# Game Programming Using Unity 3D

# Lesson 15: Visual And Sound Effects


Ferdinand Joseph Fernandez

Chief Technological Officer, Dreamlords Digital Inc.

Admin and Co-founder, Unity Philippines Users Group

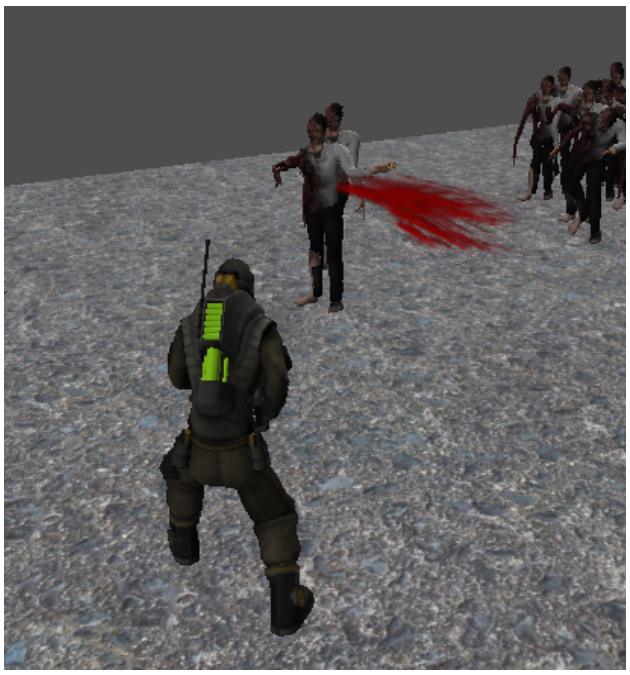
December 2011




# Overview

Again, we'll be continuing where we left off from the previous lesson. If you haven't done Lesson 14, you need to do it before you can continue here.

We'll be adding the missing particle effects and sound effects for our weapons.



# Adding Blood Effects When Zombie Is Hit

Go to the Lesson Assets folder and find the Packages folder. Inside should be a file named “BloodHit.unitypackage”. That package has a premade prefab for some blood particle effects. We'll be importing that.

So go to Unity and choose Assets > Import Package > Custom Package... and choose that Unity package.

It should give a “BloodHit” prefab in TheGame/Prefabs/ folder.

Go to your RifleWeapon script. That's the script that knows where to put blood effects. Add this code:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : MonoBehaviour
05 {
06     [SerializeField]
07     int _damageDealt = 10;
08
09     [SerializeField]
10     GameObject _hitEffectPrefab;
11
```

