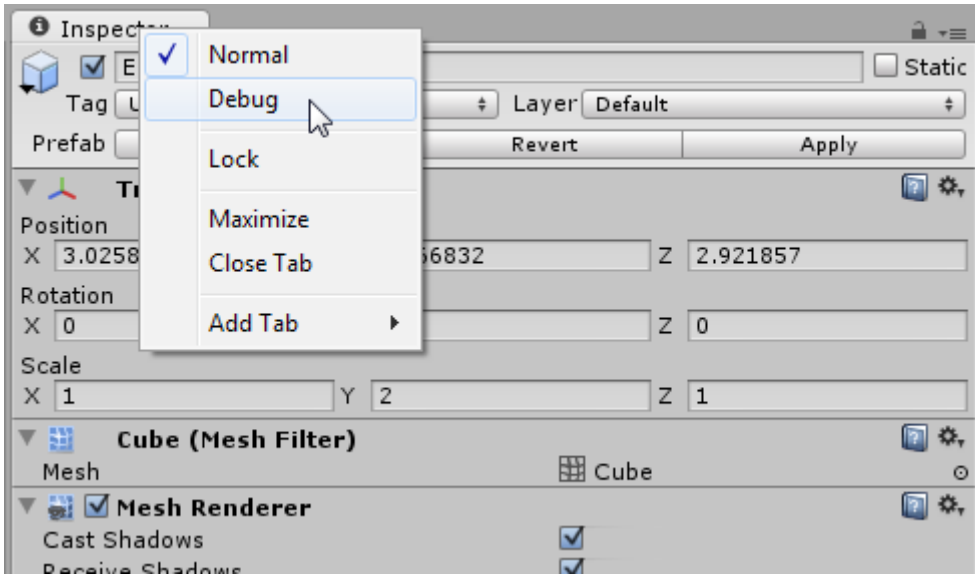
```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : MonoBehaviour
05 {
06      [SerializeField]
07      int _damageDealt = 50;
08
09      void Update()
10      {
11          if (Input.GetButtonDown("Fire1"))
12          {
13              Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
```

```
14              RaycastHit hitInfo;
15
16              if (Physics.Raycast(mouseRay, out hitInfo))
17              {
18                  Health enemyHealth = hitInfo.transform.GetComponent<Health>();
19                  if (enemyHealth != null)
20                  {
21                      enemyHealth.Damage(_damageDealt);
22                  }
23              }
24          }
25      }
26 }
```

We can test this script now. Go ahead and run the game. Then keep on clicking on the enemies. You won't see any change yet since we haven't coded what will happen when the enemies die, but you can check if the current health is indeed being lowered.

To see the enemy's current health, we'll put the Inspector into Debug Mode.

Select one of your enemies. Right-click on the Inspector tab.

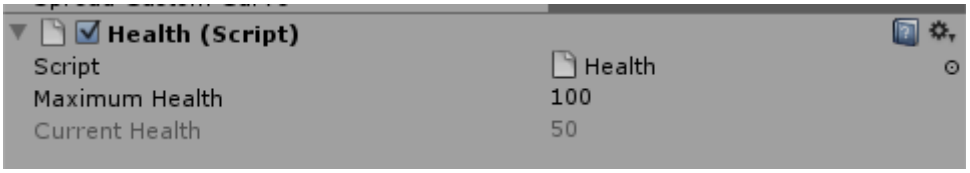


You'll see a menu pop up. Select “Debug”. Your Inspector will now be in Debug Mode.

The Debug Inspector shows you all of the scripts' variables, even ones that we didn't choose to be exposed to the Inspector normally.

We're using this so we can see the current health variable (which is otherwise hidden from us).

The current health variable will show up in grey to indicate that its a read-only value.



Now run the game if it isn't running already. Shoot the enemy that you have selected in the Inspector . You'll see its current health change accordingly.

Deletion On Death

Our enemies should fall down on the floor upon death, instead of still following the player. However we don't have a character set up in place yet, so we'll just delete the enemy when it dies.

So now go to your Health script and add this code:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Health : MonoBehaviour
05 {
06     [SerializeField]
07     int _maximumHealth = 100;
08
09     int _currentHealth = 0;
10
11     void Start()
12     {
13         _currentHealth = _maximumHealth;
14     }
15
16     public void Damage(int damageValue)
17     {
18         _currentHealth -= damageValue;
19
20         if (_currentHealth <= 0)
21         {
22             Destroy(gameObject);
23         }
24     }
25 }
```

Upon reaching zero health or lower, we simply delete the game object. This is done using the Destroy function.

Go ahead and run the game now. If you shoot at an enemy until it dies, it should disappear. (We'll add code to make the enemy fall on the ground once its using the zombie 3d model.)

Locking The Mouse Cursor

You may find it annoying that the mouse cursor may get to places while you're shooting. You can lock the cursor so it doesn't move as well as hide it from view.

Add this code to the RifleWeapon script:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : MonoBehaviour
05 {
06     [SerializeField]
07     int _damageDealt = 10;
08
```

```
09      void Start()
10      {
11          Screen.lockCursor = true;
12      }
13
14      void Update()
15      {
16          if (Input.GetButtonDown("Fire1"))
17          {
18              Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
19              RaycastHit hitInfo;
20
21              if (Physics.Raycast(mouseRay, out hitInfo))
22              {
23                  Health enemyHealth = hitInfo.transform.GetComponent<Health>();
24                  if (enemyHealth != null)
25                  {
26                      enemyHealth.Damage(_damageDealt);
27                  }
28              }
29          }
30      }
31 }
32
```

Screen.lockCursor is a boolean that you can set so the cursor gets locked (and it'll be hidden as well).

If you test the game it works, but you may find the mouse cursor appearing again after a while. Its a problem with how Windows manages the mouse cursor, but when you publish your game as an .EXE file, this problem won't happen.

Unfortunately, you may want the mouse cursor to show again when you finally want to quit the game. Here's an easy way: We'll make it that pressing Escape on the keyboard will show up the mouse again:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : MonoBehaviour
05 {
06      [SerializeField]
07      int _damageDealt = 10;
08
09      void Start()
10      {
11          Screen.lockCursor = true;
12      }
13
14      void Update()
15      {
16          if (Input.GetKey(KeyCode.Escape))
17          {
18              Screen.lockCursor = false;
19          }
20
21          if (Input.GetButtonDown("Fire1"))
22          {
```

```
23                Screen.lockCursor = true;
24                Ray mouseRay = Camera.main.ViewportPointToRay(new  Vector3(0.5f, 0.5f, 0));
25                RaycastHit hitInfo;
26
27                if  (Physics.Raycast(mouseRay, out hitInfo))
28                {
29                      Health enemyHealth = hitInfo.transform.GetComponent<Health>();
30                      if  (enemyHealth != null)
31                      {
32                            enemyHealth.Damage(_damageDealt);
33                      }
34                }
35          }
36      }
37 }
38
```

We've also made it that shooting will hide the mouse again, in case the mouse was made visible.

In Conclusion...

You've learned about quite a range of things just to get combat up and running: raycasts, destroying components, how to display messages to the console window, and locking the mouse cursor.

In the next lesson, we'll stop using boxes and capsules as our characters and start using our 3d graphics of the zombie and the player.