```
12      void Start()
13      {
14          Screen.lockCursor = true;
15      }
16
17      void Update()
18      {
19          if (Input.GetKey(KeyCode.Escape))
20          {
21              Screen.lockCursor = false;
22          }
23
24          if (Input.GetButtonDown("Fire1"))
25          {
26              Screen.lockCursor = true;
27              Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.
28                                  5f, 0));
29              RaycastHit hitInfo;
30
31              if (Physics.Raycast(mouseRay, out hitInfo))
32              {
33                  Health enemyHealth = hitInfo.transform.GetComponent<Health>();
34                  if (enemyHealth != null)
35                  {
36                      enemyHealth.Damage(_damageDealt);
37
38                      Vector3 hitEffectPosition = hitInfo.point;
39                      Quaternion hitEffectRotation = Quaternion.identity;
40                      Instantiate(_hitEffectPrefab, hitEffectPosition,
41                                          hitEffectRotation);
42                  }
43              }
44          }
45      }
46  }
```

The idea is we instantiate a new copy of the BloodHit prefab every time we damage something.

Now test it. Go to the Unity Editor window, wait for it to recompile, and drag the BloodHit prefab to the “Hit Effect Prefab” slot in the RifleWeapon script component.

Run the game and shoot some zombies. There should be some blood effects every time you hit one.

## *Fixing Blood Effects Rotation*

You'll notice the blood effects aren't facing the right direction all the time. We just need to supply the right rotation value.

Change hitEffectRotation to this code:

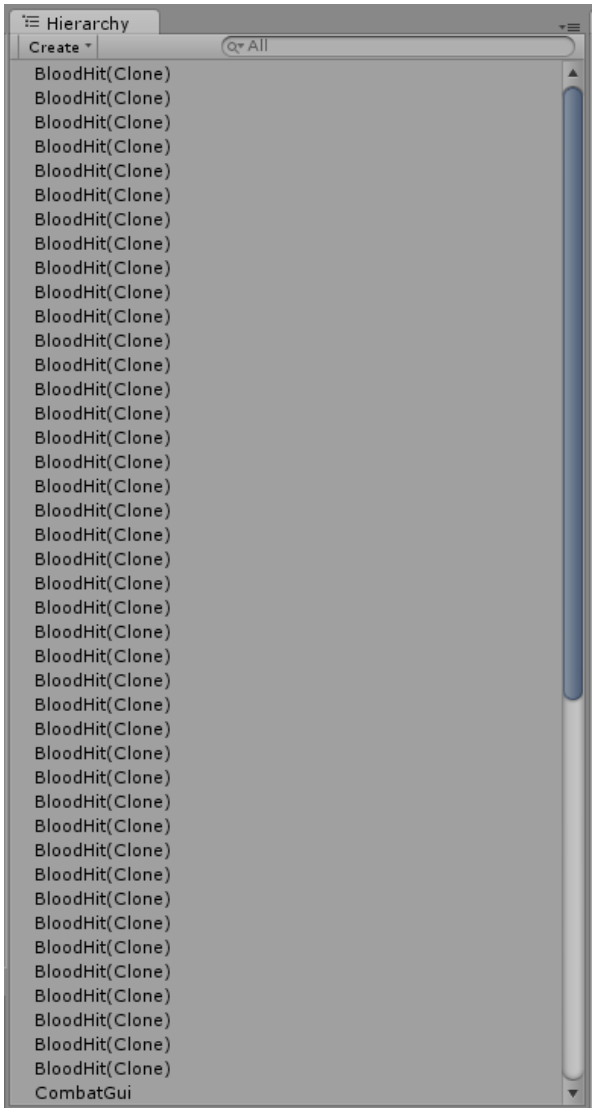
```
                                                    .
                                                    .
                                                    .
33                  if (enemyHealth != null)
34                  {
35                      enemyHealth.Damage(_damageDealt);
36
37                      Vector3 hitEffectPosition = hitInfo.point;
```

```
38                      Quaternion hitEffectRotation = Quaternion.FromToRotation(Vector3.forward,
                            hitInfo.normal);
39                      Instantiate(_hitEffectPrefab, hitEffectPosition, hitEffectRotation);
40                  }
41              }
42          }
43      }
44 }
```

Test it again. The blood effects should be facing in the right direction now. hitInfo.normal is the direction perpendicular to the surface that was hit by the raycast.

## *Cleaning Up Blood Effects Game Object*

When playing the game, look at the Hierarchy View. You'll notice the BloodHit game objects pile up on the list. This is because we aren't deleting them after they are used.




Select the BloodHit prefab from the Project View. Click on the triangle beside it. You'll see its composed of two children. Those two are particle emitters. They have Particle Animator components. If you look at those, they have their Autodestruct property checked.

This means those two child game objects of the BloodHit prefab automatically get deleted after use.

But the parent itself is left behind. Its that parent that we see piling up on the Hierarchy View and that's what we need to delete.

Here's what we'll do. There's no easy way to detect when a particle effect has finished from code. So what we'll do is we'll unparent the two children particle emitters from the parent, and we'll delete the parent.

Make a new C# script. Name it “EffectsCleanup”. Add this code:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class EffectsCleanup : MonoBehaviour
05 {
06     void Start()
07     {
08         foreach(Transform child in transform)
09         {
10             child.parent = null;
11         }
12         Destroy(gameObject);
13     }
14 }
```

We iterate over each child and set their parent to null. That effectively unparents them. Then we simply delete the current game object.

Find your BloodHit prefab in the Project View. Drag that EffectsCleanUp script (in the Project View also) to it.

Test the game again. Those BloodHit prefab copies shouldn't pile up on the Hierarchy anymore.

# Adding Blood Effect When Player Is Hit

Open the EnemyAttack script. We'll add the same code of instantiating the blood effect prefab when the zombie damages the player.

This time though, there's no data on where the hit exactly occurred; there's no raycast done when the zombie attacks.

So we'll compute the position in a different way.

Add this code:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class EnemyAttack : MonoBehaviour
05 {
06     float _nextTimeAttackIsAllowed = -1.0f;
07
08     [SerializeField]
09     float _attackDelay = 1.0f;
10
11     [SerializeField]
12     int _damageDealt = 5;
```

```
13
14      [SerializeField]
15      GameObject _hitEffectPrefab;
16
17      void  OnTriggerStay(Collider other)
18      {
19          if  (other.tag == "Player"  && Time.time >= _nextTimeAttackIsAllowed)
20          {
21              Health playerHealth = other.GetComponent<Health>();
22              playerHealth.Damage(_damageDealt);
23
24              Vector3 hitDirection = (transform.root.position - other.transform.position).normalized;
25              Vector3 hitEffectPosition = other.transform.position + (hitDirection * 0.01f) + (Vector3.up
                    * 1.5f);
26              Quaternion hitEffectRotation = Quaternion.FromToRotation(Vector3.forward, hitDirection);
27              Instantiate(_hitEffectPrefab, hitEffectPosition, hitEffectRotation);
28
29              _nextTimeAttackIsAllowed = Time.time + _attackDelay;
30          }
31      }
32 }
```

Instead of using a RaycastHit's normal, which we don't have, we compute it, in line 24. This is by simply computing the delta between the zombie and the player, then turning it into a direction.

The position computed here (line 25) starts at the player's position and nudged outwards closer to the zombie that attacked, and raised a bit so it doesn't show up on the ground.

Make sure to drag the BloodHit prefab to the Hit Effect Prefab slot of the Zombie's EnemyAttack script.

# Adding Sound On Hit

We'll play a sound every time the player or zombie gets hit.

Add this code to the Health script:

```
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      override public  string  ToString()
12      {
13          return  _currentHealth + " / "  + _maximumHealth;
14      }
15
16      public  bool  IsDead { get{ return  _currentHealth <= 0; } }
17
18      Renderer _renderer;
```

```
19
20      PlayerStats _playerStats;
21
22      [SerializeField]
23      AudioClip _hitSound;
24
25      void  Start()
26      {
27          _renderer = GetComponentInChildren<Renderer>();
28          _currentHealth = _maximumHealth;
29
30          GameObject player = GameObject.FindGameObjectWithTag("Player");
31          _playerStats = player.GetComponent<PlayerStats>();
32      }
33
34      public  void  Damage(int  damageValue)
35      {
36          _currentHealth -= damageValue;
37
38          if  (_currentHealth < 0)
39          {
40              _currentHealth = 0;
41          }
42          else
43          {
44              if  (_hitSound != null)
45              {
46                  audio.clip = _hitSound;
47                  audio.Play();
48              }
49          }
50
51          if  (_currentHealth == 0)
52          {
                                                    .
                                                    .
                                                    .
```

We simply play an audio clip using the Audio Source component that we have. Of course our game object needs to have an Audio Source component for this to work, so make sure your Player game object has an Audio Source. Add it by going to Component > Audio > Audio Source.

Do the same for the zombie enemy.

Your Lesson Assets folder should have a Sounds folder. Inside will be files named “player_killed_1.wav” and “ZombieDeath.wav”. We'll use them as our sounds. In your Project View, create a folder named “Sounds” inside the “TheGame” folder. Copy the .wav files there.

Then drag them to the corresponding Hit Sound slots in the Player's Health and the Zombie prefab's Health script components.

Test the game. They should play sounds now when hit.

# Randomizing Which Hit Sound To Play

You'll notice we have numerous player hit/killed sounds in the Lesson Assets folder. We can make use of all of them and have the game play one randomly selected among them.

Change the code in Health script to this:

```
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     override public string ToString()
12     {
13         return _currentHealth + " / " + _maximumHealth;
14     }
15
16     public bool IsDead { get{ return _currentHealth <= 0; } }
17
18     Renderer _renderer;
19
20     PlayerStats _playerStats;
21
22     [SerializeField]
23     AudioClip[] _hitSounds;
24
25     void Start()
26     {
27         _renderer = GetComponentInChildren<Renderer>();
28         _currentHealth = _maximumHealth;
29
30         GameObject player = GameObject.FindGameObjectWithTag("Player");
31         _playerStats = player.GetComponent<PlayerStats>();
32     }
33
34     public void Damage(int damageValue)
35     {
36         _currentHealth -= damageValue;
37
38         if (_currentHealth < 0)
39         {
40             _currentHealth = 0;
41         }
42         else
43         {
44             if (_hitSounds != null && _hitSounds.Length > 0)
45             {
46                 AudioClip soundToUse = _hitSounds[Random.Range(0, _hitSounds.Length)];
47                 audio.clip = soundToUse;
48                 audio.Play();
49             }
50         }
```

```
51
52          if (_currentHealth == 0)
53          {
                                                    .
                                                    .
                                                    .
```

Now we use an array of AudioClips. One of them gets randomly chosen in line 46, which will be the one played.

Import the other player killed sounds and re-add them to the Hit Sounds slot of the Health script component. Re-add the zombie hit sound since it got reset.

# Adding Sound On Death

Adding sound when killed works pretty much the same way. We just play a sound at the proper moment.

```
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      override public string ToString()
12      {
13          return _currentHealth + " / " + _maximumHealth;
14      }
15
16      public bool IsDead { get{ return _currentHealth <= 0; } }
17
18      Renderer _renderer;
19
20      PlayerStats _playerStats;
21
22      [SerializeField]
23      AudioClip[] _hitSounds;
24
25      [SerializeField]
26      AudioClip _deathSound;
27
28      void Start()
29      {
30          _renderer = GetComponentInChildren<Renderer>();
31          _currentHealth = _maximumHealth;
32
33          GameObject player = GameObject.FindGameObjectWithTag("Player");
34          _playerStats = player.GetComponent<PlayerStats>();
35      }
36
```

```
37      public void Damage(int damageValue)
38      {
39          _currentHealth -= damageValue;
40
41          if (_currentHealth < 0)
42          {
43              _currentHealth = 0;
44          }
45          else
46          {
47              if (_hitSounds != null && _hitSounds.Length > 0)
48              {
49                  AudioClip soundToUse = _hitSounds[Random.Range(0, _hitSounds.Length)];
50                  audio.clip = soundToUse;
51                  audio.Play();
52              }
53          }
54
55          if (_currentHealth == 0)
56          {
57              if (_deathSound != null)
58              {
59                  audio.clip = _deathSound;
60                  audio.Play();
61              }
62
63              Animation a = GetComponentInChildren<Animation>();
64              a.Stop();
                                                    .
                                                    .
                                                    .
```

You can use the “you are dead dead dead 2.wav” from the Lesson Assets folder as the death sound for the player.

Your Lesson Assets folder has a file named “23289__progic35__resocopter-fastE.wav” which is suitable as the sound for the helicopter.

Try using it as the sound for your helicopter.

# In Conclusion...

We've added some visual and sound effects in our game. These are some simple things that don't affect the game mechanics very much but they are important in any game. Adding effects that help immerse the player in the experience improves the overall quality of your game.